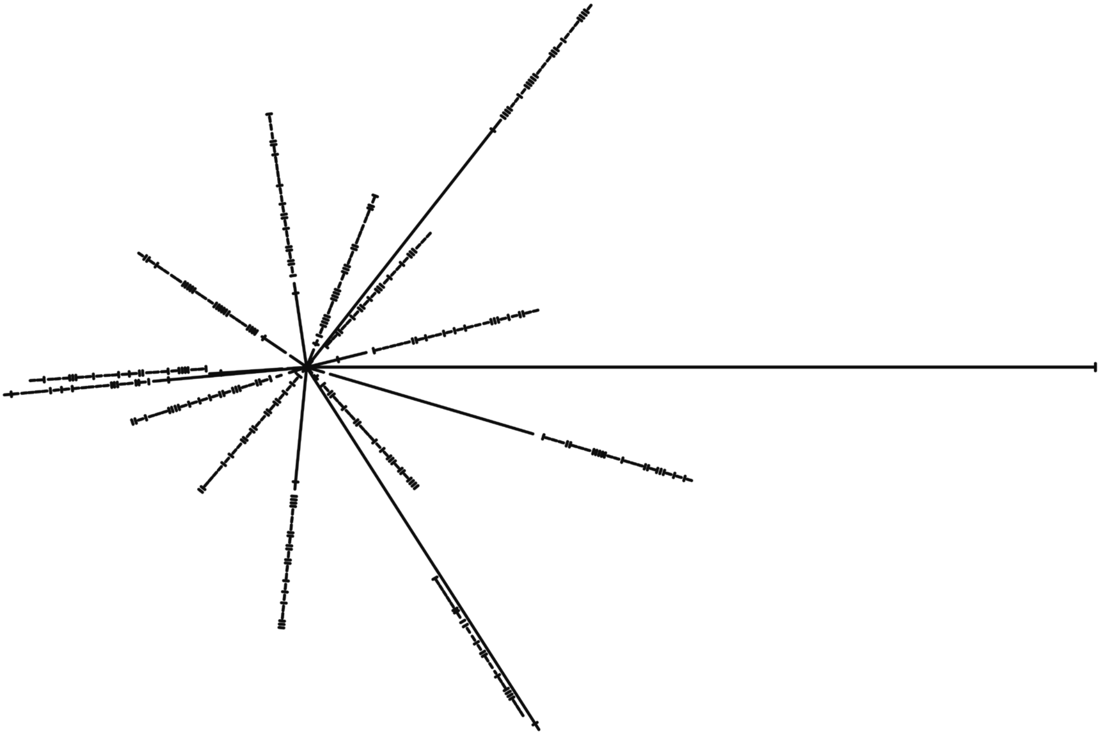
Doctoral Thesis in Computer Science

To Secure a Flow: From Specification to Enforcement of Information Flow Control

AMIR M. AHMADIAN

KTH ROYAL INSTITUTE OF TECHNOLOGY



To Secure a Flow: From Specification to Enforcement of Information Flow Control

AMIR M. AHMADIAN

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology,
is submitted for public defence for the Degree of Doctor of Philosophy on Tuesday the 11th March 2025,
at 9:00 a.m. in Kollegiesalen, Brinellvägen 6, Stockholm.

Doctoral Thesis in Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2025

Cover page photo: Voyager Pulsar Map - NASA Jet Propulsion Laboratory

TRITA-EECS-AVL-2025:22
ISBN 978-91-8106-199-4

Printed by: Universitetsservice US-AB, Sweden 2025

“*If life transcends death,*
then I will seek for you there.
If not, then there too.”

Abstract

The use of software has become increasingly prevalent, affecting nearly every aspect of our daily lives. In this landscape, ensuring the security of software systems is more critical than ever, as vulnerabilities can lead to significant social and financial consequences. Information flow control is a research area focused on developing methods and techniques to provide strong security guarantees against software attacks and vulnerabilities. Information flow control achieves this by tracking how information flows within a program, ensuring that sensitive data does not reach unauthorized outputs. This process can be challenging as it requires precisely defining the software system's security policy and developing mechanisms to enforce that policy against different types of attackers with varying capabilities.

In this thesis, we examine language-based approaches to enforcing information flow control in software systems, with a focus on defining appropriate security policies, attacker models, and enforcement mechanisms to proactively secure modern software systems. The thesis contributes to the state of the art of information flow security in several directions, both theoretical and practical, by investigating four key research questions: defining non-trivial security policies for real-world scenarios, developing appropriate attacker models, creating mechanisms to enforce information flow security conditions, and applying language-based techniques to real-world programming languages. On the policy specification side, we provide a knowledge-based security framework capable of expressing many variants of dynamic policies as well as the Determinacy Quantale, a new semantic model for expressing disjunctive policies in database-backed programs, focusing on the conflict-of-interest classes. We examine the role of attackers in defining security conditions, specifically two types of active attackers and three types of passive attackers with various degrees of capabilities. Moreover, we investigate enforcement mechanisms, such as security type systems and symbolic execution, developed to statically enforce various information flow security policies. Finally, to demonstrate the applicability of language-based approaches in real-world programs, we implement and evaluate the proposed enforcement mechanisms in the programming languages Java and P4.

Sammanfattning

Användningen av mjukvara har blivit alltmer utbredd och påverkar i stort sett alla aspekter av våra dagliga liv. I detta sammanhang är det viktigare än någonsin att säkerställa säkerheten i mjukvarusystem, eftersom sårbarheter kan leda till betydande sociala och ekonomiska konsekvenser. Informationsflödeskontroll är ett forskningsområde som fokuserar på att utveckla metoder och tekniker för att tillhandahålla starka säkerhetsgarantier mot mjukvaruattacker och sårbarheter. Informationsflödeskontroll uppnår detta genom att spåra hur information flödar i ett program och säkerställa att känslig data inte når obehöriga utdata. Denna process kan vara utmanande eftersom den kräver en exakt definition av mjukvarusystemets säkerhetspolicy och utveckling av mekanismer för att upprätthålla policyn mot olika typer av angripare med varierande förmågor.

I denna avhandling undersöker vi språkbaserade tillvägagångssätt för att upprätthålla informationsflödeskontroll i mjukvarusystem, med fokus på att definiera lämpliga säkerhetspolicies, angriparmodeller och mekanismer för verkställande för att proaktivt säkra moderna mjukvarusystem. Avhandlingen bidrar till den aktuella forskningen inom informationsflödesäkerhet på flera sätt, både teoretiska och praktiska, genom att undersöka fyra centrala forskningsfrågor: att definiera icke-triviala säkerhetspolicies för verkliga scenarier, att utveckla relevanta angriparmodeller, att skapa mekanismer för att upprätthålla villkor för informationsflödesäkerhet samt att tillämpa språkbaserade tekniker på verkliga programmeringsspråk. När det gäller policy-specifikation presenterar vi ett kunskapsbaserat säkerhetsramverk som kan uttrycka många varianter av dynamiska policies samt Determinacy Quantale, en ny semantisk modell för att uttrycka disjunktiva policies i databasstödda program, med särskilt fokus på intressekonfliktklasser. Vi undersöker angriparens roll i att definiera säkerhetsvillkor, särskilt två typer av aktiva angripare och tre typer av passiva angripare med olika grad av förmågor. Vi undersöker mekanismer för efterlevnad, såsom säkerhetstypsystem och symbolisk exekvering, som utvecklats för att statiskt upprätthålla olika informationsflödessäkerhetspolicies. Slutligen, för att demonstrera användbarheten av språkbaserade metoder i verkliga program, implementerar och utvärderar vi de föreslagna mekanismerna på språk Java och P4.

List of Papers

A. ***Language Support for Secure Software Development with Enclaves***
Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi
Proceedings of the 34th Computer Security Foundations Symposium (CSF), 2021

B. ***Enclave-Based Secure Programming with J_E***
Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi
Proceedings of IEEE Secure Development Conference (SecDev), 2021

C. ***Dynamic Policies Revisited***
Amir M. Ahmadian and Musard Balliu
Proceedings of the 7th European Symposium on Security and Privacy (EuroS&P), 2022

D. ***Disjunctive Policies for Database-Backed Programs***
Amir M. Ahmadian, Matvey Soloviev, and Musard Balliu
Proceedings of the 37th Computer Security Foundations Symposium (CSF), 2024

E. ***Securing P4 Programs by Information Flow Control***
Anoud Alshnakat, Amir M. Ahmadian, Musard Balliu, Roberto Guanciale, and Mads Dam
Manuscript

Acknowledgement

Five years ago, on a cold autumn evening in 2019, I set out for Sweden to study for a PhD, unaware of the roads life would unfold before me. I could not have imagined the lessons, the friendships, the losses, and the quiet moments of doubt that shaped this journey. To all those who offered their support and encouragement along the way, who made this journey a possibility, I offer my deepest gratitude.

To my supervisor, Musard Balliu — thank you!
For your guidance, your patience, and the knowledge you so generously shared. For the conversations, the laughter, the BBQs, and the incredible people you introduced me to. I have learned so much from you, both in research and in life.

Special thanks to my other supervisor, Dilian Gurov. Although I never had the chance to collaborate with you on research, I had the privilege of knowing you and working alongside you in the Computer Science Doctoral Programme. I'm thankful for your kindness, your support, and the care you have shown — not just to me, but to all the PhD students in the CS programme.

I would like to express my gratitude to my opponent Limin Jia, and the grading committee members Aslan Askarov, Ioana Boureanu, and Alejandro Russo for reviewing this thesis and evaluating my work. Special thanks to Roberto Guanciale for serving as the chair of my defense, Philipp Haller for being the advance reviewer of this thesis, and Romy and Oscar for helping me with the Swedish abstract.

My co-authors Anoud Alshnakat, Matvey Soloviev, Aditya Oak, Guido Salvaneschi, Roberto Guanciale, Musard Balliu, and Mads Dam, thank you for the discussions, the debates, and the opportunity of learning alongside you.

To Sofia, Xiaolin, Honglian, Priyanka, Frank, Naila, Mikhail, Ioana, SiKai, Antonio, Aman, Matvey, Khashayar, Jesper, Mojtaba, Javier Cabrera, Christian, Eric, Monica, André, Javier Ron, Shuangjie, Sijing, Carmine, Ning, Serena, Camilla, Sakib, Kilian, Vivi, Tianyi, Arve, Yuxin, Manon, Marcus, Henrik, Deepika, and all the friends and colleagues in the Division of Theoretical Computer Science, thank you for the talks, the fikas and — as someone recently pointed out — socializing with me in the kitchen.

Many thanks to my friends in the EECS PhD Council, especially Susanna, Saumey, Joel, Jura, Simon, Fereidoon, Alireza, and Sina, with whom I worked to improve the experience of PhD students in EECS and alongside whom I learned so much about the inner workings of KTH.

I'm endlessly grateful to Romy, Ånoud, Åndreas, and Raphina for always being there — for listening, for understanding, and for tolerating me through the ups and downs. I cherish your presence in my life.

To my Iranian friends in Sweden, Ehsan, Maryam, Yasmin, Mohammad, Mojtaba, Maryam, Sina, Mahmoud, Sahba, Amir, and Vahid, thank you for the camaraderie, the support, and the memories that made the long nights and dark winters of Sweden, ever so slightly, more bearable.

Last but certainly not least, my deepest gratitude goes to my family; without their support, this journey would have been impossible.
To Maman, Baba, Azadeh, Sara, and Hamze, for their endless love, encouragement, and unwavering support — your belief in me, even when I doubted myself, has been my greatest source of strength. Thank you for everything.

As I reach the end of this path, I do so with both joy and longing. There is a quiet space in all of this that belongs to my dad and my grandmother, whom I lost while on this journey, far from home. Your absence is a void that words cannot fill, yet your love, kindness, and support continue to shape me. Though you are not here to see this moment, you are with me in every word, every page, and every step forward.

Acronyms

List of commonly used acronyms:

BLP	Bell–LaPadula model
DAC	Discretionary access control
DLM	Decentralized Label Model
DL	Determinacy lattice
DQ	Determinacy quantale
IFC	Information flow control
LBS	Language-based security
LoI	Lattice of information
MAC	Mandatory access control
NSU	No-sensitive-upgrade
QoI	Quantale of information
SME	Secure multi-execution
SOT	Symbolic output tree
TINI	Termination-insensitive noninterference
TSNI	Termination-sensitive noninterference

Contents

Thesis

Papers

Bibliography

Thesis

1 | Introduction

> “*The real world was full of larger structures, smaller structures, simpler and more complex structures than the tiny portion comprising sentient creatures and their societies, and it required a profound myopia of scale and similarity to believe that everything beyond this shallow layer could be ignored.*”
>
> – **Greg Egan**, Diaspora

Software has become an indispensable component of modern life, influencing nearly every aspect of our daily life. From the smartphones in our pockets to the sophisticated systems supporting critical infrastructure, software has reshaped the way we communicate, work, and interact with the world. However, as software integrates deeper into our lives, the need to ensure its security becomes ever more critical. Vulnerabilities such as EternalBlue [141], Log4Shell [191], and Heartbleed [107] demonstrate the severe implications of software flaws and the significant damage they can cause.

To mitigate such risks and ensure software security, developers and organizations employ various practices. The history of these security measures mirrors the evolution of software systems, adapted to meet the challenges of new technologies. In the 1960s, during the mainframe era, access control became one of the first measures to ensure security. Early systems like MIT’s Compatible Time-Sharing System introduced user accounts and passwords to restrict unauthorized access. As computers began serving multiple users, researchers developed models such as the Bell-LaPadula (BLP) [1] model to enforce stricter security and provide formal security guarantees in highly sensitive systems. Models such as BLP [1] and Biba [6] became standards for defining standard security practices in military and governmental systems. Efforts like the U.S. Department of Defense’s Orange Book [10] established guidelines for building and evaluating secure software, emphasizing on principles such as access control and least privilege.

The rise of personal computers in the 1980s and the growth of network-connected systems in the 1990s shifted the focus from isolated mainframe systems to protecting software running on millions of individual devices. During this period, secure software design principles gained prominence, aiming to reduce the risk of exploitable flaws.

Problems such as buffer overflow and privilege escalation became common, which led to the development of methods such as bounds checking, stack canaries, and stricter enforcement of the principle of least privilege. During this era, malware often spread through infected files or email attachments and targeted computers, leading to a surge in the popularity of antivirus software.

As the Internet became more popular, security measures also evolved to protect the increasingly complex web applications. Unlike traditional software, web applications included many dynamically changing features and were typically made up of a mashup of various code from many sources. Input validation and sanitization became critical to defend against attacks like SQL injection and cross-site scripting (XSS). Authentication and session management techniques gained prominence for web applications that relied on user login systems. Measures such as two-factor authentication and secure cookie handling were adopted to prevent unauthorized access, and encryption techniques, such as SSL/TLS protocols, became standard to ensure secure data transmission.

In the smartphone era, new threats arose due to the unique features of mobile apps. Learning the lessons of the PC era, mobile operating systems were equipped with strict permission systems to ensure apps only have access to the data they need, and this access was transparent to the end-user. Technologies such as app sandboxing were developed to limit the applications to their own run-time environment, limiting the damage caused by issues such as buffer overflow and malware.

Modern software is inherently complex, with applications made up of interconnected components and from potentially untrusted sources. The dynamic nature of software development, driven by continuous updates, addition of new features, and external dependencies, makes ensuring security ever more challenging. Traditional security practices, such as testing, are less effective in this dynamic environment, and while practices such as secure coding and penetration testing are helpful, they often do not cover every threat, especially in today's large-scale systems.

Given the various threats and vulnerabilities discovered every day, it is clear that we need to shift from reactive security measures to proactive ones, providing provable security guarantees against the challenges of modern-day software systems. To achieve this goal, we need to define precisely what we mean by securing software. This requires defining suitable security policies to secure the system against specific attacker types, and devising practical mechanisms to enforce these security policies. This thesis is one step towards achieving this goal.

Language-based information flow control. Information flow control (IFC) is an area of software security that tracks how the information propagates within a program. This propagation typically occurs from program inputs (sources) to outputs (sinks), and generally, ensuring security means preventing undesirable flows e.g. from *sensitive* data sources to *public* sinks.

To verify a program's security via information flow control, one must formally model the program and its environment and provide a precise definition of what *security* means. These ingredients of information flow control, which we discuss in more detail in Chapter 3, are as follows:

- **System model**: Provides a precise model encompassing the possible behaviors of the program.
- **Attacker model**: Models the observations of the attacker through sinks that they can see, and specifies their capabilities, such as whether they only observe the program execution or try to actively interfere with it.
- **Security policy**: Labels the sources and sinks, and specifies which flows are considered secure.
- **Security condition**: A condition that states whether the system model complies with the security policy for a given attacker.

Once the security requirements of the program have been formally defined using these ingredients, an **enforcement mechanism** is used to enforce the security condition in practice. One approach to enforce information flow control is called Language-Based Security (LBS). Language-based security operates on the source-level of the program and leverages programming language analysis and verification techniques to enforce the security requirements defined in the IFC specifications. Through these mechanisms, language-based security can provide formal security guarantees for information flow policies and ensure that programs behave securely against various attacker models.

1.1 Research Questions

The primary objective of this thesis is to investigate challenges related to various aspects of language-based information flow control. This research seeks to advance the state of the art with regard to defining information flow control **policies** for appropriate **attacker models** in real-world settings, developing novel **security conditions** to verify the security of software systems, and proposing **enforcement mechanisms** to enforce these security conditions. This objective leads to the following research questions:

> **RQ1**: How can non-trivial security policies be defined to effectively address specific real-world scenarios?
>
> **RQ2**: What are the appropriate attacker models for scenarios where simple attacker models fail to adequately account for the full range of security threats faced by the system?

RQ3: How can we develop mechanisms to enforce realistic information flow security conditions in a sound and precise manner?

RQ4: How can information flow control techniques be effectively and manageably applied to real-world programming languages?

1.2 Research Methodology

This thesis adopts a deductive research methodology to develop and validate new language-based methods for information flow control. We follow a three-step process: (1) specification, (2) enforcement, and (3) application. The process begins by formalizing the programming language, which serves as the foundation for deriving a *program model*. Subsequently, we define the *security requirements* of the system by considering aspects such as attacker models and security policies. The outcome of this step is a security condition used to verify the desired security properties of the program model. *Language-based mechanisms* are designed and developed to enforce the security condition. *Formal proofs* establish the soundness of the developed mechanisms, ensuring their adherence to the security condition. Finally, these mechanisms are evaluated through practical *implementation* and case studies.

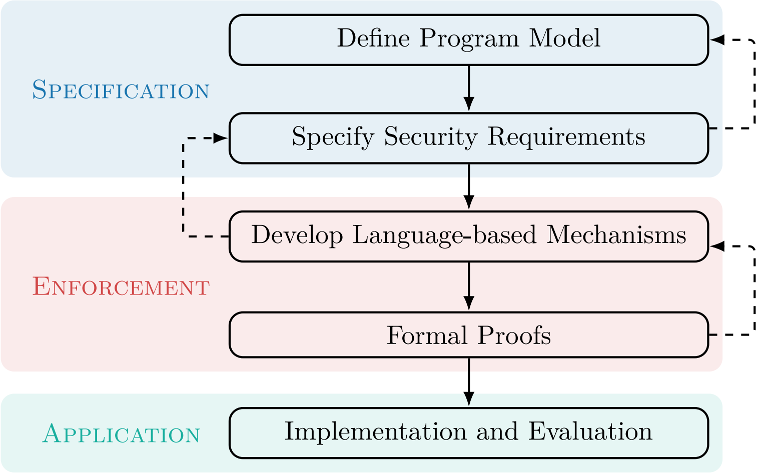

Specification
Define Program Model
Specify Security Requirements
Enforcement
Develop Language-based Mechanisms
Formal Proofs
Application
Implementation and Evaluation

Figure 1.1: Research methodology

Figure 1.1 illustrates the overall methodology employed in this thesis. We briefly outline the steps that define each component of this methodology.

Define program model. The first step is to develop a program model. This includes defining constructs for specifying the program's structure and behavior.

We formalize the programming language by defining its syntax and semantics, from which we can develop a program model for the programs written in that language.

Specify security requirements. We specify the security requirements of the system by defining attacker models relevant to the potential security threats. Information flow security policies are defined to state how the data may flow within the program. This formalization provides a rigorous foundation for developing the security condition, which is used to verify whether the given program model is secure under the defined attacker model and security policy. This step is iterative, as both the program model and security requirements may need to be refined multiple times to achieve an acceptable model.

Develop language-based mechanisms. With the program model and security condition in place, language-based mechanisms are developed to enforce the security condition. At this stage, the security condition is translated into formal constraints, such as typing rules or logical assertions, that the mechanisms must enforce. This step bridges the gap between abstract security requirements and concrete implementation requirements by clearly defining what constitutes a secure program. This step also involves an iterative interaction with the previous step, as adjustments to the security requirements and their definitions may be necessary to align them with the developing mechanism.

Formal proofs. To ensure the language-based mechanisms work as intended, formal proofs are provided to demonstrate the soundness of the proposed mechanisms with respect to the security requirements. A mechanism is sound if it only accepts programs that are indeed secure. This is irrespective of whether all secure programs are accepted by the mechanism, as, due to the overapproximation inherent in many language-based mechanisms, they sometimes reject secure programs. It is ideal for a language-based mechanism to guarantee soundness while rejecting as few secure programs as possible.

These proofs rely on deductive reasoning on the program model and the security condition, establishing that the developed mechanisms meet their theoretical objectives. This step closely follows the development of the language-based mechanisms. Occasionally, during the proof process, it becomes necessary to revisit and modify the mechanisms to ensure they can be formally proved.

Implementation and evaluation. Applying the theoretical mechanisms to practical programs is the final step in this process. This involves implementing the proposed mechanisms in prototype tools and evaluating them on practical case studies. This step is separate from the previous ones. The previous steps, focused only on theoretical aspects, generally model the core of a real-world programming language and propose and prove the language-based mechanisms on this core. At the application step, this formalized core is adapted and extended, enabling the application of the theoretical results of previous steps to practical programs.

1.3 Contributions

Figure 1.2 provides an overview of the included papers and their relation to the research questions. We briefly outline these contributions in Chapter 4. For detailed explanations, readers are referred to the full papers in their respective chapters.

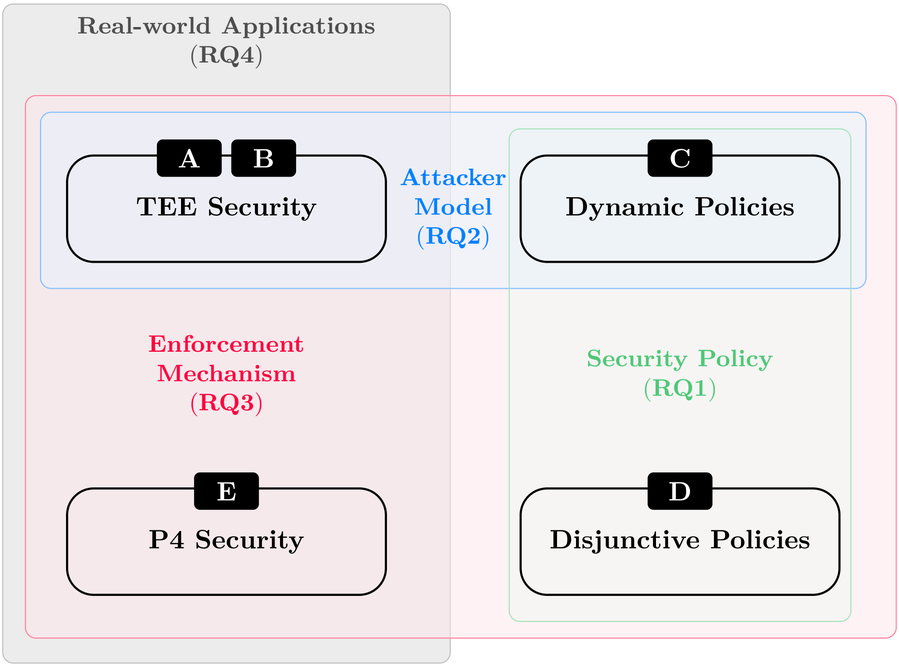


Figure 1.2: The papers included in the thesis

For **RQ1**, in Paper C, we investigate scenarios where the security policy is dynamic and can change during the execution of the program. By revisiting the existing security conditions for dynamic policies, we highlight the critical role of the attacker in defining the security condition. Consequently, we propose a novel knowledge-based security framework that establishes a clear connection between dynamic policies and attacker models.

In Paper D, we explore disjunctive policies, which address the problem of defining policies that restrict access to information based on conflict-of-interest classes. They allow the flow of information only if the observer has not accessed any conflicting data from other classes. Given the relevance of this class of policies to database applications, in Paper D, we focus on database-backed programs. We propose the Determinacy Quantale as a new semantic model to express disjunctive policies in terms of database views, and develop a security condition to verify the security of database-backed programs with respect to this class of policies.

For **RQ2**, we investigate active attackers in the context of trusted execution environments in Papers A and B. In contrast to passive attackers, who only observe parts of the execution of the program, active attackers actively interfere with the normal execution of the program through data or control flow modifications. A trusted execution environment (also called an enclave) is a secure area within a device's main processor that ensures sensitive data and operations are protected, isolated, and executed securely, even in the presence of a compromised operating system. Active attackers are a natural choice here because the host of the trusted execution environment may be untrusted and potentially malicious. Since the host controls the execution environment, it is important to consider attackers that are more powerful than a simple passive attacker.

We investigate security against two types of active attackers: one that manipulates the data passed to the enclave, and another that manipulates the control flow of the program by calling any enclave method as many times and in any order. In Papers A and B, we propose a security type system that statically guarantees robustness against these types of active attackers, ensuring that active attackers cannot learn more information than a passive attacker.

As mentioned before, our security framework for dynamic policies, introduced in Paper C, also accommodates various attacker models. We investigate three types of passive attackers: perfect recall attackers (who remember all past information), bounded memory attackers (who have limited memory), and forgetful attackers (who forget their past knowledge when the policy changes).

For **RQ3**, we investigate various static enforcement mechanisms, each tailored to specific use cases. Papers A and B propose a security type system that incorporates labelings for confidentiality and integrity. In Paper E, we develop a security type system augmented with interval analysis. This interval analysis tracks the range of possible values for each type, enabling the expression and enforcement of data-dependent security policies. We formally prove that this permissive yet sound type system enforces noninterference in P4 programs.

To enforce disjunctive policies, in Paper D, we propose a *disjunctive* type-based dependency analysis that represents database dependencies of each possible run of the program as a set of queries. The result of this analysis is a set of sets of queries, capturing the dependencies of all possible runs of the program. To ensure security with respect to disjunctive policies, we verify that each set of dependencies is in line with the constraints of the conflict-of-interest classes described by the policy.

We adapt a symbolic execution approach to enforce dynamic policies in Paper C. We symbolically execute the program and capture the program model as a symbolic output tree, where each node contains information about observable outputs, path conditions, and the security policy active at the time the output was produced. Using this information, we verify the program's security compliance with respect to the active policy for each execution point in the program.

To address **RQ4**, we develop static enforcement mechanisms for programming languages Java and P4. In Paper E we implement a security type system for the P4 programming language, We use this type system to statically enforce data-dependent security policies on the input and output packets of a programmable switch, ensuring the security of P4 programs.

To verify the security of a Java program in the presence of dynamic policies, in Paper A, we introduce DynCoVer. It relies on symbolic execution to analyze a Java program and produce a symbolic output tree, encoding observable outputs and the active policy. This symbolic output tree is later used, in combination with Z3 solver, to verify the security of the program with respect to dynamic security policies.

Furthermore, in Paper A, we develop a programming model and a security type system for enclave-based Java programs. In this approach, user-defined annotations on standard Java code are used to automatically partition the program into enclave and non-enclave parts, and our type system ensures that this partitioned program enforces robustness.

1.4 Outline

Chapter 2 provides an overview of the field of software security in general, emphasizing how security guarantees can be described in a formal manner and briefly discussing the limitations of existing solutions. Chapter 3 introduces language-based information flow control, which is the primary focus of the thesis. It examines how information flow requirements can be specified, enforced, and applied using language-based techniques, and highlights the thesis's contributions to the current state of the art. Chapter 4 presents a summary of the publications included in the thesis, along with an overview of the author's contributions. Chapter 5 concludes the thesis with a discussion on potential future work and directions for further research.

2 | Software Security

> “*She had racked her piecemeal recollection of her species’ history and found only a hierarchy of destruction: of her species devastating the fauna of planet Earth, and then turning on its own sibling offshoots, and then at last, when no other suitable adversaries remained, tearing at itself. Mankind brooks no competitors, She has explained to them — not even its own reflection.*”
>
> – **Adrian Tchaikovsky**, Children of Time

The recognition of the importance of formalizing software security dates back to the early 1970s, a period during which several research initiatives focused on this area were funded by the United States Department of Defense. During this time, significant progress was made in understanding how to formally define and evaluate the security properties of computing systems. A key contribution to this effort was the development of the so-called *Orange Book* (formally known as the Trusted Computer System Evaluation Criteria) [10], which established a framework for evaluating and certifying the security of computer systems. This period also saw the emergence of formal verification methods, which aimed to bridge the gap between theoretical security models and practical systems.

While traditional software engineering methods focus on developing functional software that meets user requirements, formalization goes further by ensuring that the system’s behavior is secure under all conditions. The software development process typically involves stages such as requirements gathering, design, implementation, testing, and maintenance. Formal methods contribute at many of these stages by providing formal specifications of the system, defining security requirements and policies, and proving the correctness of the system.

Security policy. The first step in having any meaningful formalization of security is to identify what constitutes sensitive information within the system and needs protection. A security policy provides this definition by specifying the sensitive and non-sensitive information. It serves as a baseline against which the security of the system can be evaluated.

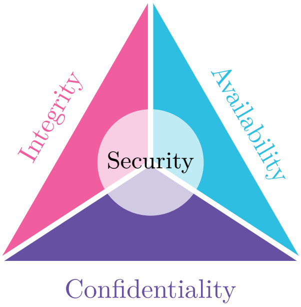


Figure 2.1: CIA triad

Traditionally, the security properties of a system are divided into three categories: confidentiality, integrity and availability, often dubbed as the CIA triad (depicted in Figure 2.1).

- **Confidentiality** focuses on preserving the secrecy of sensitive information. In this context, information is classified into various sensitivity levels, such as *secret* and *public*. Confidentiality requirements stipulate that sensitive information should not be accessible to unauthorized entities (e.g. users or processes). Here, *accessible* refers to the ability to observe, read, or copy the information, while an unauthorized entity may technically overwrite sensitive information without violating confidentiality.

- **Integrity** focuses on the trustworthiness of information. Integrity is often regarded as the dual of confidentiality, with information classified into trustworthiness levels, such as *trusted* and *untrusted*. Integrity requirements state that information should remain trustworthy throughout its lifecycle. This means that unauthorized entities cannot create, modify, or tamper with the information in any way.

- **Availability** guarantees that the system is reliable, and information is accessible by an *authorized* entity when requested, ensuring that services are not denied to authorized users, even in the face of disruptions or attacks.

While each of these principles addresses a distinct dimension of the overall security of the system, they lack formal definitions and are *not entirely orthogonal* to one another, meaning that they can sometimes overlap or even conflict in certain scenarios. For example, confidentiality may be enforced through integrity violations, where corrupting or tampering with *secret* data makes it invalid, thus preserving the secrecy of the original data. Similarly, confidentiality can also be enforced through availability violations; for instance, denying access to *secret* data altogether makes it inaccessible to any user, thereby protecting the confidentiality of the data.

Attacker model. An important factor in guaranteeing the system's security is defining who (or what) the system is being protected against. This is known as the attacker model and typically involves specifying the scope of the actions and the capabilities of the attacker attempting to compromise the system. An attacker's capabilities may range from passively observing the program's behavior to actively interfering with its execution by manipulating data or code. Accurately defining the attacker — whether it is one measuring small timing deviations in program execution or one modifying a web page's behavior through code injection — must correspond to the specific requirements of the program and the context in which it is deployed. This alignment ensures that the implemented security measures are both theoretically sound and practically effective.

A well-known example of an attacker model in network systems is the Dolev-Yao model [9], which provides a theoretical framework for analyzing the security of cryptographic protocols. It assumes an active attacker who has complete control over the network, meaning they can eavesdrop, intercept, modify, and inject packets. The attacker in this model is assumed to have unlimited computational power but is constrained in their abilities regarding the cryptographic primitives in use, such as encryption and authentication methods [9]. The Dolev-Yao model is widely used to evaluate the security of protocols in terms of their resilience to various attacks, such as message interception, replay, and forgery.

Enforcement. Once a suitable attacker model and security policy for a system are defined, the next step is to enforce security in practice. This can be achieved through an enforcement mechanism, which consists of a set of safeguards, methods, and techniques designed to verify the system's compliance with the policy and either prevent or flag unauthorized operations. Enforcement mechanisms can take various forms depending on the specific security goals and the nature of the system. Examples of such mechanisms include access control, data encryption, firewalls, and sandboxing.

Access control. Access control is one of the most well-known and widely used enforcement mechanisms in computer systems. It focuses on regulating who (or what) can access specific resources or information. The fundamental idea behind access control is to limit access to resources only to authorized entities, thereby protecting systems and data from unauthorized access or modification. Typically, access control mechanisms define sets of rules based on subjects (users or processes), objects (resources or information), and the operations (actions) that subjects can perform on objects [14].

Access control can be broadly classified into two types: mandatory access control (MAC) and discretionary access control (DAC). MAC is a rigid approach where access rules are created by a central authority based on policies and security labels of the system. MAC is often used in highly secure environments like government or military systems. On the other hand, DAC allows resource owners to define their own access policies, granting them the ability to decide who can access or modify their resources.

Access control mechanisms assume that the attacker cannot directly tamper with the enforcement mechanism itself, but can still employ attacks such as privilege escalation to gain higher-level access. This can be accomplished through various methods, ranging from exploiting bugs in programs to using social engineering tactics. A related concern is the Confused Deputy Problem [11], where an authorized program with high-level access mistakenly uses its privileges on behalf of an attacker. This happens when the program is tricked into performing actions for the attacker, allowing unauthorized access despite the program's original permissions.

One of the initial attempts to formalize access control was the Bell–LaPadula (BLP) [1] model, developed in 1973 with the aim of providing a formal basis for confidentiality using access control policies. As an example of a mandatory access control (MAC) system, the main purpose of BLP was to formalize the U.S. Department of Defense multilevel security policy. The security policy is described as a set of access control rules that use security levels on objects and clearances for subjects. Security levels range from the most sensitive (e.g. *Top Secret*) down to the least sensitive (e.g. *Unclassified*). This gives rise to a multilevel security requirement, which essentially ensures that a subject at a higher level does not convey information to a subject at a lower level. These requirements are formalized in terms of the *No Read Up* property, stating that a subject can only read from objects at a lower or equal security level, and the *No Write Down* property, stating that a subject can only write to an object at a higher or equal security level. The Bell–LaPadula model has influenced the design and implementation of the early security-aware operating systems, such as Multics [3].

The Biba Model [6] was developed in 1975 as the dual of the Bell–LaPadula model with an emphasis on integrity policies. It ensured integrity through two key principles: the simple integrity property (No Write Up) and the star integrity property (No Read Down) [6]. These principles ensure that users cannot write data to a higher integrity level (thus preventing modification of *trusted* data) and cannot read data from a lower integrity level (thus preventing *untrusted* data from affecting *trusted* ones).

An example of access control systems in action, which is widely used today, is the permission system of modern mobile operating systems. Android, for example, employs a permission-based model to regulate how apps interact with the underlying system and user data. An app must declare the permissions it needs and the user must grant them, in many cases explicitly, in order for the app to function as intended. This approach is designed to provide users transparent control over the data they share with the app.

For example, consider a fitness tracking app that is designed to help users monitor their workouts, track running routes, and set health goals. To function, it requests permissions such as access to *Access Fine Location* for GPS tracking, *Activity Recognition* to detect motions like walking or running, and *Internet Access* to sync data with the cloud. These permissions are critical for providing core features, such as displaying detailed workout statistics and allowing users to review their progress over time. The user, wanting to use any of these functionalities, must either grant them explicitly, or in the case of Internet Access, can at least be aware of it.

Even though such an app might appear benign and does not request any unrelated permissions, such as access to contacts or SMS, it can still misuse the data it legitimately procured. For example, the app requests *Access Fine Location*, which is critical for route tracking. However, without additional safeguards, it can misuse this permission to track users' movements for unrelated purposes, such as selling their location data to advertisers. Having *Internet Access* permission to sync fitness data also allows the app to upload user's personal data, such as sensitive location history, to its servers.

Issues such as these are well-known limitations of access control mechanisms. While these mechanisms regulate access to data, they cannot ensure that the data is used appropriately after access is granted. By design, access control mechanisms restrict who can access data, but they cannot control what an authorized entity does with the data after access has been granted. For example, in the case of the fitness app, if the app is authorized to access certain data, it may, either through error or malice, propagate that data to an unauthorized entity, thereby violating the security policy. This issue is particularly relevant in the mobile ecosystem, where apps are typically provided by external developers, often unknown to the user, and downloaded from an app store. As a result, the code is inherently untrusted and could potentially have been provided by an attacker.

Covert channels. The root of this limitation of access control mechanisms is that they can only offer protection when the sensitive data is accessed *explicitly.* This means the subject must attempt to *directly* access the protected object. There are many indirect information channels in a computer system, often referred to in the literature as covert channels [2]. These channels use mechanisms not designed for information transfer to convey data, and are classified into several categories [28]:

- Implicit flows: These occur when the control flow of a program depends on some *secret* information, and different *secret* values produce different attacker-observable behaviors.

- Termination channels: Termination leaks occur when a program's termination (or non-termination) depends on *secret* information. In such cases, an attacker who can observe whether the program halts or continues running, can infer some information about the *secret* information based on the observed behavior.

- Timing channels: Timing leaks occur when the time it takes for a program to execute depends on *secret* data. In such cases, an attacker that can observe the time taken to execute specific parts of the program can deduce some information about the value of sensitive data based on these observations.

- Power channels: A power covert channel exploits variations in power consumption to convey information, that is, a malicious process modulates its power usage in a way that is detectable by another process, thereby using it as a side channel to leak information.

- Probabilistic channels: A probabilistic covert channel uses the stochastic behavior of the system to leak information indirectly, such that the leakage occurs with some probability rather than deterministically. Here, *secret* information influences the observable behavior of the program, modifying the probability distribution of the observable data.

- Resource exhaustion channels: This channel leaks information by exhausting a finite, shared resource (e.g. memory, CPU, disk space). The exhaustion of this *shared* resource, or its absence, can be observed by another party, allowing the information to be conveyed without using traditional explicit communication channels.

Challenges such as covert channels highlight the importance of approaches that can track how the information propagates through a system, ensuring that it adheres to the security policy in an end-to-end manner. In the security literature, this approach is referred to as Information Flow Control (IFC) [28].

IFC mechanisms operate by defining *sources* and *sinks* of information and tracking how data flows within the program from these sources to the sinks. This is unlike traditional access control mechanisms, which manage access to data only at specific points. For example, in the case of a fitness app, the user's location provided by the location service (e.g. a GPS sensor) is a source of information, and the map where this information is displayed or the network socket where it is sent are sinks of information. Android's permission system controls only the source. It checks whether the app has the *Access Fine Location* permission, and if granted, allows the app to read the location information. An IFC mechanism, however, goes further by tracking how the information read from a source, such as the location service, is used throughout the program. It ensures, based on a security policy, that sensitive information read from a location service source does not flow to unauthorized sinks, such as a network socket. By tracking information flows at a granular level, IFC can prevent unauthorized propagation of data, even in the presence of covert channels.

However, implementing information flow control in practice faces significant challenges, including defining precise models of attackers and systems, enforcing flow policies in complex software systems, and designing policies that accurately reflect real-world requirements. In the next chapter, we will look at information flow control in detail and investigate these challenges while highlighting the contributions of the thesis.

3 | Information Flow Control

> “*And because, in all the Galaxy, they had found nothing more precious than Mind, they encouraged its dawning everywhere. They became farmers in the fields of stars; they sowed, and sometimes they reaped. And sometimes, dispassionately, they had to weed.*”
>
> – **Arthur C. Clarke**, 2010

Security enforcement mechanisms like access control fall short in preventing unauthorized information flows, especially through covert channels. Information flow control addresses this limitation by tracking how data is propagated within a system and ensuring it adheres to security policies. IFC defines sources and sinks of information and enforces rules on how information can flow from sources to sinks, preventing leaks by tracking what happens to the data even after it has been accessed. However, information flow control requires a complete analysis of the system as a whole, which, given the increasing complexity of modern software, poses both theoretical and practical challenges [37]. One popular approach in this domain is called Language-Based Security (LBS) [28], which relies on programming language analysis techniques to enforce information flow control.

This chapter introduces language-based information flow control. It is divided into three sections, each focusing on a key aspect: how to specify information flow control properties, how to enforce them using language-based mechanisms, and how to apply these techniques to real-world programming languages. Each section includes a brief overview of the concepts, followed by our contributions.

3.1 Specification

Figure 3.1 illustrates the main ingredients necessary for analyzing a program using information flow control. We rely on these ingredients to precisely define what it means for a program to be secure in a specific setting.

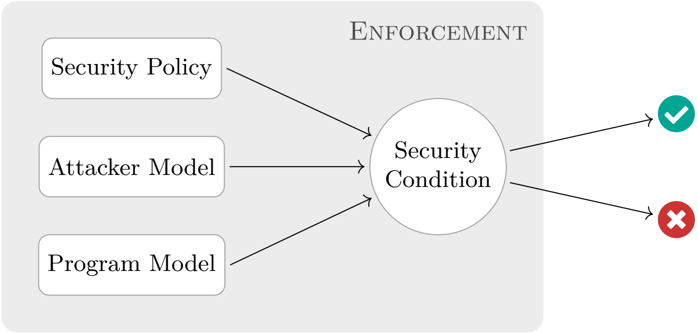


Figure 3.1: Information flow analysis ingredients

The program model provides a precise representation of the behavior of the program. For example, the program model can be represented as a state transformer, which produces a set of executions, capturing various behaviors of the program.

To define an information flow security policy, we need to identify sources and sinks of information. The security policy categorizes these sources and sinks into various security classifications (i.e. labels) and defines acceptable information flows between sources and sinks, specifying what constitutes a *secure flow* within the system. For example, for confidentiality, information from *secret* sources should not flow to *public* sinks, and dually, for integrity, information from *untrusted* sources should not affect (or flow to) *trusted* sinks.

The attacker model is used to represent the security level and the capabilities of the attacker. It identifies which parts of the system the attacker can observe (e.g. program outputs, specific memory regions, or network communications). It also defines the information the attacker has about the system's internal workings (e.g. whether they know the source code or can observe execution termination). The model considers the attacker's abilities with regard to the execution environment (e.g. whether they can track execution steps or measure execution time) and their computational power (e.g. whether they can break cryptographic primitives or are computationally bounded). Additionally, it defines whether the attacker is passive or active (e.g. do they just observe the execution or actively interfere with the execution). Ideally, one would like to show that a system is secure even against the most powerful attackers; however, this is often neither necessary nor feasible.

These ingredients allow us to define the *semantic security condition*, which verifies whether the program model complies with the security policy for a given attacker. The enforcement mechanism, used to practically validate the security of the program, should in practice imply this security condition. In other words, the semantic security condition provides a baseline against which the correctness of the enforcement mechanism is validated.

Program Model

A program model provides a well-defined specification of the program's behavior. The overall behavior of the program, as well as the internal details of the system, such as the stored values in memory, is part of this specification. A common approach is to model a computer program as a state transformer, representing the execution of the program as a sequence of transitions between states. A *state* is a snapshot of the memory of the system at a given point in time. Initial memory, as well as inputs to the system, are modeled via initial states [57]. The program takes an initial state as input and produces a final state based on the program instructions.

Take Program 3.1 as an example. It is a simple program that has two boolean variables: a *secret* variable `h` and a *public* variable `l`. A state, in this case, assigns values to these two variables. For brevity, we represent false and true values with 0 and 1, respectively. Figure 3.2 depicts the program model as a set of runs. A run is a sequence of states that describes the behavior of the program as it executes from an initial state (depicted as a dashed node) to a final one. Runs r_1 to r_4 capture all the possible executions of Program 3.1.

```
bool h;    // secret variable
bool l;    // public variable

l = 0;
if (h)
    skip;
else
    l = 1;
l = 1;
```

Program 3.1

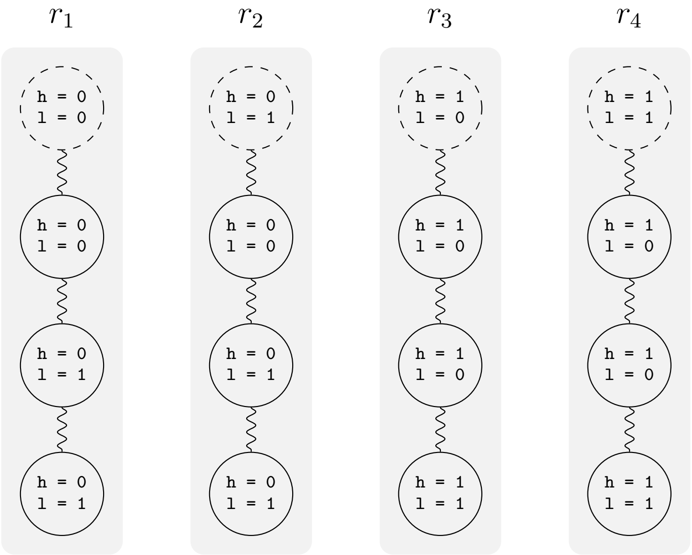


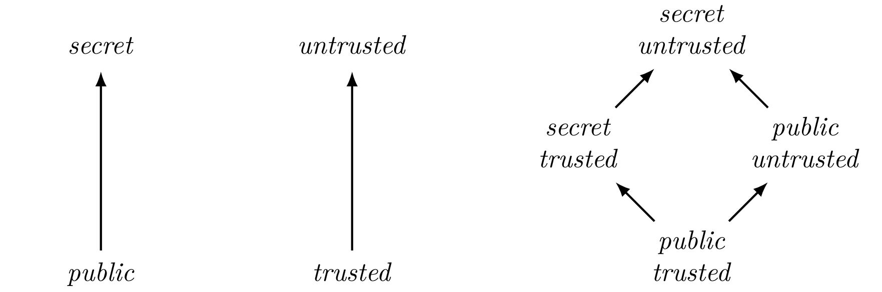
Figure 3.2: Program 3.1 model

Security Policy

In an information flow control context, we need to specify sources and sinks of information. Commonly, the initial value of program variables or the inputs to the program are considered sources of information, and the final values of variables or the outputs produced by the program are sinks of information.

The security policy assigns security labels to these sources and sinks and defines the allowed information flow between them. A common approach for defining a secure flow is to define an ordering between security labels, and state that information for a specific label is only allowed to flow to the same label or a higher ones. These labels and their ordering can form a lattice [4].

We depict a simple lattice for confidentiality in Figure 3.3a, where there are two security labels: *secret* for sensitive information and *public* for public information. The ordering puts *secret* information above *public* information, which means that data from *public* sources may flow to *secret* sinks but not vice versa. This closely aligns with the Bell-LaPadula (BLP) model's *No Read Up* and *No Write Down* rules, which prevent the unauthorized disclosure of sensitive information between different security levels.



(a) Confidentiality (b) Integrity (c) Confidentiality and integrity

Figure 3.3: Security lattices

Dually, the integrity lattice depicted in Figure 3.3b classifies data into *untrusted* and *trusted* categories, with the ordering stating that only *trusted* data may flow to *untrusted* sinks. This is in line with the intuition that *untrusted* data should not affect (and corrupt) *trusted* data. For example, consider a web application where users can provide inputs through form submissions. These user-provided inputs are labeled as *untrusted*, whereas the records of the backend database of the web application are *trusted*. The integrity policy ensures that *untrusted* inputs should not be able to directly update or modify the *trusted* database records.

For a system in which both confidentiality and integrity constraints are important, labels and orderings such as the one depicted in Figure 3.3c may be employed. For example, here the ordering indicates that (*public*, *trusted*) data can flow to (*secret*, *trusted*) but (*secret*, *trusted*) cannot flow to (*public*, *untrusted*). Because there is no arrow between (*secret*, *trusted*) and (*public*, *untrusted*), this means that these labels are incomparable.

Static vs dynamic policies. The security policies are broadly divided into two categories: static and dynamic. A security policy is called *static* when the initial labeling of the data does not change throughout the life-cycle of the system. In other words, what was initially declared *secret* will always remain *secret*, and what was *public* will always remain *public* (i.e. its sensitivity level is not elevated to *secret* while the system is running).

This, however, is not always the case with most real-world systems, where the security requirements of the system change during the life cycle of the system. This means that the sensitivity level of the information can change to a lower level (downgrade) or a higher level (upgrade). This makes dynamic policies a natural choice for many real-world applications, e.g. healthcare systems, social networks, and database systems, where access to information may be granted or revoked for different principals in accordance with their specific role at a given moment. For example, assume a user Alice who works within an organization and can observe *secret* data. When Alice leaves the organization, it is necessary to change the policy and set her security level to *public*.

Conjunctive vs disjunctive policies. Another class of information flow policies is disjunctive policies. These policies capture scenarios where the security policy must restrict access to information based on conflict-of-interest classes, allowing the flow of information only if the observer has not seen conflicting data from any of the other classes. An example of such policies is the so-called *ethical wall policy* [12], which is widely known in the business world and used to manage information where there are conflicts of interest. A disjunctive policy is used to prevent conflicts of interest by restricting access to conflicting information. For instance, consider Alice, a lawyer at a firm that represents two rival companies, A and B. Ethical constraints dictate that if Alice is working with company A, she must not be able to access information related to company B, as her involvement with A creates a conflict of interest with B. In this scenario, a disjunctive policy ensures that Alice can only access information about B if she has not previously accessed any data about company A.

Disjunctive policies contrast with conjunctive policies, where access to information is determined by the *conjunction* of some access criteria. In conjunctive policies, an entity can access certain information only if all required conditions are satisfied. For example, a conjunctive policy might state that Alice is permitted to access a company's data only if she works for the law firm *and* has completed a particular training course.

Attacker Model

The security level of the attacker, their capabilities, and what they can observe about the program's behavior play a crucial role in our definition of security. Our assumptions of the attacker depend on the setting in which the program is supposed to be used. In a system where multiple processes have to share the same memory, such as multi-threaded or mainframe systems, we can assume an attacker that can observe the *public* part of the memory, which is shared between all processes. In this setting, the attacker is just another process that observes how the other processes modify this *public* memory section.

While in scenarios such as web applications, where the program is executed on the server and only the results are returned to the client, we can assume the attacker is a client and can only observe the values produced by the server program.

To illustrate this, let us revisit Program 3.1. Figure 3.4 depicts the runs of this program as observed by an attacker that can only see the *public* part of the memory. Hereafter, we call what an attacker can observe of a run as an *observation trace.*

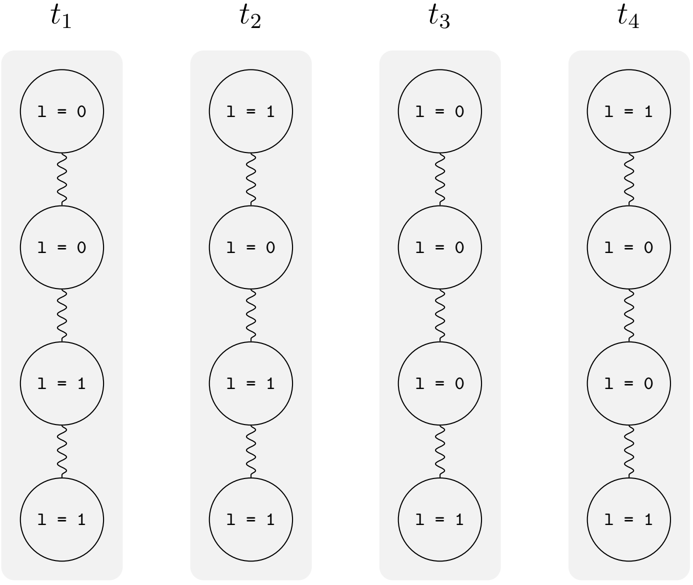


Figure 3.4: *public* observation traces of Program 3.1

In language-based context, the program's source code and the resulting program model are considered *public* knowledge. This implies that an attacker has full knowledge of the program's source code and can predict its behavior under various initial conditions. An attacker thus attempts to learn sensitive information by trying to match which specific run of the program aligns with their observation trace.

A common trait in recent works on information flow control has been to model attackers by relying on epistemic models of knowledge, representing the knowledge of the attacker as an equivalence relation on the initial states of the program, which partitions the domain of all possible states into equivalence classes by relating two states whenever the attacker cannot distinguish between them.

For example, an attacker knowing the source code of Program 3.1 knows that its program model can be represented by the runs depicted in Figure 3.2. Suppose the attacker sees the observation trace t_1 from Figure 3.4. Initially, the attacker considers all the runs in Figure 3.2 possible, meaning they cannot distinguish any of the initial states, partitioning the set of all initial states as depicted in Figure 3.5. At the first step of the observation trace t_1, seeing `l = 0`, the attacker can narrow down the possible runs to r_1 and r_3. This means that they can partition the initial states as depicted in Figure 3.6, reflecting the fact that the attacker can distinguish the initial states in which `l` has a different value. Later, at step 3 of t_1, when the attacker observes the transition from `l = 0` to `l = 1`, they can further deduce that only r_1 could have produced this observation. By this reasoning, the attacker infers that only the initial state corresponding to r_1 could have generated the observation trace t_1, thus further refining the partition of initial states and producing the equivalence relation of Figure 3.7. As a result, the attacker can conclude that the initial value of the *secret* variable `h` must have been 0.

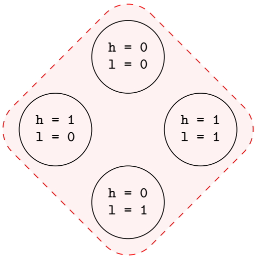


Figure 3.5

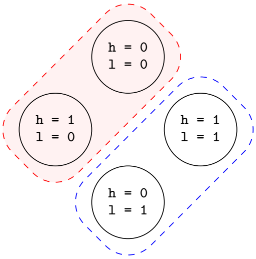


Figure 3.6

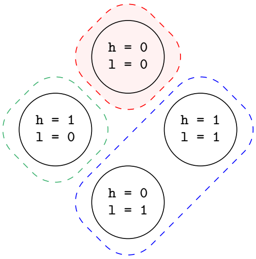


Figure 3.7

Termination. Another capability of the attacker that must be considered is whether they can detect the termination of the program's execution. To illustrate this, assume an attacker that observes the outputs of Program 3.2. An attacker that can observe termination sees the observation traces depicted in Figure 3.8, where each node is the observable output, and ✖ indicates the termination of the execution, whereas an attacker that *cannot* detect termination sees the observation traces depicted in Figure 3.9.

```
bool h;    // secret variable

print(1);
print(2);
print(3);
while (h)
{
    skip;
}
```

Program 3.2

The attacker seeing the observation traces of Figure 3.8 can distinguish between the initial states that will cause the program to output 1, 2, and 3 and loop forever (i.e. `h = 1`) and the ones that output the same values but terminate (i.e. `h = 0`), thereby learning the value of the *secret* variable `h`. In contrast, the second attacker cannot learn this, as the observation traces of Figure 3.9 do not let them distinguish between the terminating and non-terminating behaviors.

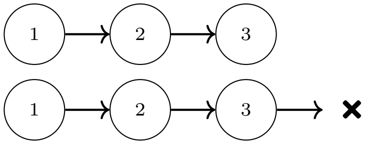


Figure 3.8: Traces with termination

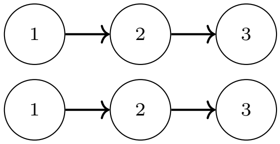


Figure 3.9: Traces without termination

Passive and active attackers. An attacker can either just observe the execution of the program or try to actively interfere with it by code injection or data manipulation. The former is referred to as a passive attacker, while the latter is called an active attacker. Whether an attacker is passive or active plays a crucial role in the security of the system.

Consider Program 3.3, which simply prints the contents of a *public* variable. We usually model the abilities of active attackers by placing holes `[•]` in the program through which the attacker can inject their code. For example, in Program 3.3, an active attacker can simply inject `l = h` into the hole `[•]` on line 5 and observe the value of the *secret* variable `h` being printed.

```
1  int h;    // secret variable
2  int l;    // public variable
3
4  l = 0;
5  [•];
6  print(l);
```

Program 3.3

Active attackers are relevant in web applications, where code from different providers may be included on the same web page, or in trusted execution environments, where the host is not trusted and may attempt to alter the program's execution by manipulating the data being sent to the enclave.

Semantic Security Condition

The semantic security condition establishes a baseline for evaluating the program's security, ensuring that the program model complies with the security policy under that specific model of attacker. The definition and guarantees of a semantic security condition depend on the attacker's capabilities and our definition of security. As a result, different security policies and attacker models lead to distinct security conditions, each addressing specific security concerns.

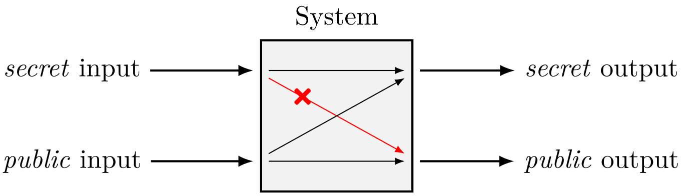


Figure 3.10: Noninterference

Noninterference [8] is the most well-known semantic security condition in information flow control. It ensures that *secret* inputs do not influence the observable *public* outputs of the system. Consider Figure 3.10, which depicts a system with both *secret* and *public* inputs and outputs. *secret* inputs represent sensitive data, such as passwords, while *public* outputs are the observable results visible to external observers. Noninterference follows the intuition that the values of *secret* inputs should not interfere with the *public* outputs, meaning that any change in the system's *secret* inputs should not affect the *public* outputs, ensuring that actions involving *secret* data remain completely opaque to *public* observers.

Noninterference is defined formally by analyzing the observation traces of the system:

Definition 3.1

A system satisfies noninterference if, for any two different inputs differing only in their *secret*-level values, the observation traces visible to a *public*-level observer are indistinguishable.

There are two well-known variants of noninterference based on the attacker's ability with regard to termination. Termination-insensitive noninterference (TINI) refers to definitions in which the attacker cannot distinguish between terminating and non-terminating observation traces. In contrast, termination-sensitive noninterference (TSNI) applies when the attacker can detect termination, meaning that programs which leak *secret* information through termination, such as Program 3.2, are considered interfering.

Limitations of noninterference. Noninterference ensures that the system does not reveal any information about the *secret* inputs. However, this strong security guarantee comes at a price. It is often not possible for many practical computer systems to satisfy this property, as in many cases, a system needs to release certain sensitive information as part of its intended functionality.

Consider Program 3.4, which implements a simple password checker. Such a program must reveal some information that depends on the *secret* value (i.e. the password), as the output needs to differ when a user enters the correct password compared to when they enter an incorrect one.

```
string pass;    // secret variable

bool check(string guess) {
    if (pass == guess)
        return 1;
    else
        return 0;
}
```

Program 3.4

Declassification and endorsement. In the security literature, the intentional release of *secret* information is called *declassification*. Noninterference is unable to capture programs that include declassification. While declassification is intuitively easy to understand, it is challenging [69] to define formal security policies and semantic conditions for programs with declassification that are not only enforceable in practice but also cover a wide range of scenarios where the secure release of information is necessary.

The challenge of formally defining declassification in relation to standard noninterference is largely due to the various aspects and forms of declassification. Generally, declassification requirements are categorized into four dimensions [69]: *What* information is released? *Who* performs the release of information? *Where* in the system the information release takes place? *When* the information release takes place?

The concept of *endorsement* is the dual of declassification in systems focused on integrity. Endorsement raises the integrity level of *untrusted* data to *trusted*, allowing it to safely affect the system's *trusted* data. Program 3.5 depicts a common use case of endorsement in web applications. The user-provided input on line 4 is considered *untrusted* because it might contain malicious content, such as SQL injection, and therefore should not be directly passed to the backend database. To address this, the user-provided input is sanitized to ensure it does not contain any malicious content. After the sanitization operation on line 5, the input is considered safe. At this point, the `endorse` operator on line 6 is used to raise the integrity level of the sanitized input to *trusted*, allowing it to be safely inserted to the backend database.

Robustness. In a system that incorporates active attackers and declassification, another key information flow challenge is to precisely define the effects of the active attacker on confidentiality. This concept, known as robustness [34], focuses on

```
database db;
string input, sanInput, enInput;

input = form.data();    // untrusted user input
sanInput = sanitize(input)
enInput = endorse(sanInput)
db.insert(enInput)
```

Program 3.5

whether an adversary can exploit declassification to bypass confidentiality guarantees. Specifically, it ensures that active attackers, who can pass data or inject code into the running program, cannot learn more information than a passive attacker who merely observes the program execution.

```
1  string h;                     // secret variable
2  int releaseTime = 2042;       // public variable
3
4  bool release(int time) {
5      if (time >= releaseTime)
6          return declassify(h);
7      else
8          return 0;
9  }
```

Program 3.6

To illustrate this, let us consider Program 3.6, which implements a function `release` whose return value depends on the `time` parameter. Under a passive attacker, *secret* variable `h` is only declassified when `time` meets or exceeds the predefined `releaseTime` threshold, checked on Line 2. However, an active attacker capable of manipulating the `time` variable can prematurely trigger declassification, effectively controlling the release of information via declassification.

This problem can arise in many real-world settings, such as trusted execution environments. A trusted execution environment (also called an enclave) is a secure area within a device's main processor that ensures sensitive data and operations are protected, isolated, and executed securely, even in the presence of a compromised operating system. Since in this setting an untrusted host controls the execution environment, the data passed to the system, or even the order of function executions, can be controlled by active attackers.

Contributions

Having introduced the main ingredients necessary for analyzing a program using information flow control, in this section, we put our contributions within this context and discuss closely related works.

Security policies. Denning's seminal work, "A Lattice Model of Secure Information Flow" [4] laid the mathematical foundation for reasoning about information flow security in computer systems. The model employs a lattice structure to represent different security levels, where each level corresponds to a specific degree of confidentiality or sensitivity of the information, allowing the systematic enforcement of security policies that govern how information can flow between entities. Landauer and Redmond [13] introduced the Lattice of Information (LoI) as a model for representing and comparing information. Their model is a lattice structure whose elements are equivalence relations that partition a given set into subsets of indistinguishable elements. In this construction, the partial order is defined on the notion of refinement: an equivalence relation is deemed *higher* in the hierarchy if it contains finer partitions (i.e. smaller equivalence classes) than another equivalence relation. The Lattice of Information is limited to join and meet operations on equivalence relations, making it suitable only for expressing conjunctive policies. To address this limitation, Hunt and Sands [181] extended the Lattice of Information and proposed the Quantale of Information, which includes an additional tensor operator making it suitable for representing the disjunction of equivalence relations. Unlike the Brewer and Nash model for disjunctive policies [12], which was rooted in the area of access control, the Quantale of Information [181] is the first model to provide an extensional characterization of disjunctive policies as an information flow policy.

Bender et al. [99] introduced the Disclosure Lattice as a database counterpart of the Lattice of Information. This model orders sets of queries based on the amount of information they reveal about the underlying database. We observe that the definition of disclosure order is not precise enough to capture information disclosure in the information flow sense. Therefore in Chapter D we use a query determinacy-based ordering and propose the Determinacy Lattice to enforce information flow policies on database queries. Building on the quantale model of Hunt and Sands [181], we extend this definition to the Determinacy Quantale, intended as a semantic model to express disjunctive policies in terms of database views. Building on the Determinacy Quantale and the Quantale of Information [181], we present a security condition for verifying disjunctive policies in database-backed programs.

Robustness. Over the years, active attackers and their effect on security have been investigated by researchers [23, 34, 63, 71, 80]. Zdancewic and Myers [23, 34] introduced the notion of robustness for systems that include the controlled downgrading of information via declassification. Research on robustness, in general, investigates the effect of active attackers in comparison to passive attackers, ensuring that a system under active attack reveals no more information than the system under a passive attacker's observation, meaning that declassification cannot be exploited to obtain more information than was intended.

Given the relevance of active attackers in trusted execution environments, in Chapter A, we explore active attacker models and their effect on the robustness of programs running on trusted execution environments. We examine two attacker models: one that can manipulate data by modifying the parameters passed to the methods of the enclave program, and another that can affect the control flow of the program by changing the order and frequency of calling the enclave program's methods. We propose two robustness-based security conditions to ensure security against these types of attackers.

Dynamic policies. Attacker models have also been investigated with regards to dynamic policies [43, 64, 72, 88, 114, 116]. In this setting, usually the epistemic capabilities of the attacker, such as their logical omniscience or their memory recall, are investigated.

Askarov and Chong [88] proposed a general framework for capturing the semantics of dynamic policies for all attackers. Their security condition defines learning as a change in knowledge, based on the intuition that an attacker only learns something from an event if the attacker's knowledge after the event is more precise than their knowledge before. They argued that in a dynamic policy setting, security for the most powerful attacker does not necessarily imply security against all attackers, demonstrating that a secure program under a perfect recall attacker can be insecure under a weaker attacker. Van Deft et al. [116] improved on Askarov and Chong's [88] framework by proposing a new definition for progress-insensitive security, eliminating the problematic behavior of some types of willfully stupid attackers who may remember some observations while ignoring others.

To understand the relationship between various attacker models and the security condition, and to capture scenarios where the release of information is transient, we investigate dynamic security policies in Chapter C. We focus on three types of attackers: a perfect recall attacker who can remember all observed information from the past, a bounded memory attacker with limited memory who can only retain observations within a certain range, and a forgetful attacker who forgets past observations when the policy changes (used to model scenarios where released information was transient). We introduce the notion of inconsistent policies, capturing cases where the policy change is not in line with the attacker's knowledge and restricts the attacker's access to some information they already possess. We introduce a security framework that incorporates policy consistency and the attacker model to capture the semantics of dynamic policies. In contrast to Askarov and Chong [88], we prove that, in the absence of inconsistent policies, security against a perfect recall attacker does indeed imply security against weaker attackers.

3.2 Enforcement

The next step is to enforce the semantic security condition through a language-based enforcement mechanism. This mechanism consists of methods and techniques for verifying information flow properties, ranging from syntactic to semantic and from static to dynamic approaches. There is an intricate trade-off between the expressiveness of the security condition and the practical ability of the language-based mechanism to enforce it.

In general, enforcement mechanisms are evaluated by two key criteria: *soundness* and *completeness*. Soundness ensures that the mechanism accepts only secure systems, while completeness guarantees that secure systems are not incorrectly flagged as insecure by the mechanism. Another important aspect of the usability of enforcement mechanisms is their precision. Due to the overapproximation inherent in language-based mechanisms, they sometimes reject secure programs. While rejecting a secure program does not compromise the soundness of the mechanism, it does reduce its usability. The ideal is to have a permissive yet sound language-based mechanism that guarantees soundness while rejecting as few secure programs as possible.

Although one can come up with advanced semantic security conditions able to express a wide range of security properties, their practical value is limited if there is no enforcement mechanism that can effectively validate whether a program adheres to them or not. The enforcement mechanisms discussed in this section vary in terms of their permissiveness and their ability to enforce different security conditions.

The enforcement of information flow typically requires mechanisms to track the data flow within a system. In general, enforcement mechanisms can be categorized into two classes: static and dynamic. Static techniques enforce security at compile time by analyzing the program code without executing it. This approach is appealing because it verifies the program prior to execution, thereby eliminating runtime overhead. Dynamic techniques, on the other hand, ensure program security by using runtime information to perform information flow analysis. They prevent insecure behaviors during execution, through techniques such as modifying the program's semantics to prevent information leaks or terminating the program if a leak is detected.

Dynamic Techniques

Dynamic techniques enforce security policies during program execution by monitoring the program's runtime behavior. These methods supervise the actual execution of the program and make adjustments as needed to ensure security. Dynamic approaches are especially useful in highly dynamic environments, like web applications, where code and data are not always known before runtime.

We provide a brief overview of some dynamic techniques but omit excessive details, as the main focus of this thesis is static information flow control and no contributions were made toward dynamic enforcement mechanisms.

Security monitors. A popular approach in dynamic enforcement is called a security monitor, which is another program that monitors the execution of the target program, checking for policy violations and intervening by halting the execution or modifying the program's behavior in the event of a policy violation. The flexibility of monitors in enforcing security in highly dynamic contexts often comes at the cost of runtime overhead. In many cases, such as real-time systems or network devices, this additional overhead can be unacceptable. Additionally, dynamic monitoring is inherently limited in enforcing noninterference, as it is a *hyperproperty* [74]. Unlike traditional properties, such as safety and liveness [7], which can be enforced on a single run, hyperproperties are defined over multiple runs and capture the relationships between them. As a result, hyperproperties cannot be precisely enforced in practice by examining only one run [74].

To overcome this issue, monitors sometimes need to overapproximate. For instance, in a dynamic setting, updating the value of a *public* variable in only one branch of a conditional guarded by a *secret* condition may lead to partial information leakage. This occurs because the variable's value changes in one execution while remaining unchanged in an alternative execution. To address this problem, many dynamic IFC methods use the so-called no-sensitive-upgrade (NSU) check [62], which terminates the program's execution whenever a *public* variable is updated within a *secret* context.

Secure multi-execution. Secure multi-execution (SME) is another runtime enforcement mechanism for information flow [75, 92]. This approach enforces noninterference by running multiple copies of the program, each corresponding to a specific security level. Outputs are allowed only if the security level of the program matches the security level of the output channel, and inputs that are not visible at the program's security level are replaced with default values. Figure 3.11 depicts how secure multi-execution achieves noninterference by running the program twice, once as a *secret* copy and once as a *public* copy.

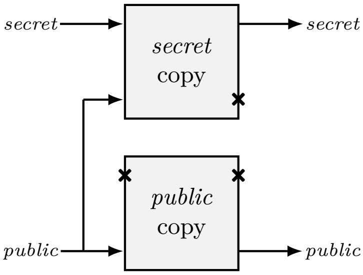


Figure 3.11: Secure multi-execution

Secure multi-execution has been shown to be both sound and precise [86]. Soundness ensures that every execution at a specific security level cannot access the data from higher security levels. As a result, all outputs it produces are generated only from data at or below its level, guaranteeing noninterference. Precision, on the other hand, guarantees that if a program is noninterfering under normal execution, its behavior remains identical to that under secure multi-execution for terminating runs [86].

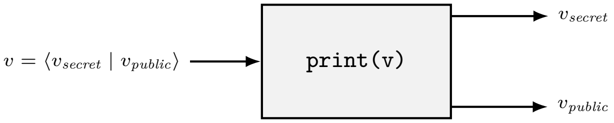


Figure 3.12: Faceted execution

Faceted execution. Faceted execution [89] is another dynamic enforcement mechanism closely related to secure multi-execution [123]. It simulates the multiple executions of SME while avoiding unnecessary redundant executions by creating multiple facets, or views, of data corresponding to different security levels. For example, a value v would be represented as $\langle v_{secret} \mid v_{public} \rangle$, where v_{secret} is the value observable to *secret* observers (i.e. the *secret* facet) and v_{public} is the default value observable to *public* (i.e. the *public* facet). As depicted in Figure 3.12, the program is executed as if both facets are being executed simultaneously, but the outputs to any observer are restricted to the facet they are allowed to see.

Static Techniques

Various static mechanisms exist for analyzing a program's source code to track information flow and enforce security, each offering unique advantages and disadvantages. Next, we will introduce some of these mechanisms.

Security type system

A type system is a formal system consisting of a set of rules that assign a *type* (e.g. integer, float, string) to each term of a program. A term is a language construct of the programming language, such as variables, expressions, or functions. The rules of the type system define valid operations and values for these terms, aiming to prevent type-related errors and reduce bugs in the program. Type-related errors occur when operations are applied to terms of incompatible types, such as attempting to perform arithmetic on non-numeric data, potentially leading to unpredictable behavior or program crashes.

A security type system builds upon standard type systems by assigning security labels as types. The type checker then follows a set of rules designed to enforce a specific security condition, often a variant of noninterference. Ultimately, the goal of the security type system is to guarantee that a well-typed program — one that satisfies the typing rules — is secure with respect to the defined security condition.

```
bool h;    // secret variable
bool l;    // public variable

if (h)
    l = 1;
else
    l = 0;
```

Program 3.7

Consider Program 3.7. This program is insecure because, although it does not explicitly assign the value of the *secret* variable `h` to the *public* variable `l`, a *public* observer can still deduce the value of `h` by observing `l`. This deduction is possible because of an *implicit flow*, which arises due to the control flow dependency of `l` on `h` through the `if` statement.

We use this program to illustrate how a type system can prevent implicit flows and enforce noninterference. Consider the typing rules depicted in Figure 3.13.

T-Assign

$$\frac{\vdash x : \ell_x \qquad \vdash e : \ell_e \qquad pc \sqsubseteq \ell_x \qquad \ell_e \sqsubseteq \ell_x}{pc \vdash x := e}$$

T-If

$$\frac{\vdash e : \ell_e \qquad pc' = pc \sqcup \ell_e \qquad pc' \vdash c_1 \qquad pc' \vdash c_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2}$$

Figure 3.13: Security type system

Assume that the type ℓ (hereafter referred to as a label) is chosen from the lattice shown in Figure 3.3a, i.e. $\ell \in \{public, secret\}$. These rules aim to protect confidentiality by enforcing noninterference, which means preventing information flows from *secret* variables to *public* variables.

This type system uses judgments of the form $pc \vdash c$, signifying that a program c is typable within the security context pc. The *program context* pc is a special type (i.e. $pc \in \{public, secret\}$) that denotes the security label of the current control flow context. The pc prevents implicit flows by enforcing restrictions on the execution of commands that could implicitly convey *secret* data through program behavior.

Consider the rule T-Assign. This rule ensures that an assignment to a variable x is permitted only if all variables in the expression e are allowed to flow to x. Here, pc prevents implicit flows by ensuring that an assignment is typable only if the label of the assigned variable, ℓ_x, is at least as restrictive as pc, which is checked by the ordering relation of the lattice $pc \sqsubseteq \ell_x$. The check $\ell_e \sqsubseteq \ell_x$ ensures that the labels of all variables in the expression e are below the label of x, and thus, are permitted to flow to it.

The rule T-If says that a conditional statement is typable in context pc only if each of its sub-commands, c_1 and c_2, are typable in context $pc \sqcup \ell_e$. Here, ℓ_e is the least upper bound of the labels of all the variables in e, and $\sqcup$ is the lattice's join operator, which returns the least upper bound of the labels *pc* and ℓ_e. In the context of the lattice shown in Figure 3.3a, this means that if either *pc* or ℓ_e is *secret*, their least upper bound will also be *secret*.

To illustrate how a type system works, we type-check Program 3.7 using these rules. At the start of the program, the *pc* is *public*, reflecting that no *secret* control flow dependencies exist. However, when type checking the `if (h)` statement, the *pc* is raised to *secret*, as the control flow now depends on the *secret* variable `h`. We type-check the assignments on lines 5 and 7 under this *secret* context, pc'. Starting with the assignment `l = 1`, we observe that the label of the assigned variable is *public*, since variable `l` is *public*, and the label of expression 1 is also *public*, as it is a constant value without any variables. However, the type checking of this command fails. While the check $\ell_e \sqsubseteq \ell_x$ succeeds, indicating that an explicit flow is permissible, the check $pc' \sqsubseteq \ell_x$ fails. This is because $\mathit{secret} \not\sqsubseteq \mathit{public}$, indicating an implicit flow. Consequently, this assignment is rejected under the current program context pc', effectively preventing an implicit information flow.

The ability to soundly track implicit flows is one of the key advantages of static enforcement mechanisms. A type system can ensure that no possible path within the if statement contains insecure flows.

Another major advantage of security type systems is compositionality. Compositionality refers to the ability of the type system to maintain that the security guarantees of individual program components hold when these components are combined. Consider the typing rule depicted in Figure 3.14. This rule states that the sequential composition of two commands, c_1 and c_2, is well-typed if each of those commands is individually well-typed.

$$\textsc{T-Seq}\quad \frac{pc \vdash c_1 \qquad pc \vdash c_2}{pc \vdash c_1; c_2}$$

Figure 3.14: Sequential composition rule

Compositionality is essential for building secure, modular, and scalable systems, where security guarantees of the whole system can be inferred from the security guarantees of its individual components.

However, this compositionality comes at a price. Since individual components are type-checked independently before being composed together, the order in which components are type-checked does not affect the final result. As a result, these type systems are unable to enforce policies where the security policy and the labeling of data depend on various factors, such as location in code, time, or user-specific conditions.

Type systems' overapproximation is to ensure soundness, as they must conservatively account for all potential flows of information within the program. Since precise runtime behavior may not be fully determined statically, type systems err on the side of caution, rejecting programs that may be insecure, even if they are secure at runtime.

To illustrate this, consider Program 3.8. This program is rejected by the typing rules of Figure 3.13 due to overapproximation, despite being secure. Similar to Program 3.7, the type system raises the program context *pc* to *secret* when entering the `if (h)` statement. Inside the branches, the *pc* remains *secret*, and any assignments within these branches are treated as potentially leaking *secret* information. However, both branches assign the same value (`1`) to the *public* variable `l`, meaning no information about `h` is being leaked. The type system overapproximates by assuming the possibility of different behaviors in the branches based on `h`. This overapproximation leads to the program being rejected because the assignment to `l`, which is *public*, occurs in a context with a *secret pc*, violating the T-Assign rule, even though the actual runtime behavior is secure.

```
bool h;   // secret variable
bool l;   // public variable

if (h)
    l = 1;
else
    l = 1;
```

Program 3.8

More expressive type systems, such as the ones based on dependent types [117] or refinement types [176], can capture additional information about the types and make the security type systems more precise. These systems usually involve complex typing rules and require more sophisticated reasoning, making them challenging to implement and difficult to use effectively. Nevertheless, one could argue that these more expressive type systems are necessary for verifying the security of real-world programs.

The expressiveness of the type system goes hand in hand with the expressiveness of the security policies they are designed to enforce. Simple security type systems, such as the one in Figure 3.13, are limited to security policies that can be expressed by annotating program terms with fixed labels. These policies have limited ability to express complex scenarios, such as those involving distributed systems, side channels, and data-dependent policies.

A data-dependent policy defines the security label of the data based on its value. Such policies are particularly useful in scenarios where data is considered *secret* only under certain conditions. For instance, consider a network firewall that aims to ensure that internal network packets do not leak to external, attacker-visible networks. In this case, a packet would be labeled *secret* only if its source and destination IP addresses belong to the internal network address range (`192.168.*.*`). These data-dependent policies cannot be represented using simple security labels, such as those employed in Figure 3.13.

The attacker model is another important aspect to consider when designing a type system. Generally, type systems such as the one presented in Figure 3.13 consider only passive attackers, who observe the *public* variables or the outputs of the program. When the threats to the program's security involve active attackers, the type system needs to be modified to accommodate the effects of the attacker's actions and provide security guarantees such as robustness [34]. Such issues are particularly important in trusted execution environments, where the host system might be compromised (and *untrusted*), enabling an attacker to affect the data and control flow of the program.

Type-based dependency analysis

Security type systems, such as the one introduced in Section 3.2, are flow-insensitive, meaning that they assign a fixed security label to each term. Such fixed labels have limited flexibility. For example, in the case of variables, they do not account for changes in the sensitivity of the data stored within the variable. To illustrate the limitations of flow-insensitivity, consider Program 3.9 from [85].

```
 1  int h1, h2;   // secret variables
 2  int l1, l2;   // public variables
 3  int tmp;
 4
 5  tmp = h1;
 6  h1 = h2;
 7  h2 = tmp;
 8
 9  tmp = l1;
10  l1 = l2;
11  l2 = tmp;
```

Program 3.9

This program is rejected by flow-insensitive security type systems due to the inconsistent label of the variable `tmp`. If `tmp` is labeled as *secret*, the assignment on line 5 will be insecure, and if `tmp` is labeled *public*, the assignment on line 9 will be rejected. The issue lies in the inability of flow-insensitive systems to adapt the label of `tmp` over time to accurately reflect the security label of its contents.

Flow-sensitive security type systems have been investigated over the years to address this limitation [78, 136]. These systems, however, have their own limitations, such as difficulty in handling aliasing, imprecision when analyzing loops, and limitations in analyzing individual components in isolation.

Another approach to address the limitation of flow-insensitive type systems is *type-based dependency analysis* [85]. A type-based dependency analysis explicitly tracks dependencies between program variables. Unlike security labels, where each variable is assigned a fixed security label (e.g. *secret* for confidentiality or *trusted* for integrity), a type-based dependency analysis captures how the variables interact with each other and how their dependencies propagate through the program. For example, consider the code `z = x + y`. Here, the value of `z` is computed based on the values of `x` and `y`, and the type-based dependency analysis records that `z` depends on both `x` and `y`.

To illustrate how a type-based dependency analysis works, consider the rules depicted in Figure 3.15.

$$\textsc{T-Assign}\quad \frac{\Gamma = \Gamma_{id}[x \mapsto fv(e) \cup \{pc\}]}{\vdash x := e : \Gamma} \qquad \textsc{T-If}\quad \frac{\vdash c_i : \Gamma_i \qquad \Gamma'_i = \Gamma_i ; \Gamma_{id}[pc \mapsto fv(e) \cup \{pc\}] \quad i = 1,2}{\vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : (\Gamma'_1 \cup \Gamma'_2)[pc \mapsto \{pc\}]}$$

Figure 3.15: Type-based dependency analysis

The typing rules use judgments of the form $\vdash c : \Gamma$, which state that the typing of program c produces the dependencies described in the environment Γ. Here, $\Gamma : \mathit{Var} \rightarrow 2^{\mathit{Var}}$ is an environment that maps each variable to a set of variables, describing its dependencies. The identity environment Γ_{id} maps each variable to a singleton set that includes only the variable itself. For instance, $\Gamma_{id}(x) = \{x\}$. The function $fv(e)$ returns the set of free variables in the expression e. For example, $fv(\texttt{y * z + 1}) = \{y, z\}$. The composition $\Gamma_2; \Gamma_1$ represents the sequential composition of two environments. Intuitively, the result of sequential composition is Γ_2 with all dependencies already established in Γ_1 substituted accordingly.

In the T-Assign rule, the dependencies of the assigned variable are determined by the free variables of the assignee expression $fv(e)$ and the program context pc. Here, pc is just a special variable that is also mapped to a dependency set. The T-If rule works by first extracting the dependencies of each sub-command. For each sub-command, it updates the dependencies of its program context pc to include the free variables of the condition guard expression e. The final result is the union of these environments, indicating the merging of the dependencies of both branches.

Type-based dependency analysis, like other type-based analyses, also overapproximates. Consider Program 3.10, a simple program with a single `if` statement. In this program, the final value of x depends only on the initial value of x and either y *or* z. However, according to the T-If rule, the final dependency set of x would be x, y, z. This is because the analysis, unaware of the actual value of x, must

```
if (x != 0)
    x = y;
else
    x = z;
```

Program 3.10

conservatively track dependencies from both branches to guarantee soundness.

One of the advantages of type-based dependency analysis is its independence from the security policy. In security type systems such as the one introduced in Section 3.2, the security policy must be known at the time of type checking. This means that the sensitivity level of the terms must be known in order to correctly assign a security label to them. A type-based dependency analysis eliminates this requirement. A program can be analyzed once to extract the dependencies of its terms, and these dependencies can then be checked against various security policies without the need to reanalyze the program.

Symbolic execution

Another static enforcement method is symbolic execution [5], which analyzes a program by tracking symbolic values rather than actual values during execution. Instead of executing the program with concrete input values, symbolic execution assigns symbolic values to statically unknown inputs. As the program executes, symbolic expressions over concrete and symbolic values are built up to represent the state of variables at each point, capturing their relationship to inputs and control flow conditions. Wherever the program's execution branches, such as upon reaching conditionals or in loops, symbolic execution explores all possible paths. It generates a predicate that encodes the conditions under which that specific path was reached. These predicates are referred to as *path conditions*. By constructing and analyzing these path conditions, symbolic execution can systematically explore all possible paths, while keeping track of the conditions under which a specific path was visited. These symbolic expressions and path conditions form *symbolic states*, which describe the program's state at various points during the execution.

A symbolic execution tree is a structure that represents all possible runs of a program. In this tree, each node corresponds to a symbolic state. For instance, an if statement splits the tree into two branches, one where the condition holds true and one where it is false. To illustrate how symbolic execution operates, consider Program 3.11.

The diagram of Figure 3.16 represents the symbolic execution tree of Program 3.11. Symbolic execution visits all paths of the program by tracking symbolic values for x and y (i.e. inputs) rather than their actual values. It initially sets $x \mapsto \alpha$ and $y \mapsto \beta$, where α and β represent symbolic values for any integer value. The variable z is assigned the concrete value 0.

The path conditions and the symbolic expressions assigned to x, y, and z form the symbolic states of Program 3.11. For presentation purposes, in Figure 3.16, we show the path conditions over the edges connecting the symbolic states.

The if statement at line 4 splits the tree with two path conditions: $\beta \geq 0$ (`then` branch) and $\beta < 0$ (`else` branch). The `then` branch goes through the assignments

```
 1  int function(int x, int y)
 2  {
 3      int z = 0;
 4      if (y >= 0)
 5      {
 6          y = y + x;
 7          x = y - x;
 8          y = y + x;
 9          return(x);
10      }
11      else
12      {
13          if (y + z < 0)
14              z = y * (-1);
15          return(z);
16      }
17
18      return(x);
19  }
```

Program 3.11

of lines 6-8 and swaps the values of x and y, eventually returning x. The `else` branch checks the condition $y + z < 0$, again splitting the tree. If $y + z < 0$ holds, the variable z is assigned the negation of y, and the program returns it. Finally, if neither of the if conditions hold, the program returns x. Note that this path is unreachable because the path condition $\beta \geq 0 \wedge \beta + z \geq 0$ is unsatisfiable and is therefore pruned during symbolic execution. However, for presentation purposes, we still depict this path in Figure 3.16.

Each path in the tree from the start node to a leaf represents a set of runs. The conjunction of path conditions along each path represents the conditions that, if satisfied by an initial state, will result in the execution of the program along that path. For example, the concrete initial state $(x = 42, y = 7)$ will execute along the path starting with the condition $\beta \geq 0$, eventually leading to the final state $(x = 7, y = 42)$, and returning 7.

Recall that information flow security conditions specify a relationship between inputs (sources) and outputs (sinks). In symbolic execution, this relationship is captured by the output's symbolic expression, which represents the effect of symbolic values (inputs) on the output. This characteristic makes symbolic execution an effective static analysis technique for enforcing various information flow security conditions.

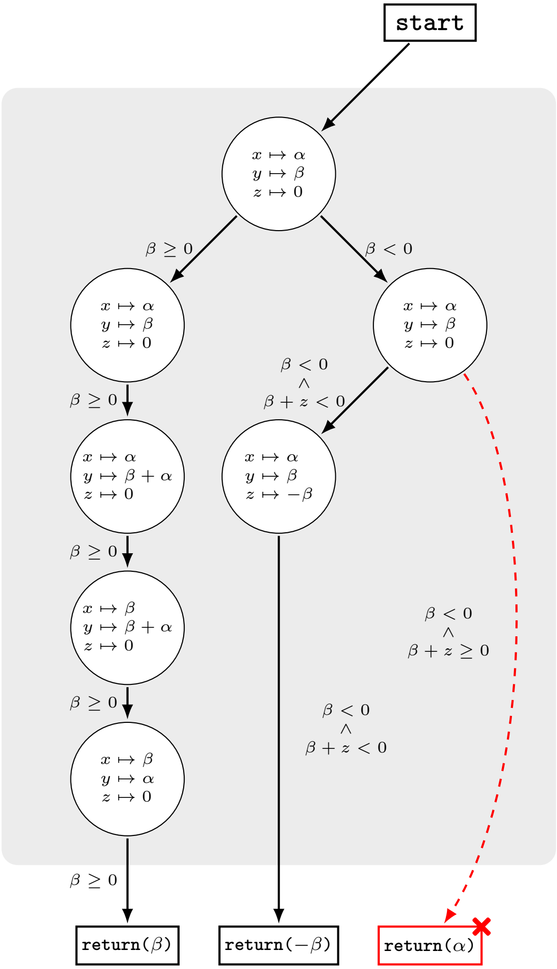

start
$x \mapsto \alpha$ $y \mapsto \beta$ $z \mapsto 0$
$\beta \geq 0$
$\beta < 0$
$x \mapsto \alpha$ $y \mapsto \beta$ $z \mapsto 0$
$x \mapsto \alpha$ $y \mapsto \beta$ $z \mapsto 0$
$\beta \geq 0$
$\beta < 0 \wedge \beta + z < 0$
$x \mapsto \alpha$ $y \mapsto \beta + \alpha$ $z \mapsto 0$
$x \mapsto \alpha$ $y \mapsto \beta$ $z \mapsto -\beta$
$\beta \geq 0$
$x \mapsto \beta$ $y \mapsto \beta + \alpha$ $z \mapsto 0$
$\beta < 0 \wedge \beta + z \geq 0$
$\beta \geq 0$
$\beta < 0 \wedge \beta + z < 0$
$x \mapsto \beta$ $y \mapsto \alpha$ $z \mapsto 0$
$\beta \geq 0$
return(β)
return($-\beta$)
return(α)

Figure 3.16: Symbolic execution tree of Program 3.11

For example, we want to enforce noninterference on Program 3.11 under a security policy that specifies input `x` is *secret* and input `y` is *public*. By the definition of noninterference, the *secret* input `x` should not influence the return value of this function. Referring to the diagram in Figure 3.16, we can see that the symbolic output values, represented in the leaf nodes, are β, $-\beta$, and α. Since the input to symbolic variable mapping was $x \mapsto \alpha$ and $y \mapsto \beta$, the first two outputs are noninterfering. However, even though the last output (i.e. α) is interfering, the path condition leading to it is $\beta < 0 \wedge \beta + z \geq 0$, and since the symbolic state maps $z \mapsto 0$, this path condition is unsatisfiable. This means there is no initial state that can possibly lead to this output. Thus, Program 3.11 is deemed noninterfering.

Programs such as Program 3.11 are rejected by security type systems and type-based dependency analysis systems because, due to overapproximation, they cannot determine that the insecure path leading to the return of variable x is actually unreachable. This example highlights a key advantage of symbolic execution, particularly that enforcement mechanisms based on symbolic execution are generally more precise.

Symbolic execution, however, suffers from scalability problems. The root of the problem is loops (or recursion) that depend on inputs. Since we do not know the actual value of the input, we do not know how many times the loop must be unfolded. This leads to path explosion, where the number of paths grows exponentially, making exhaustive exploration computationally infeasible.

Contributions

Having introduced the most common language-based mechanisms to enforce information flow control, we now proceed to discuss closely related works while presenting our contributions in this area.

Security Type systems. The most well-known approach for statically enforcing information flow control is a security type system [28]. Since the seminal work of Volpano et al. [18], which introduced the first type system proven to guarantee secure flow analysis based on Denning's lattice model [4], type systems have come a long way.

Security type systems have been employed to enforce information flow control in various contexts [19, 34, 80, 122, 126, 134, 184]. Many of these approaches are mainly focused on application-level security guarantees [18, 19, 34]. The security guarantees of security type systems may fail to hold if the underlying software stack or the operating system is compromised [126]. For example, if the attacker exploits some vulnerabilities to gain extra information about the running program (such as in the Heartbleed vulnerability), or if the host in a cloud provider is compromised and the attacker has gained privileged access to the machine, they can bypass the guarantees of the type system and access sensitive information. To address this challenge, Gollamudi and Chong [126] propose the use of trusted execution environments. By partitioning the program and putting all the components dealing with the sensitive information in a trusted execution environment, one can ensure that even if the underlying system is compromised, the guarantees of the trusted execution environments still protect the sensitive data.

The approach of Gollamudi and Chong [126] relies on trusted execution environments to enforce security against active attackers who can arbitrarily corrupt non-enclave code. They propose a calculus and security type system modeling SGX-like trusted execution environments. They employ a translation that analyzes the code and infer the appropriate placement of code and data into enclaves to ensure security guarantees against low-level attackers. Even though their approach guarantees confidentiality by safely storing sensitive data within the enclave, it does not consider robustness [23, 34] against active attackers.

In Chapter A, we propose a security type system for enforcing robustness for programs running on trusted execution environments. In our model, we assume the integrity of the program inside the enclave is intact, and that active attackers can only influence the execution of this enclave program through gateway method calls. Gateways are special methods within the enclave program that serve as access points for interaction between non-enclave and enclave programs. We consider two attacker models, one that manipulates data by modifying the parameters of gateway calls, and the other that influences execution by arbitrarily calling gateways, in any order or frequency. We prove that our type system can guarantee robustness against these attackers.

Enforcing data-dependent polices. For systems with more complicated security policies, we need more precise enforcement mechanisms, such as flow-sensitive security types [46] and dependent types [117]. Liquid information flow types [176] were developed on top of refinement type systems [103, 112, 182] to statically express and enforce data-dependent information flow policies. A refinement type enriches types with predicates that circumscribe the values described by the type. Liquid information flow types [176] rely on this concept and associate each security type with a predicate describing the values under which that security type holds. This approach allows the security type system to enforce expressive data-dependent information flow policies.

In Chapter E, we develop a security type system to enforce noninterference for P4 programs. P4 is a domain-specific language for programming programmable network devices, such as switches. Because of the data-dependent nature of network environments, simple *public* and *secret* labels are not enough to model the security requirements of P4 programs. To overcome this problem, we propose an idea similar to refinement security types and augment security types with intervals of the form $[a, b]$, where a and b are integers. These intervals express the range of values associated with the security type. By performing interval analysis on P4 types and making judgments based on the value of these intervals, we are able to express and enforce data-dependent policies, such as if the packet's destination address is in range `192.168.*.*`, its fields are *secret*, otherwise they are *public*.

One of the drawbacks of many security type systems is the burden they impose on the programmer to modify and annotate the code according to a specific security policy, effectively shaping and constraining how the program is written. This limits the applicability of security type systems, as it essentially requires a programmer who is also a security expert and knows the security requirements of the system.

Policy-agnostic enforcement mechanisms have been developed to reduce this burden [96, 97, 131]. In this approach, the policy is specified once and managed by the runtime, eliminating the need to repeat the policy checks throughout the program code [96].

We adopted a annotation-free approach in our security type system for P4 programs described in Chapter E. It uses programmer-specified input (output) policies to derive the security types of the initial (final) states. In this approach, the programmer can omit policy-related annotations from the code and instead associate policies with the initial and final states.

Symbolic execution. In addition to type systems, various other methods have been proposed over the years to enforce information flow control, including abstract interpretation [33], model checking [93], symbolic execution [90], relational logics [42, 83, 84], and theorem proving [39].

Balliu et al. [90] proposed EnCoVer, a framework for verifying information flow security in programs by combining symbolic execution and epistemic logic. They rely on symbolic execution to extract a bounded model of program behavior in the form of a *symbolic output tree* (SOT). A symbolic output tree is a structure that represents the possible outputs of a program based on different symbolic inputs. Like a symbolic execution tree, each path in the symbolic output tree corresponds to a unique sequence of nodes and path conditions, but with each node representing an observable output's value, it focuses on how variations in input conditions lead to different outputs. They formalize noninterference using epistemic logic and present an algorithm to reduce the epistemic model over the SOT to a first-order formula that can be checked via an SMT solver.

In Chapter C, we extend the approach of EnCoVer to enforce dynamic policies. We use symbolic execution to extract a symbolic output tree from the program's code and represent the dynamic security policies as first-order formulas over the nodes of this SOT. The program's security is then verified against various attacker models using an SMT solver. This approach allows us to have a more precise verification compared to the type systems, because the SOT not only prunes the unreachable paths, but also precisely tracks and records the active security policy at each node in the SOT.

Enforcing IFC in database-backed programs. A significant amount of research has been conducted into enforcing information flow control for database-backed applications [41, 52, 59, 65, 109, 122]. The main challenge is to track information flows across the program-database boundary, reconciling database access control mechanisms with program-level information flow control mechanisms. Guarnieri et al. [163] proposed Daisy, a security monitor for database-backed applications. It tracks the dependencies between program variables and database tuples by leveraging the Disclosure Lattice [99, 104] supporting for both column-level and row-level policies. Existing works in this domain have focused solely on conjunctive policies. However, disjunctive policies are a natural choice in databases, as they align with real-world scenarios where information is stored in databases and ethical wall policies need to be enforced based on conflict-of-interest classes. In Chapter D, we address this issue by relying on the Determinacy Quantale and query determinacy to specify and verify disjunctive policies in terms of database queries and views.

Hunt and Sands [85] proposed a type-based dependency analysis that is parametric in the choice of the security lattice. In their approach, the powerset of program variables is used as the security lattice, and a principal type is derived, from which all other types (for any choice of lattice) can be inferred. This approach provides a clean separation between analysis and policy, allowing the dependency analysis to be performed only once and then verified against various policies.

To enforce disjunctive policies for database-backed programs in Chapter D, we adapted the type-based dependency analysis of Hunt and Sands [85] and modified it to keep track of the dependencies of program paths separately. Instead of merging the dependency sets after `if` statements, we kept them as separate sets. In this approach, the result of the dependency analysis is a set of sets of dependencies, each set representing the dependencies of a possible run. When verifying the program's security with respect to disjunctive policies, we ensure that the dependencies of each path are secure under at least one of the disjuncts of the policy.

3.3 Application

Despite significant progress in language-based information flow control research over the past two decades in defining security policies, semantic conditions, and enforcement mechanisms, their application to real-world programming languages and systems still faces many challenges. Popular programming languages do not support information flow control out of the box, and designing sound language-based mechanisms for them is often a complex and difficult process. In this section, we focus on practical language-based information flow control tools developed for mainstream languages.

Contributions

We present our contributions to this area and discuss the closely related work.

Java language. The Java programming language has been the focus of many works on enforcing information flow control. The Jif compiler [19], developed by researchers at Cornell, is one of the initial attempts to integrate information flow control into Java programs. Jif's security type system was built using the Decentralized Label Model (DLM) [20] and is capable of statically enforcing confidentiality and integrity information flow policies.

We leverage Jif in Chapter A to develop J_E, a programming model for partitioning Java programs into enclave and non-enclave parts. The idea is that the program is annotated by the programmer, and these annotations are used to partition the program. For example, the code and the data of the classes annotated with `@Enclave` are placed inside the enclave, while all the other classes will remain outside. The enclave and non-enclave environments are then put into two separate execution environments, where they communicate with each other via Java RMI [203]. J_E uses security annotations, such as `@Secret`, to mark *secret* variables. The security type system uses this annotations to verify the security of the partitioned program. Our implementation relies on Jif [19] to implement our proposed type system. The partitioned program is translated to Jif, type checked to verify its security, and then transformed to use RMI communication. Drawing on the guarantees of the type system, we can be confident that the partitioned program satisfies robustness against active attackers who control the non-enclave environment.

There has been a long line of work focused on verifying static information flow policies by automated theorem proving [39, 82, 90, 95, 135]. Balliu et al. developed ENCoVer [90], which uses Java Pathfinder [30] to extract program dependencies from Java programs and verifies static noninterference policies by means of SMT solving and model checking. In Chapter C, we use a similar approach to develop DynCoVer, which is a tool designed to verify dynamic policies in Java programs. DynCoVer uses Symbolic Pathfinder [102] to extract a symbolic output tree from Java programs. Each node in this tree captures an observable output of the program along with the information about the active policy at the time the output occurred. DynCoVer uses this information to generate a quantifier-free first-order formula for each node, capturing the security policy, the observable traces, and the program model. The satisfiability of these formulas is then checked via the Z3 solver [58], verifying the security of the program at each output point. We evaluate DynCoVer on a benchmark suite of various dynamic policy scenarios and attacker models, as well as on a social network use case that investigates scenarios such as publishing public posts, following users, and blocking them.

P4 language. P4BID [192] developed a security type system for Core P4 [180]. They extend P4 data types with security labels and prove that well-typed programs satisfy noninterference. However, P4BID only supports simple security policies and does not take into account tables or external functions. To address these limitations, in Chapter E, we develop Tap4s, a prototype tool implementing a P4 security type system enriched with interval analysis. Our formalization of P4 closely resembles the real-world P4 syntax and semantics, enabling our prototype tool to enforce security policies on P4 programs with minor modification. The interval analysis coupled with the type checking enables Tap4s to specify and enforce data-dependent policies. In addition, Tap4s's implementation is annotation-free, meaning that there is no need to annotate the target P4 program with types. Instead, we define separate input and output policies, the type checker then uses the input policy to initialize program types (e.g. for packet headers, variables, metadata), propagates these types through the program by following the typing rules, and ensures that the final types are in line with what is allowed in the output policy.

4 | Thesis Results

> “ *The Answer to the Great Question... Of Life, the Universe and Everything... Is... Forty-two, said Deep Thought, with infinite majesty and calm.* ”
>
> – **Douglas Adams**, The Hitchhiker’s Guide to the Galaxy

In this section, we provide an overview of the papers included in this thesis, along with a statement of the author’s contributions. The thesis consists of five papers, labeled Papers A through E. Four of these papers — Papers A to D — have already been published in peer-reviewed conference proceedings, while Paper E is currently *submitted* to a peer-reviewed conference and is undergoing the review process.

Table 4.1 summarizes the contributions of the thesis author to each of the papers included in this thesis. In the table, ● indicates that the task was completed entirely by the thesis author, ◐ denotes that the task was completed in collaboration with coauthors, and ○ indicates that the thesis author did not contribute to the task.

Table 4.1: Summary of contributions

	Specification	**Mechanism**	**Proof**	**Implementation**	**Evaluation**
Paper A	◐	●	●	○	◐
Paper B	◐	●	●	○	◐
Paper C	◐	●	●	●	●
Paper D	◐	◐	◐	●	●
Paper E	◐	◐	○	●	◐

The formatting and style of the papers have been adjusted to align with the overall format of the thesis, but their content remains unchanged from the original publications. Additionally, the bibliography for all the papers and the introductory part of the thesis has been unified and is included at the end of the thesis.

Paper A: Language Support for Secure Software Development with Enclaves

> Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi. "Language support for secure software development with enclaves" In *2021 IEEE 34th Computer Security Foundations Symposium* (CSF), pp. 1-16. IEEE, 2021.

The paper presents J_E, a programming model for Java that relies on a security type system to enforce robustness against realistic active attackers in the context of trusted execution environments (TEE). A trusted execution environment, also referred to as an enclave, is a secure area within the main processor. The operations within the enclave are opaque to the host operating system, as guaranteed by the processor. This ensures that sensitive data and operations are protected, isolated, and executed securely, even in the presence of a compromised host.

In this paper, we highlight the importance of considering realistic active attacker models in the trusted execution environment setting. To this end, we consider two attacker models: the havoc active attacker (HAA), who is limited to modifying parameters passed through method calls to the enclave program, and the havoc reordering active attacker (HRAA), who is capable of controlling the order and frequency of enclave method calls. Both of these active attackers model scenarios in which the non-enclave environment is compromised and under active attacker control. We investigate robustness in enclave-enabled programs, ensuring that these non-enclave active attackers cannot learn more than a passive attacker.

We provide a core calculus for J_E and propose a semantic security condition to capture robustness, ensuring that HAA and HRAA attackers do not learn more information than a passive attacker. We statically enforce this security condition using a security type system that accounts for both confidentiality and integrity labels and incorporates special operators to model declassification and endorsement. We prove the soundness of this type system, guaranteeing that well-typed programs are robust with respect to HAA and HRAA attackers.

On the implementation side, J_E relies on user-specified annotations to automatically partition the program into enclave and non-enclave parts. It uses Java RMI [203] to enable remote communication between the two partitions and Jif [19] to type-check the resulting program and ensure it satisfies robustness.

Statement of contribution. The idea of using annotations to partition a program into enclave and non-enclave parts was proposed and developed collaboratively by all authors, resulting in the development of the J_E programming model. The formalization of J_E, the development of the type system, and the proofs were carried out by the author of the thesis in collaboration with Musard. All authors contributed to the writing of the paper.

Paper B: Enclave-Based Secure Programming with $\mathbf{J_E}$

> Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi. "Enclave-based secure programming with $\mathrm{J_E}$" In *2021 IEEE Secure Development Conference* (SecDev), pp. 71-78. IEEE, 2021.

In this paper, we detail the implementation of J_E, which was formally introduced in our previous paper. To guarantee robustness and enforce security against active attackers, we proposed a security type system, detailed in our CSF paper, "Language Support for Secure Software Development with Enclaves."

We implemented J_E on Intel SGX, by relying on the Jif [19] type system and the SGX-LKL framework [204]. The design goals of J_E were to: (1) Abstract away the SGX implementation details by allowing the programmer to easily specify the parts of the program that must run inside the trusted execution environment. (2) Provide simple means to specify and enforce security policies.

To achieve this goal, we provided a set of security annotations and operators, such as `@Secret` specifying *secret* variables, `@Enclave` specifying enclave classes, and `declassify` and `endorse` operators to define declassification and endorsement operations, respectively. The J_E compiler relies on these annotations to automatically partition the program, generate the logic for enclave management, and convert the program to Jif in order to verify the information flow policies.

Initially, J_E analyzes the program and, based on the annotations, splits it into two partitions: the enclave and the non-enclave partitions. This process puts the classes annotated with `@Enclave` and all their dependencies in the enclave partition, while keeping all the other classes in the non-enclave partition. Next, the enclave partition is converted into an equivalent Jif [19] program, and Jif's type checker is used to statically enforce our robustness condition. Finally, the J_E compiler generates a remote interface for all the gateway methods using Java RMI [203] to enable communication between the enclave and the non-enclave partitions.

We evaluate J_E on several use cases from the literature, including a battleship game, a secure event processing system, and a popular processing framework for big data, showing that it can correctly handle complex cases of program partitioning and robust information flow control.

Statement of contribution. This paper focused on showcasing the programming model introduced in Paper A. The author of the thesis was involved in developing the use cases to better showcase J_E. Furthermore, the thesis author, in collaboration with Musard, contributed to the sections explaining the theoretical contributions of the paper, including the attacker models and the security guarantees of the implementation. All authors contributed to the writing of the paper.

Paper C: Dynamic Policies Revisited

> Amir M. Ahmadian and Musard Balliu. “Dynamic policies revisited” In *2022 IEEE 7th European Symposium on Security and Privacy* (EuroS&P), pp. 448-466. IEEE, 2022.

This paper presents a knowledge-based security framework for dynamic policies. We revisit existing knowledge-based security conditions for dynamic policies, highlight their limitations, and emphasize the role of the attacker and their capabilities in the security conditions. By examining this relationship, we propose a new framework that establishes a clear connection between dynamic policies and the associated attacker model.

We provide formal definitions of attacker knowledge for three categories of attackers: perfect recall (who remembers all past information), bounded memory (who has limited storage), and forgetful attackers (who forget information at policy change). We introduce the novel notion of policy consistency, ensuring that policy changes do not occur if the attacker already possesses the information the new policy intends to protect. Building on policy consistency, we define the concept of policy repair which adjusts inconsistent policies to produce a consistent policy that respects the attacker’s current knowledge. Together, these components — attacker models, policy consistency, and policy repair — allow us to build a security framework for dynamic policies. This framework presents a unified approach to enforcing dynamic policies and can address the *facets of dynamic policies* discussed in the literature [114].

We develop a verification algorithm that combines symbolic execution with SMT solving to verify policy compliance, identify policy inconsistencies, and generate repaired policies. Using symbolic execution, we extract the program’s model and generate a quantifier-free first-order formula representing both the security policy and the program model. The unsatisfiability of this formula, verified through an SMT solver, confirms the security of the program.

We implement this approach in DynCoVer, a prototype tool designed to verify dynamic information flow policies in Java programs. We evaluate DynCoVer on a micro benchmark suite covering various dynamic policy scenarios, as well as a social network use case, demonstrating the practical effectiveness of our framework in real-world scenarios.

Statement of contribution. The theoretical parts of the paper were developed collaboratively by Musard and the thesis author. We explored the idea of a novel security condition to address the limitations of existing works, and, in collaboration with Musard, further refined this idea into a security framework capable of accommodating various facets of dynamic policies. The prototype tool DynCoVer, the benchmark, and the use case were developed by the author of the thesis. Both authors contributed to the writing of the paper.

Paper D: Disjunctive Policies for Database-Backed Programs

> Amir M. Ahmadian, Matvey Soloviev, and Musard Balliu. "Disjunctive policies for database-backed programs" In *2024 IEEE 37th Computer Security Foundations Symposium* (CSF), pp. 388-402. IEEE, 2024.

Disjunctive policies are a special family of security policies that restrict access to information based on conflict-of-interest classes. They specify conflict-of-interest classes and define a so-called ethical wall, permitting the flow of information from one class only if the observer has not accessed any data from conflicting classes. The Quantale of Information (QoI) [181] was proposed as a semantic model to describe such disjunctive dependencies, extending the previously established Lattice of Information.

Building on QoI, and motivated to provide a formal model for reasoning about disjunctive dependencies in database-backed programs, we introduce the Determinacy Quantale (DQ), a query-based lattice structure designed to represent disjunctive dependencies in databases.

Relying on QoI and DQ, we define an extensional security condition tailored to database-backed programs. This condition ensures that the knowledge conveyed by the program's execution adheres to at least one of the disjuncts specified by the policy, indicating that the execution relied on at least one set of non-conflicting classes.

We propose a type-based dependency analysis for a simple imperative language that includes simple database queries. This analysis tracks row and column level database dependencies while also accounting for disjunctions introduced by the program's control flow.

To validate the practical applicability of this approach, we developed DiVerT, a prototype tool that enforces query-based disjunctive policies leveraging the proposed type-based dependency analysis. We evaluate DiVerT on a functional test suite and various realistic use cases.

Statement of contribution. The idea of adapting the Quantale of Information to the database setting was proposed by the author of the thesis. This was further developed in collaboration with Musard and Matvey into the Determinacy Quantale. The proposed type-based dependency analysis was designed and developed by the author of the thesis. The development of the query abstraction method and the soundness proofs was a joint effort among all the authors. The prototype tool DiVerT, the functional test suite, and the use cases were developed by the author of the thesis. All authors contributed to the writing of the paper.

Paper E: Securing P4 Programs by Information Flow Control

> Anoud Alshnakat, Amir M. Ahmadian, Musard Balliu, Roberto Guanciale, and Mads Dam. “Securing P4 programs by information flow control” **Manuscript**

In the area of software-defined networking, Programming Protocol-independent Packet Processors (P4) is a domain-specific language that is the leading standard for programming programmable network devices, such as switches.

In this paper, we propose a security type system to ensure information flow security in P4 programs. The goal is to protect the confidentiality of sensitive network information, such as congestion information and MAC addresses, and preventing their leakage to public networks. Developing a security type system for P4 presents two main challenges: (1) P4 programs are highly data-dependent, meaning that packets are processed based on the values of their various fields, such as address, time-to-live, or ports. (2) The existence of non-native components, such as externs and tables.

Externs are functions not implemented in P4, such as hash or checksum functions, where their behavior depends on the underlying device architecture. Tables are statically unknown components that contain the routing information and are configured and updated at runtime by the network.

To address the first challenge, we extend security types with simple intervals of the form $\langle a, b \rangle$, where a and b are integers. These interval-based security types enable us to represent data-dependent policies, such as “a packet is only *secret* if its source and destination IP addresses fall within the range `192.168.*.*`.” The security type system then uses these intervals to perform a more permissive, data-aware type checking of the program.

To address the second challenge, we rely on user-defined contracts which capture a bounded model of the behavior of the externs and tables. When the type system encounters these components, it uses the contracts to drive the analysis.

We formalize this type system and prove it sound, guaranteeing that well-typed P4 programs satisfy noninterference. The proposed type system has been implemented in a prototype tool, Tap4s, and evaluated on a functional test suite and five use cases.

Statement of contribution. The idea of using a security type system for P4 security, as well as the rules of the type system, was developed collaboratively by all the authors. The implementation of Tap4s was undertaken by the author of the thesis, while the use cases were developed in collaboration with Anoud. All authors contributed to the writing of the paper.

5 | Conclusion and Future Work

> “*Schiller. A German dramatist of three centuries ago. In a play about Joan of Arc [...] said, ‘Against stupidity, the gods themselves contend in vain.’ I’m no god and I’ll contend no longer. Let it go, Pete, and go your way. Maybe the world will last our time and, if not, there’s nothing that can be done anyway.*”
>
> – **Isaac Asimov**, The Gods Themselves

This thesis contributes to the state of the art of language-based information flow control in theoretical and practical domains. Throughout this thesis, we have addressed the following research questions: (RQ1) How can non-trivial security policies be defined to effectively address specific real-world scenarios? (RQ2) What are the appropriate attacker models for scenarios where simple attacker models fail to adequately account for the full range of security threats faced by the system? (RQ3) How can we develop mechanisms to enforce realistic information flow security conditions in a sound and precise manner? (RQ4) How can information flow control techniques be effectively and manageably applied to real-world programming languages?

To address RQ1, we investigated security conditions for settings with dynamic security policies and proposed a knowledge-based security framework that emphasizes the critical role of the attackers and a policy’s consistency with the attacker’s knowledge in the definition of security condition for dynamic policies. In addition, we proposed the Determinacy Quantale, a new semantic model capable of expressing disjunctive policies in terms of database views. It enabled us to develop a security condition capable of verifying the security of database-backed programs with respect to disjunctive policies.

The thesis also examined a range of passive and active attackers in accordance with RQ2. To guarantee the robustness of programs running on trusted execution environments, we investigated two types of realistic active attackers, one that manipulates the data passed to the enclave, and another that can manipulate the control flow of the program. We proposed a security type system that can guarantee a well-typed program running inside the enclave is robust with respect to these attackers. Furthermore, to highlight the importance of attacker capabilities on security conditions in settings with dynamic policies, we investigated three types of attackers: perfect recall, bounded memory, and forgetful, and accommodated them into our dynamic policy security framework.

RQ3 has been addressed throughout this thesis by investigating a range of static language-based mechanisms to enforce information flow control. We examined security type systems for enforcing robustness against active attackers for programs running inside trusted execution environments. We augmented security types with interval analysis to develop a permissive, yet sound type system capable of expressing and enforcing data-dependent policies for P4 programs. We utilized a disjunctive type-based dependency analysis to capture the disjunctive dependencies of database-backed programs in terms of database queries, enabling us to develop an enforcement mechanism capable of verifying the security of these programs with respect to disjunctive policies. Finally, we combined a symbolic execution-based approach with epistemic logic to express and enforce dynamic policies in Java programs.

RQ4 has been addressed throughout this thesis by implementing and evaluating the proposed enforcement mechanisms on Java and P4 programming languages. We developed a parser and a type checker for the P4 programming language, capable of expressing and statically enforcing data-dependent security policies on P4 programs. We implemented our proposed symbolic execution-based approach in a tool named DynCoVer to ensure the security of Java programs in the presence of dynamic policies. In addition, we developed a programming model and a security type system capable of automatically partitioning Java programs into enclave and non-enclave parts, ensuring that the partitioned program is robust against active attackers.

Future work

From a broader perspective, the field of information flow control presents many opportunities for advancement in both theory and practice. There is growing potential for extending existing methods and expanding their applicability to increasingly complex, real-world environments.

One natural target is the heterogeneous software systems, where code from multiple sources interacts in complex ways. The key challenge here is adapting language-based methods to large-scale codebases spanning across various programming languages and platforms. This challenge becomes even more pronounced when the heterogeneous program interacts with databases. In such scenarios, it is necessary to track information flows across program code, query languages, and storage backends. Future research should focus on developing unified, cross-boundary enforcement mechanisms that ensure security guarantees are maintained even when components are developed with different programming languages and libraries.

Another important research avenue is cloud-based systems, where applications are distributed across various platforms and containers. In this setting, information flow control can ensure that sensitive data does not flow to unauthorized parts of the infrastructure. However, this is a challenging task due to the complexity of cloud-based systems, as data moves through many stages, gets processed by different services, logged for debugging, and transferred between containers. Orchestration tools, such as Kubernetes, manage the deployment and communication of these services. information flow control could help enforce strict data flow policies, ensuring that sensitive data is only accessible to authorized services. Monitoring tools that track system performance often collect logs. IFC methods could prevent sensitive information from being included in these logs, reducing the risk of accidental data leakage. Automated deployment pipelines, which facilitate rapid software updates, present another challenge. IFC could verify that incremental updates do not introduce new ways for sensitive information to leak.

Finally, the development of robust IFC toolchains remains an open challenge. Future efforts should focus on creating mature language-based tools for mainstream programming languages and improving their scalability and usability for large codebases. The ultimate goal in this area should be to create a seamless developer experience, enabling precise data-flow tracking and stronger security guarantees without imposing significant overhead or requiring drastic changes to existing development practices. This line of research highlights the need for refined enforcement mechanisms, scalable toolchains, and user studies to guide the practical adoption of IFC in real-world applications.

On a more focused level, future research may improve and extend the approaches proposed in this thesis in several directions.

J_E programming model can be extended to support more Java features, handling the automated partitioning and type checking of Java programs beyond the core subset formalized in J_E. Furthermore, the type system of J_E can be generalized to support multiple enclaves and nonmalleable information flow [134]. Future avenues of research on disjunctive policies can be enhancing our type-based dependency analysis to make it more precise by eliminating the dependencies of unreachable paths, supporting database operations such as insert and update, and adapting a more realistic database model capable of expressing advanced features such as triggers and procedures. DynCoVer can be updated to support newer versions of Java, while the underlying security framework can be extended to take into account integrity policies as well as the interplay between integrity and confidentiality in the presence of dynamic policies. Future directions for improving Tap4s and its data-dependent type system include adapting a more permissive security condition to support declassification, improving the contract language for a more precise modeling of externs such as cryptographic primitives, and extending the type system to account for side channels.

Papers

Paper A

Language Support for Secure Software Development with Enclaves

Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi

Abstract

Confidential computing is a promising technology for securing code and data-in-use on untrusted host machines, e.g., the cloud. Many hardware vendors offer different implementations of Trusted Execution Environments (TEEs). A TEE is a hardware protected execution environment that allows performing confidential computations over sensitive data on untrusted hosts. Despite the appeal of achieving strong security guarantees against low-level attackers, two challenges hinder the adoption of TEEs. First, developing software in high-level managed languages, e.g., Java or Scala, taking advantage of existing TEEs is complex and error-prone. Second, partitioning an application into components that run inside and outside a TEE may break application-level security policies, resulting in an insecure application when facing a realistic attacker.

In this work, we study both these challenges. We present J_E, a programming model that seamlessly integrates a TEE, abstracting away low-level programming details such as initialization and loading of data into the TEE. J_E only requires developers to add annotations to their programs to enable the execution within the TEE. Drawing on information flow control, we develop a security type system that checks confidentiality and integrity policies against realistic attackers with full control over the code running outside the TEE. We formalize the security type system for the J_E core and prove it sound for a semantic characterization of security. We implement J_E and the security type system, enable Java programs to run on Intel SGX with strong security guarantees. We evaluate our approach on use cases from the literature, including a battleship game, a secure event processing system, and a popular processing framework for big data, showing that we correctly handle complex cases of partitioning, information flow, declassification, and trust.

A.1 Introduction

Confidential computing includes recent technologies to protect data-in-use through isolating computations to a hardware-based Trusted Execution Environment (TEE). TEEs provide hardware-supported enclaves to protect data and code from the system software. Over the past few years, an array of TEE designs has been developed, including Intel's Software Guard Extensions (SGX) [101, 108], ARM TrustZone [60], MultiZone [175] and others [125, 129, 159, 168, 172]. Using TEEs, data can be loaded securely in plain text and processed at native speed within an enclave even on a third-party machine. SGX is a TEE implementation from Intel which has been successfully used in a number of industry products [197, 198].

The issue with confidential computing Supporting confidential computing in a way that is both accessible for developers and technically secure is still an open problem.

First, **seamless integration** of enclave programming into software applications remains challenging. For example, Intel provides a C/C++ interface to the SGX enclave but no direct support is available for managed languages. As managed languages like Java and Scala are extensively used for developing distributed applications, developers need to either interface their programs with the C++ code executing in the enclave (e.g. using the Java Native Interface [202]) or compile their programs to native code (e.g. using Java Native [199]) relinquishing many advantages of managed environments.

A second aspect concerns security with **realistic attackers**. Standard security analysis of code protects against passive attackers, as common for untrusted/buggy code executing in a single trusted host [28]. Yet, with enclaves, programs run in a trusted environment within an untrusted host: the attacker can control the untrusted environment to cause additional leaks of sensitive information through the interface between the two environments. An active attacker may force the enclave program to violate the security policy by compromising the integrity of inputs at the interface or by controlling the execution order of interface components, e.g. to trigger execution paths and side effects that were not possible in the original program. Current research adopts Information Flow Control (IFC) to ensure that the code within an enclave does not leak sensitive information to the non-enclave environment [119, 126, 178]. This research, however, either takes a limited view of a passive attacker that only observes the data leaving the enclave, or it incorporates the effects of an active attacker into the execution semantics and the security condition, thus requiring additional verification effort to secure enclave programs.

These challenges lead us to the following key research questions addressed by the paper: (a) How to enable seamless integration of enclaves and managed languages like Java? (b) What is the right security model for realistic enclave attacks and how to statically verify the security of enclave programs with respect to these attacks? (c) How to harden state-of-the-art IFC tools to verify security in the TEE context?

(d) How to demonstrate feasibility via realistic use cases?

Accessible and secure confidential computing To address the questions above, we present J_E, a programming model that offers language-level integration of enclave technology. We leverage IFC to secure applications running within TEEs and propose a security condition and an enforcement mechanism targeting realistic attackers in the context of TEEs.

To support **seamless integration** of enclave programming, J_E defines a programming model for developers of enclave applications as a Java extension based on three annotations and two operators. Programmers can execute computations securely just by adding the `@Enclave` annotation to classes to be placed inside the enclave and the `@Gateway` annotation to the methods accessible from the non-enclave environment. Also, programmers can label secret data with `@Secret` annotations and control the release of secret information to the non-enclave environment via the `declassify` operator as well as the influence of untrusted information from the non-enclave environment via `endorse` operator. J_E builds on the Java information flow (Jif) security-typed language [19] to statically check confidentiality and integrity policies of code running within an enclave. A J_E program is automatically translated into an equivalent Jif program.

To provide security against **realistic attackers**, our key observation is that a program is secure in the presence of enclave active attackers iff the program does not leak additional sensitive information as compared to executing the program in the presence of a passive attacker. Inspired by the line of works on IFC for distributed applications [24, 34], we propose *robustness*, a semantic characterization of security for enclaves for two realistic active attackers. Robustness captures the interplay between the integrity of untrusted data coming from the non-enclave environment and the confidentiality of secret data within the enclave, ensuring that an active attacker does not learn additional information. We enforce robustness with a security type system for a core of J_E to check that the active attackers' control over inputs at the enclave interface and over the execution order of interface components does not enable them to learn more sensitive information than a passive attacker who merely observes the outputs at the interface. Importantly, our security type system can be used to check robustness for the partitioned programs leveraging existing verification efforts for the original program before partitioning. In contrast to existing type systems for robust declassification [24, 34, 80], our security type system is flow sensitive [46] which poses additional challenges with declassification policies and enclave attackers.

To validate the design of J_E, we verify several case studies from the literature. To validate our approach to security for enclaves, we prove our security type system sound with respect to robustness, showing that it accepts only secure programs. The implementation of J_E and the case studies discussed in the paper are publicly available [1]. In summary, this paper makes the following contributions:

- We present J_E, a programming model that seamlessly supports computations inside an enclave. In this model, programmers use annotations to identify the computations to be executed inside the enclave.
- We provide a core calculus for J_E and propose a semantic security model to capture the essence of information flows in the presence of active enclave attackers.
- We enforce robustness statically via a security type system, which we prove sound for programs implemented in J_E.
- We evaluate the applicability of J_E using different case studies.

The paper is structured as follows. Section A.2 introduces enclave technology and the different approaches for enclave software development. Section A.3 overviews the J_E design. In Section A.4, we present a security framework for J_E. Section E.7 describes the implementation. The evaluation is in Section A.7. Section A.8 presents related work and Section E.9 concludes.

A.2 Trusted Execution Environments in a Nutshell

In this section, we introduce TEEs and we refer to Intel SGX, as a concrete implementation. TEEs make use of dedicated processor instructions and modified memory access mechanisms to enable private computations. The principle behind a TEE is to provide applications with memory isolation. With the help of the dedicated instructions, applications can create private memory regions known as *enclaves*. An enclave is essentially a reserved area in the system memory protected by the CPU. Data inside an enclave is protected from high privilege software such as OS, VMM and BIOS. This design leads to two distinct execution environments – the enclave environment (trusted environment) and the non-enclave environment (regular system memory).

Intel Software Guard Extensions (SGX) [101, 108] is an enclave implementation from Intel introduced with the sixth generation Intel core and Intel Xeon E3 v6 server processors.

The SGX Mechanism

In SGX enabled systems, the BIOS reserves a contiguous part of the DRAM as Processor Reserved Memory (PRM). Currently, the size of the PRM is limited to

[1] https://prg-grp.github.io/je-lang

128 MB. Out of this PRM, a memory region of about 90 MB is used as Enclave Page Cache (EPC) which stores the enclave code and data. The content of the enclave memory is encrypted using the Memory Encryption Engine (MEE) and is decrypted only when inside the CPU. This solution protects the enclave data from an attacker having physical access to DRAM. An application creates a new enclave using the `ECREATE` instruction. The enclave memory can only be accessed from inside the enclave. The CPU rejects any attempt to access the enclave memory from the non trusted environment. SGX provides a Remote Attestation (RA) service to verify the integrity of the data loaded inside the enclave. We refer to Costan and Devadas [124] for a detailed description of SGX.

Software Development with SGX

There are two main development methodologies used for programming with SGX enclaves. One is to use the C/C++ interface provided by Intel along with the Microsoft Visual Studio IDE. In this approach, a programmer writes their application in C/C++, programs to be executed inside and outside the enclave are written in two separate projects and together they form a complete application. The application is compiled using the Intel compiler integrated into the IDE. The benefit of this approach is that the developer can keep the trusted code base (TCB) minimal. Unfortunately, no such support is available for applications written in managed languages. The other approach is to use systems based on library OSes [113, 121, 147, 148, 166]. They allow running unmodified applications inside the enclave. The application and the library OS are compiled together into an image file and the host OS runs the image inside the enclave.

Attacker models with enclaves

In this work, we consider two attacker models: a **passive** attacker and (two variants of) an **active** attacker. A passive attacker cannot modify the execution state of the program (inside or outside the enclave) and can only observe the entire execution state of the non-enclave environment. The rationale is that a developer can partition the original program into an enclave component and a non-enclave component, assuming that the partitioned program will be as secure as the original program, so long as the enclave component contains the secret data as well as any code that accesses these data.

The active attacker model extends the threat model considered in [119, 126] and considers that an active attacker can arbitrarily alter the state of the non-enclave environment and hence it can modify the data exchanged between the two environments. An active attacker can influence the execution of the enclave component either by compromising the integrity of inputs at the interface or by controlling the execution order of the interface components. Both cases may lead to triggering execution paths and side effects that were not possible in the original program, thus enabling learning more sensitive information than a passive attacker.

For both attacker models, we only assume that the enclave hardware and software, except for the J_E source program, are trusted. The J_E uses static analysis to check that a passive attacker learns no more information about secret data than allowed by the original program, and to check that an active attacker learns no more information than a passive attacker. Denial-of-service attacks, arbitrary destruction of enclaves by the host OS, hardware side-channel attacks, and power analysis attacks are out of the scope of this paper. We also ignore covert channels including termination and timing. While termination channels have well-understood information leakage bounds [55], orthogonal techniques such as constant-time programming [120] and predictive mitigation mechanisms [70] can help mitigating some of these concerns.

A.3 J_E Design

We propose a language design that supports computations inside an SGX enclave. Our approach, J_E, relieves the programmer from dealing with the lower-level details of SGX, such as enclave creation, initialization, and destruction. Moreover, J_E supports security annotations to define application-level policies that are subsequently interpreted as information flow policies and are verified via a security type system.

```
@Enclave
class Encrypter {

    @Secret static String key;

    @Gateway
    public static String encrypt(String plaintext) {
        String cipher = encode(plaintext, key);
        return declassify(cipher);
    }
}
```

Program A.1: Language constructs

J_E Annotations

J_E extends the Java language with annotations to specify the parts of the program that run inside SGX as well as the sensitive data. We demonstrate these features using a routine to securely encrypt data (Program A.1). Encryption is based on a key that should be kept secret; hence the whole encryption operation should run within the enclave. J_E supports the following annotations to define security policies.

```java
@Enclave
class Encrypter {

    @Secret static String key;

    @Gateway
    public static String encrypt(String plaintext) {
        String cipher = "";
        String plaintextE = endorse(plaintext);
        if (plaintextE.length() == 8) {
            String var1 = encode(plaintextE, key);
            cipher = declassify(var1);
        }
        return cipher;
    }
}
```

Program A.2: The endorse operator

Class annotation `@Enclave` A programmer can annotate any class with the `@Enclave` annotation. Hereafter, we refer to such classes as enclave classes. The code and the data that belong to enclave classes are placed inside the enclave. To ensure that data and computations related to encryption take place within the enclave, the class `Encrypter` in Program A.1 is annotated with the `@Enclave` annotation.

Field annotation `@Secret` A programmer can annotate fields of an enclave class with the `@Secret` annotation – we refer to such fields as *secret* fields. The annotation denotes that the field holds sensitive information. Any program construct influenced by a secret field will also be considered as secret in order to prevent flows of any sensitive data from the enclave to the non-enclave environment. In Program A.1, the `key` field is used as a key for encryption. The `key` field is annotated with the `@Secret` annotation to denote that its value must not be leaked to the non-enclave environment.

Method annotation `@Gateway` Static methods defined inside an enclave class can be annotated with the `@Gateway` annotation. We refer to these methods as `@Gateway` methods. These methods act as an interface between the enclave and the non-enclave environment: the execution switches from the non-enclave environment to the enclave when a gateway method is called from the non-enclave environment. In Program A.1, the `encrypt` method is annotated with the `@Gateway` annotation to ensure that encryption is performed within the enclave. The `encrypt` method encrypts the `plaintext` argument with the secret key and returns the corresponding cipher text to the external environment. The return value of a gateway method should not be influenced by secret information.

Operator `declassify` The `declassify` unary operator downgrades a secret value into a public value. In Program A.1, the value in `cipher` (Line 8) is influenced by the secret field `key`. Hence, `cipher` is considered sensitive and cannot be returned to the non-enclave environment. However, the encryption can be considered secure as an observer cannot learn useful information by observing only the ciphertext. The `declassify` operator is used to control the release of sensitive information by explicitly declassifying a secret value provided it can be trusted (see the following paragraph).

Operator `endorse` The `endorse` operator endorses an untrusted value into a trusted one. A declassification can get triggered based on the specific value of a variable. As a result, a malicious user can influence the value of a variable to trigger the declassification. J_E ensures that only trusted values are declassified and application of the `declassify` operator does not depend on untrusted values. By default, the arguments to the gateway methods are considered untrusted as they are received from the non-enclave environment. In Program A.2, the gateway method `encrypt` accepts the `plaintext` argument from the non-enclave environment. We explicitly trust the identifier `plaintext` (Line 9). As a result, `plaintextE` is a trusted version of `plaintext` and the declassification no longer depends on an untrusted variable. Program A.2 is valid because the `var1` identifier is declassified (Line 12) and it is trusted. For this reason, the application of the `declassify` operator (Line 12) does not depend on an untrusted value. Also, the code in Program A.1 is invalid as the identifier `cipher` being declassified (Line 8) is untrusted.

Compilation phases

The first step of the compilation process involves automatically translating a J_E program into an equivalent Jif program. The generated program is verified using the Jif compiler. If the compilation succeeds, the code for data exchange between the enclave and non-enclave environments along with the initialization code is added. Finally, two JARs corresponding to each environment are generated. A detailed description of the compilation steps is provided in the technical report [185].

Execution Model

We execute J_E programs in two separate JVMs, one running in the external environment and one within the enclave. By default, a J_E program starts its execution inside the non-enclave environment. When a program running in the non-enclave environment calls a gateway method, the corresponding parameters are passed to the enclave-environment and the called gateway method executes inside the enclave.

When a gateway method is called from the non-enclave environment, the corresponding arguments are serialized and copied (deep copy) into the enclave. Hence, this semantics results in a distinct copy of each argument inside the enclave. To avoid inconsistent copies of an object in the two environments, once a gateway method is called, we prevent any further usage of all its arguments in the non-enclave environment.

$$
\begin{array}{rcl}
CL & ::= & \mathit{class}\ C\ \{\overline{f}, \overline{m}\}\ \mid\ \mathit{classEnclave}\ C\ \{\overline{f}, \overline{m}\} \\
m & ::= & \mathit{method}_{\phi}(\overline{p})\{S; \mathit{return}(e)\} \\
S & ::= & \mathit{skip}\ \mid\ \mathit{if}\ e\ \mathit{then}\ S_1\ \mathit{else}\ S_2\ \mid\ S_1; S_2\ \mid\ \mathit{while}\ e\ \mathit{do}\ S \\
 & & x := \mathit{declassify}(e) \mid\ x := e_G \mid\ C.f := e_G \\
e & ::= & x\ \mid\ v\ \mid\ e.f \mid\ e_1 \oplus e_2 \\
e_G & ::= & e \mid\ e.m(\overline{p}) \\
v & ::= & n\ \mid\ C\ \mid \mathit{unit}
\end{array}
$$

Figure A.1: Syntax of J_E

A.4 Security Framework

This section presents a security framework for reasoning about the confidentiality and integrity properties of enclave programs. We formalize a core of J_E by defining the syntax and semantics (Section A.4 and A.4). Drawing on IFC, we present a security model (Section A.4) with the following ingredients: a security policy specifying the parts of the program that contain secret/trusted information, and the parts that contain public/untrusted information (Section A.4); an attacker model specifying the capabilities of active and passive attackers (Section A.4); a security condition capturing a semantic characterization of security with respect to the program semantics, the attacker model, and the security policy. Finally, we present a security type system that enforces the security condition in a sound manner (Section A.4).

J_E Syntax

J_E is an imperative language extended with constructs for static classes, methods and fields as shown in Figure A.1. CL represents the list of class definitions in a program. We write $\overline{l}$ for a finite list of elements $l_1, \cdots, l_n$. We distinguish two types of classes: *normal* classes (*class*) which are executed in the non-enclave environment and *enclave* classes (*classEnclave*) which are executed in the enclave environment. Each class is defined with a list of fields $\overline{f}$ and methods $\overline{m}$. We assume classes $C \in Class$, methods $m \in Method$, and fields $f \in Field$ are uniquely identified.

We define a method by the list of formal parameters $\overline{p}$ and method body S, and use the annotation $\phi \in \{G, NG\}$ to indicate whether (G) or not (NG) a method is a gateway. The method body S is a sequence of commands that is executed when the method is called, followed by $return(e)$ for the return value of the method. J_E distinguishes two types of expressions, side-effect free expressions e including variables $x \in Vars$, values $v \in Val$, fields accesses $e.f$, and binary operations $\oplus$, and side-effectful expressions e_G which extend e with method calls $e.m(\overline{p})$. Commands S include standard features such as assignment to class fields and variables, conditionals, loops, and sequencing. Command $x := declassify(e)$ is semantically equivalent to an assignment and it is used for declassification. Command $x := e_G$, assigns the result of evaluating e_G to the variable x. If e_G is a method call, e.g. the assignment $x := C.m(\overline{p})$ denotes a call to method m of class C with actual parameters $\overline{p}$, and stores the return value in variable x. Similarly $C.f := e_G$ assigns the result of evaluating e_G to the field f of class C. Values are integers $n \in \mathbb{Z}$, class identifiers or unit.

J_E Operational Semantics

We define the (big-step) operational semantics of J_E. A configuration $\langle S, \mathcal{M}, \mathcal{H}\rangle$ consists of a command S, a memory $\mathcal{M} = Vars \rightarrow Val$ mapping variables to values, and a heap $\mathcal{H} = Class \times Field \rightarrow Val$ mapping class and field identifiers to values. A state $\sigma = \langle \mathcal{M}, \mathcal{H}\rangle$ is a pair of a memory $\mathcal{M}$ and a heap $\mathcal{H}$. An observation trace t is a (possibly empty) sequence of events $\beta \in Val$, and $t_1.t_2$ denotes trace concatenation.

We use judgments of the form $\alpha \vdash_\delta \langle S, \mathcal{M}, \mathcal{H}\rangle \Downarrow_t \mathcal{M}', \mathcal{H}'$ to denote that a configuration $\langle S, \mathcal{M}, \mathcal{H}\rangle$ evaluates to memory $\mathcal{M}'$ and heap $\mathcal{H}'$ in *execution mode* $\alpha \in \{N, E\}$ with *static mode* $\delta = Class \cup Var \rightarrow \{N, E\}$, and produces an observation trace t. We write $\alpha \vdash_\delta \langle S, \mathcal{M}, \mathcal{H}\rangle \Downarrow \mathcal{M}', \mathcal{H}'$ if the observation trace is empty and $\alpha \vdash_\delta \langle S, \mathcal{M}, \mathcal{H}\rangle \Downarrow_t \_$ to ignore the final state. We use modes to distinguish between the enclave environment (E) and the non-enclave environment (N). An execution mode α indicates the current execution environment of a configuration, while a static mode δ associates a class identifier or a variable with the execution environment it was assigned to, statically. Abusing notation, we write $\alpha \vdash_\delta \langle S; return(e), \mathcal{M}, \mathcal{H}\rangle \Downarrow \mathcal{M}, \mathcal{H} \triangleright v$ for commands that yield a value v, via the *return* command. To simplify the presentation, we assume that the sets of variable, field, and class identifiers of the enclave environment and non-enclave environment are disjoint. An initial configuration starts in the non-enclave execution mode and it executes a sequence of commands S_0 from an initial state $\langle \mathcal{M}_0, \mathcal{H}_0\rangle$, i.e. $N \vdash_\delta \langle S_0, \mathcal{M}_0, \mathcal{H}_0\rangle$.

The semantics of expressions, Figure A.2, is mostly standard. We use judgments of the form $\alpha \vdash_\delta \langle e_G, \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle$ to denote that an expression e_G evaluates to value v, memory $\mathcal{M}'$, and heap $\mathcal{H}'$ in state $\langle \mathcal{M}, \mathcal{H}\rangle$, execution mode α, static mode δ, and it emits the event β. Rule Method interacts with the execution semantics of commands to execute the method body, hence it can potentially alter the execution state. We use the auxiliary functions $getMethod : Class \times Method \rightarrow m$ to extract a method definition, and $fields : Class \rightarrow \wp(Field)$ extract the set of fields of class C. We write $\mathcal{M}[x \mapsto v]$ to denote a memory $\mathcal{M}$ with variable x assigned the value v. Similarly, $\mathcal{H}[(C, f) \mapsto v]$ denotes a heap $\mathcal{H}$ with field f of class C assigned the value v. The full semantics of expressions is reported in Figure A.7.

Rule Field Access evaluates expression e to a class identifier C, ensures that the execution mode and static mode are the same, and performs a lookup of its field f in the heap $\mathcal{H}$. It then returns the resulting value v as well as the memory and heap, which are unchanged.

Rule Method evaluates expression e to a class identifier C and uses the *getMethod* function to get the definition of its method m. Next, it checks that either the execution mode and static mode are the same ($\alpha = \delta(C)$) or that the execution mode is non-enclave (N), the static mode is enclave (E), and m is a gateway method ($\alpha = N \wedge \delta(C) = E \wedge \phi = G$). This condition ensures that only methods from the classes that have the same static mode as the current execution mode can be called, with the exception that the non-enclave execution mode can call the gateway methods of enclave classes. For the latter, our rules enforce the

Field Access

$$\frac{\alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle \qquad \alpha = \delta(C) \qquad v = \mathcal{H}(C, f)}{\alpha \vdash_\delta \langle e.f, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle}$$

Method

$$\frac{\begin{array}{c} \alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle \qquad method_\phi(\overline{p})\{S; return(e)\} = getMethod(C, m) \\ \Big(\big(\alpha = \delta(C)\big) \lor \big(\alpha = N \land \delta(C) = E \land \phi = G\big)\Big) \qquad \mathcal{M}^* = \mathcal{M}[p_i \mapsto \sigma(q_i)] \quad i = 1, ..., |\overline{p}| \\ \delta(C) \vdash_\delta \langle S; return(e), \mathcal{M}^*, \mathcal{H}\rangle \Downarrow \mathcal{M}^{*'}, \mathcal{H}' \rhd v \qquad \mathcal{M}' = \mathcal{M}^{*'} \setminus [p_i] \quad i = 1, ..., |\overline{p}| \\ (\alpha = \delta(C) \Rightarrow \beta = \epsilon) \qquad (\alpha = N \land \delta(C) = E \land \phi = G \Rightarrow \beta = v) \end{array}}{\alpha \vdash_\delta \langle e.m(\overline{q}), \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle}$$

Figure A.2: Excerpt of J_E expression semantics

Store

$$\frac{\alpha \vdash_\delta \langle e_G, \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle \qquad \alpha = \delta(C) \qquad \mathcal{H}'' = \mathcal{H}'[(C, f) \mapsto v]}{\alpha \vdash_\delta \langle C.f := e_G, \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \mathcal{M}', \mathcal{H}''}$$

Return

$$\frac{\alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle}{\alpha \vdash_\delta \langle return(e), \mathcal{M}, \mathcal{H}\rangle \Downarrow \mathcal{M}, \mathcal{H} \rhd v}$$

Figure A.3: Excerpt of J_E command semantics

copy semantics in a call-by-value fashion. The rule substitutes the actual parameters for the formal parameters and executes the method body, returning a value. Finally, if the method is a gateway call, the emitted event is the return value v, otherwise the empty event ϵ.

Figure A.3 provides an excerpt of evaluation rules for J_E commands. The full set of rules can be found in Figure A.9. Rule Store updates the field in the heap with the associated value and checks that $\alpha = \delta(C)$ to ensure that execution mode, and the static mode of class C are the same. The event that Store emits is equal to the event emitted during the evaluation of expression e_G. Rule Return evaluates the expression in the current state and returns the associated value.

Security Model

Security Policy

J_E adopts security labels to specify application-level policies as information flow policies. A security label ℓ is a tuple $\langle \ell_C, \ell_I \rangle$ of a confidentiality label ℓ_C and an integrity label ℓ_I. We use two levels, *Public* ($\mathbb{P}$) and *Secret* ($\mathbb{S}$), for confidentiality, and two levels, *Trusted* ($\mathbb{T}$) and *Untrusted* ($\mathbb{U}$), for integrity. Intuitively, for confidentiality, data from *Secret* sources should not flow to *Public* sinks unless it is explicitly declassified by the developer, and, for integrity, data from *Untrusted* sources should not flow to *Trusted* sinks unless it is explicitly endorsed by the developer. These requirements are reflected by the ordering relation between security labels, namely $\mathbb{P} \sqsubset \mathbb{S}$ and $\mathbb{T} \sqsubset \mathbb{U}$. The product lattice $\mathcal{L}$ lifts the constraints to an ordering relation over label pairs such that $\ell_1 \sqsubseteq \ell_2$ iff ${\ell_1}_C \sqsubseteq {\ell_2}_C$ and ${\ell_1}_I \sqsubseteq {\ell_2}_I$. The lattice $\mathcal{L}$ defines the *join* $\ell_1 \sqcup \ell_2$ and *meet* $\ell_1 \sqcap \ell_2$ operators to compute *least upper bound* and *greatest lower bound* of two labels, respectively. A security policy is then defined by an assignment of security labels to variables, classes and fields of an J_E program.

Attacker Model

We use the security labels to define the view of the memory and heap from the attacker's perspective. In our setting, the attacker is at level $\langle \mathbb{P}, \mathbb{U} \rangle$ and it can observe all of initial program state (i.e. the initial memory and the initial heap) that is labeled as public $\mathbb{P}$. This is because an attacker has full control of the non-enclave state and it is allowed to learn any public data of the enclave state. We assume that each variable, class, and field has an associated security label from the lattice $\mathcal{L}$ as defined by a security mapping $\Gamma : (Var \cup Class \cup Field) \rightarrow \mathcal{L}$. We then define indistinguishability over pairs of program states for a security mapping Γ and an attacker at security level $A = \langle \mathbb{P}, \mathbb{U} \rangle$.

Definition A.1 (State indistinguishability)

Two memories $\mathcal{M}_1$ and $\mathcal{M}_2$ are indistinguishable for the attacker A (written $\mathcal{M}_1 =_A \mathcal{M}_2$) iff $\forall x.\ \delta(x) = N$, $\mathcal{M}_1(x) = \mathcal{M}_2(x)$, and $\forall x.\ \delta(x) = E$ such that $\Gamma(x) = \langle \mathbb{P}, - \rangle$, $\mathcal{M}_1(x) = \mathcal{M}_2(x)$.
Two heaps $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable for the attacker A (written $\mathcal{H}_1 =_A \mathcal{H}_2$) iff $\forall C.\ \delta(C) = N, \forall f \in \mathit{fields}(C)$, $\mathcal{H}_1(C, f) = \mathcal{H}_2(C, f)$ and $\forall C.\ \delta(C) = E, \forall f \in \mathit{fields}(C)$ such that $\Gamma(C.f) = \langle \mathbb{P}, - \rangle$, $\mathcal{H}_1(C, f) = \mathcal{H}_2(C, f)$.
Two program states $\sigma_1 = \langle \mathcal{M}_1, \mathcal{H}_1 \rangle$ and $\sigma_2 = \langle \mathcal{M}_2, \mathcal{H}_2 \rangle$ are indistinguishable for the attacker A (written $\sigma_1 =_A \sigma_2$) iff $\mathcal{M}_1 =_A \mathcal{M}_2$ and $\mathcal{H}_1 =_A \mathcal{H}_2$.

Intuitively, two indistinguishable memories assign the same value to the public variables inside and outside of enclave. Similarly, two indistinguishable heaps assign the same value to all of the fields of *non-enclave classes* and the public fields of *enclave classes*.

As secret values are stored inside the enclave, the only way for an attacker to learn secret information is by observing the return values of gateway methods. In fact, the attacker observations are captured by our semantics in Figure A.3 via traces t and events β. Therefore, we can define indistinguishability for program executions by requiring that any two indistinguishable initial states produce the same observation traces. This implies that an attacker cannot discriminate the two initial states, thus it cannot learn secret information from the enclave environment.

Definition A.2 (Execution indistinguishability)

Let S be a J_E program and σ_1 and σ_2 be two initial states such that $\sigma_1 =_A \sigma_2$. Two executions are indistinguishable (written $N \vdash_\delta \langle S, \sigma_1 \rangle \approx_A N \vdash_\delta \langle S, \sigma_2 \rangle$) if $N \vdash_\delta \langle S, \sigma_1 \rangle \Downarrow_{t_1} \_$ and $N \vdash_\delta \langle S, \sigma_2 \rangle \Downarrow_{t_2} \_$ then $t_1 = t_2$.

In line with existing works [55], execution indistinguishability ignores information leaks that are due to program (non) termination. The definition ensures security w.r.t. a passive non-enclave attacker that observes only the results of gateway method calls. In particular, it rejects all programs that leak secret information to the non-enclave environment. As such condition can be restrictive for most practical scenarios (see Section A.3 for examples), developers resort to various forms of declassification operations to release secret information in a controlled manner. We refer to prior works for an overview of various dimensions of declassification [69]. In our setting, declassification can be dangerous as it can be abused by an active attacker to release secret information in a way that it was not intended by the developer [34]. Next, we define the attackers that are relevant in the context of enclave programs.

Passive and Active Attackers

We consider three types of attackers: the passive attacker (PA), the *havoc* active attacker (HAA) limited to modifying parameters passed to gateway calls, and the *havoc reordering* active attacker (HRAA) capable of controlling the order and frequency in which gateway methods are called. Both active attackers are reasonable in our context as the attacker fully controls the non-enclave environment. We use the PA attacker as a reference model to show that the HAA attacker and the HRAA attacker do not learn more secret information than the PA attacker.

The PA does not intervene in the execution of the program, it just observes observation traces (as in Figure A.3) to learn secret information from the enclave. The active attackers can influence the execution. The HAA attacker can modify the non-enclave state and hence the parameters passed to gateway methods, thus modifying the behavior of the program executing inside the enclave. This may change the behavior of the program in a way that leaks information about the enclave secrets, e.g. by abusing declassification operations that were intended in the context of a PA attacker. We illustrate the issue via an example inspired by Askarov and Myers [80].

Program A.3 contains a gateway method foo whose return value depends on the *time* parameter. The intention of the developer, who assumes a PA attacker, is to declassify $secretVal$ only after the *releaseTime* has elapsed. This is implemented by comparing the *time* in which method call was issued with the predefined *releaseTime* (Line 10). Yet, an HAA attacker can arbitrarily change the value of *time* and control the release of information via declassification. This example motivates the need for a security condition that rejects scenarios where an active attacker learns more secret information than a passive attacker.

To model the active capability of the HAA attacker, we introduce program holes [•] [34]. Holes represent program contexts where an HAA attacker can insert an untrusted code a

```
@Enclave
class FooClass {

    @Secret static int secretVal;
    static int releaseTime = 2025

    @Gateway
    public static int foo(int time) {
        int res = 0;
        if (time >= releaseTime)
            res = declassify(secretVal);
        else
            res = 0;

        return res;
    }
}
```

Program A.3: HAA attacker

to modify the program's state. Because the only way an attacker can affect the execution of enclave code is via gateways, we extend our program syntax by adding $[\bullet]; x := e.m(\overline{p})$ and $[\bullet]; C.f := e.m(\overline{p})$, and define the active attacks.

Definition A.3 (J_E program with holes)

A program with holes $S[\overrightarrow{\bullet}]$ is defined by extending the syntax in Figure A.1 as follows:

$$S[\overrightarrow{\bullet}] ::= \ldots \mid [\bullet]; x := e.m(\overline{p}) \mid [\bullet]; C.f := e.m(\overline{p})$$

where m is a gateway method.

An HAA attacker can execute any untrusted code a in a hole before method calls. However, this code can only contain variables, classes, and fields in the non-enclave environment which are by definition public and untrusted. We define the attacker's code as follows:

$$a ::= \mathit{skip} \;\mid\; a_1; a_2 \;\mid\; q := e \; (\mathit{where}\ q \in \overline{p}) \tag{A.1}$$

While an HAA attacker can inject arbitrary untrusted code in the non-enclave environment, we argue that the definition above captures the most powerful attack strategies of HAA attacker.

Lemma A.1

The attack code definition presented in A.1, captures the most powerful attack strategies available to an HAA attacker, who controls the non-enclave environment.

Proof. The proof of this lemma is presented in Appendix A.1. □

We write $S[\overrightarrow{a}]$ for a program under an HAA attack $\overrightarrow{a}$. In this setting, a passive attack can be modeled as $S[\overrightarrow{skip}]$.

An HRAA attacker controls both the code and data memory outside the enclave. Hence in addition to the HAA attack capability, an HRAA attacker can change the *order* and *frequency* of gateway method calls issued from the non-enclave environment. This can cause information leaks as the (attacker-controlled) order and frequency of gateway calls may influence the values returned to the non-enclave environment.

```
@Enclave
class FooClass {

    @Secret static int secret1, secret2;
    static boolean releaseTrigger = false;

    @Gateway
    public static void bar() {
        releaseTrigger = true;
    }
    @Gateway
    public static int foo() {
        int res = 0;
        if (releaseTrigger) {
            releaseTrigger = !releaseTrigger;
            res = declassify(secret1); }
        else {
            releaseTrigger = !releaseTrigger;
            res = declassify(secret2); }

        return res;
    }
}
```

Program A.4: HRAA attacker

Program A.4 illustrates the problem. Consider the non-enclave program `FooClass.bar(); FooClass.foo()`. The intended order of issuing gateway methods is `bar();foo()`, hence this program is secure with respect to an HAA attacker. `secret1` is always going to be declassified, and the resulting trace depends on its value. However, an HRAA attacker that controls the code memory outside the enclave can change the order of gateway calls to `foo();bar()` and learn the declassified value of `secret2`.

A similar argument applies to calling gateway methods multiple times. For instance, if a gateway was intended to be called only once, calling it more than once might leak sensitive information. We revisit Program A.4 to illustrate the problem. Consider the non-enclave program `FooClass.foo()` revealing the value of `secret1`. An HRAA attacker can instead call `FooClass.foo();FooClass.foo()` and learn the declassified value of `secret2`.

Since the HRAA attacker has full control over the non-enclave code and memory, and the secrets reside only inside the enclave, we model them as sequences of gateways calls.

Definition A.4 (Program under HRAA control)

We define the program under HRAA control as a sequence of gateway calls:

$$S'[\overrightarrow{\bullet}] ::= S'_1[\overrightarrow{\bullet}]; S'_2[\overrightarrow{\bullet}] \mid [\bullet]; x := C.m(\overline{p})$$

where m is a gateway method defined in $S[\overrightarrow{\bullet}]$.

In this model, the attack definition of A.1 will remain unchanged, indicating that HRAA attacker subsumes the power of HAA attacker.

Lemma A.2

The attacker code defined in A.1, captures the most powerful attack strategies available to the HRAA attacker.

Proof. The proof is similar to Lemma A.1 and is presented in Appendix A.1. □

Security Condition

In our setting, programs use declassification to release secret information in a controlled manner. Intuitively, a program is secure if an active attacker cannot learn more secret information than a passive attacker. Drawing on the idea of robust declassification [34], we present robustness, a security condition that formalizes this intuition.

Definition A.5 (Robustness under HAA)

Program $S[\overrightarrow{\bullet}]$ is robust w.r.t an HAA attacker A if for all $\sigma_1, \sigma_2, \vec{a}_1, \vec{a}_2$

$$N \vdash_\delta \langle S[\vec{a}_1], \sigma_1 \rangle \approx_A N \vdash_\delta \langle S[\vec{a}_1], \sigma_2 \rangle \Rightarrow N \vdash_\delta \langle S[\vec{a}_2], \sigma_1 \rangle \approx_A N \vdash_\delta \langle S[\vec{a}_2], \sigma_2 \rangle$$

Robustness holds whenever for an attack vector $\vec{a}$ and two indistinguishable initial states σ_1, σ_2, if the program $S[\vec{a}_1]$ satisfies execution indistinguishability (Definition A.2), then for another attack vector $\vec{a}_2$, the program $S[\vec{a}_2]$ also satisfies execution indistinguishability. In other words, the attacker observations of $S[\vec{a}_2]$'s observation traces do not reveal any secrets apart from what the attacker already knows by the observation traces of the program $S[\vec{a}_1]$. The PA attacker is captured by executions of the program $S[\overrightarrow{skip}]$.

We extend robustness to capture HRAA attackers.

Definition A.6 (Robustness under HRAA)

Program $S[\overrightarrow{\bullet}]$ is robust w.r.t an HRAA attacker A if for all $\sigma_1, \sigma_2, \vec{a}_1, \vec{a}_2$ and for all $S'[\overrightarrow{\bullet}]$:

$$N \vdash_\delta \langle S[\vec{a}_1], \sigma_1 \rangle \approx_A N \vdash_\delta \langle S[\vec{a}_1], \sigma_2 \rangle \Rightarrow N \vdash_\delta \langle S'[\vec{a}_2], \sigma_1 \rangle \approx_A N \vdash_\delta \langle S'[\vec{a}_2], \sigma_2 \rangle$$

This definition ensures that the HRAA attacker does not learn more information by changing the order and frequency of gateway calls. Observe that if $S'[\overrightarrow{a'}]$ triggers an execution that was not possible $S[\overrightarrow{a}]$, and that execution's return value depended on some declassified secret, the active power may enable the HRAA attacker to learn information that they would not have learned originally. In Program A.3, the program $S[\overrightarrow{a}] ::= [a_1];$ `FooClass.bar`$();[a_2];$ `FooClass.foo`$()$ is not robust under HRAA because the program $S'[\overrightarrow{a'}] ::= [a'_1];$ `FooClass.foo`$();[a'_2];$ `FooClass.bar`$()$ enables the HRAA attacker to reveal the declassified value of `secret2`. A similar argument applies to calling gateway methods that were defined in $S[\overrightarrow{\bullet}]$ but were never called. As expected, this definition is stronger than Definition A.5.

Delayed Declassification

While our security condition extends the definition of robust declassification [34] to the setting of realistic enclave attackers, there are some key differences pertaining to traces and attacker observations. Robust declassification considers every assignment to public variables as immediately visible to the attacker, because it defines the observations as projection over the public part of the memory. This definition does not reflect the enclave attacker model, because TEEs encrypt the entire enclave memory, so even if a public variable is modified inside the enclave, it is not visible to the attacker unless it is written to the non-enclave memory. This motivates our use of a trace-based observation model as generated by the return values of gateway method calls.

```
@Enclave
class FooClass {

    @Secret static int secretVal;

    @Gateway
    public static int foo(int input) {
        int x = declassify(secretVal);
        int l = 0;
        if (input > 0)
            l = x;
        else
            l = 7

        return l;
    }
}
```

Program A.5: Delayed declassification

This model poses additional challenges with handling of declassification policies. For example, Program A.5 satisfies robust declassification because the attacker *input* neither affects the decision to declassify nor the declassified value itself. This is achieved by making the value in variable x observable to the attacker immediately. However, in our model, the declassified value in x will not be visible to the attacker until it is returned by the gateway method. Because the attacker controls the input and therefore the assignment in line 11, this results in controlling the decision to declassify the secret value. Hence, the program is not robust. In fact, for attack vectors $a ::= \mathit{input} := 0$ and $a' ::= \mathit{input} := 1$, our definition of robustness will correctly reject the program. We dub this concept *delayed declassification*. Observe that delayed declassification is orthogonal to the well-known *Where* and *What* dimensions of declassification [69] and it appears as result of the trace-based observation model.

Security Type System

This section presents a security type system to enforce robustness. In line with the enclave attacker model, the program from the non-enclave environment is public and untrusted, while program from the enclave environment is trusted and its input data can be labeled by the developer as either public or secret. We label secret fields with $< \mathbb{S}, \mathbb{T} >$as they contain sensitive information and are protected by the enclave. The security label of the arguments of a gateway method is defaulted to $< \mathbb{P}, \mathbb{U} >$and while the return parameter is labeled as $< \mathbb{P}, \mathbb{T} >$or $< \mathbb{P}, \mathbb{U} >$.

The goal of our type system is to ensure robustness against the active attackers. To achieve this, the type system checks that the decision to declassify a secret, or to return it, is not influenced by untrusted non-enclave data, thus ensuring that the attacker cannot control the decision to release secret data. The security type system enforces the *Where* dimension of declassification for a PA attacker [69] and it has been proved sound with respect to the security condition of *gradual release* [51]. We extend the type system to additionally account for active attacks in our setting and prove it sound for robustness. This has the advantage of reusing existing verification efforts via security type systems, which assume a PA attacker, and verifying only on the effect of an active attacker whenever these programs have been verified for the PA attacker. In our setting, this may happen whenever a developer partitions an existing (secure) program to execute with enclaves.

Our security type system uses a typing environment $\Gamma : (\mathit{Var} \cup \mathit{Class} \cup \mathit{Field}) \rightarrow \mathcal{L}$ mapping variables, classes, and fields to security labels from the security lattice $\mathcal{L}$. We also use another environment $\Pi : (\mathit{Var} \cup \mathit{Class} \cup \mathit{Field}) \rightarrow \mathcal{B}$ that maps variables, fields, and classes to boolean flags. A flag $d \in \mathcal{B}$ can be true (T) or false (F), and is used to track the propagation of declassified values. Initially, every variable and field has the flag initialized to false and the flag is set to true whenever they store a value that may be affected by declassification. We define the ordering relation $F \sqsubset T$ on flags, which will be useful in the sub-typing and method rules.

The label of every variable, field, and class in our setting, is a tuple consisting of a security label and a flag (ℓ, d). We use indexes ℓ and d to access elements of this tuple. e.g. pc_ℓ will show the security label of pc. Methods are typed in isolation using type signatures of the form $\left(\Gamma^-, \Pi^- \xrightarrow[pc']{} \Gamma^+, \Pi^+\right)_{(\ell,d)}$ which require a environments Γ^- and Π^- before the method is invoked, environments Γ^+ and Π^+ after the method invocation, the label of its return value (ℓ, d), and the program counter label pc' capturing the lower bound on method's side-effects [126]. The typing judgments for expressions are of the form $\Gamma, \Pi \vdash_\delta e_G : \tau$, meaning that in mode δ, and environments Γ and Π, an expression e_G

has the type τ. If the expression e_G is a method, then τ is a method type, otherwise τ is a type (ℓ, d) representing the security label of e. Similarly, the typing judgments for commands have the form $pc, \Gamma, \Pi \vdash_\delta S : \Gamma', \Pi'$ where Γ and Π are the environments before, and Γ' and Π' are the environments after the execution of command S, δ is the static, and pc is the program counter label used to prevent implicit flows. The judgment for the return command is of the form $pc, \Gamma, \Pi \vdash_\delta \texttt{return}(e) : \Gamma, \Pi \rhd (\ell, d)$ to capture the security label ℓ and the flag d of the returned value.

Figure A.4 depicts an excerpt of the typing rules for expressions. The full list of rules is reported in Figure A.8. The only non-standard rule here is T-Method. Rule T-Method checks that method's body $S; \texttt{return}(e)$ is well-typed under pc', Γ^-, Γ^+, Π^-, and Π^+ and it returns the label (ℓ, d) of the result.

$$\textsc{T-Int}\quad \frac{}{\Gamma, \Pi \vdash_\delta n : (\langle \mathbb{P}, \mathbb{T} \rangle, F)} \qquad \textsc{T-Op}\quad \frac{\Gamma, \Pi \vdash_\delta e_1 : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta e_2 : (\ell_2, d_2)}{\Gamma, \Pi \vdash_\delta e_1 \oplus e_2 : (\ell_1 \sqcup \ell_2, d_1 \vee d_2)}$$

$$\textsc{T-Method}\quad \frac{\text{method}_\phi(\overline{p})\{S; \texttt{return}(e)\} = \text{getMethod}(C, m) \qquad pc', \Gamma^-, \Pi^- \vdash_\delta S; \texttt{return}(e) : \Gamma^+, \Pi^+ \rhd (\ell, d)}{\Gamma, \Pi \vdash_\delta C.m(\overline{p}) : \left(\Gamma^-, \Pi^- \underset{pc'}{\longrightarrow} \Gamma^+, \Pi^+\right)_{(\ell, d)}}$$

Figure A.4: Excerpt of typing rules for expressions

Figure A.5 presents a few interesting rules for commands; we refer to Figure A.10 for the full list. In our type system, variables are flow-sensitive (see rule T-Assign in Figure A.10), while fields are flow-insensitive, i.e. their security label is defined via annotations. In fact, rule T-Store ensures that the join of the security labels of pc and expression e is at least as restrictive as the label of field $C.f$. Moreover, the rule ensures that if the security label of $\ell_1 \sqcup \ell_2 \sqcup pc_\ell$ is secret $\langle \mathbb{S}, - \rangle$, the store is defined in the enclave, and it ensures that if the flag of e is true, this command can only be executed in a trusted context. This is to prevent untrusted input from controlling the propagation and release of declassified values. Finally, even though $C.f$'s security labels are flow-insensitive, its flag is not, and it is updated to the disjunction of the flags of e and pc. This rule also updates the flag of class C, so if a class has a field affected by a declassified value, the whole class is going to be flagged true. The rule T-Store is only for side-effect free expressions (i.e. e). There is also $e.m(\overline{p})$ that combined with Store can act as a method call or a *gateway* method call. Rule T-Store-Call and T-Store-Gateway-Call type check method calls and gateway method calls, respectively. Similarly, type checking assignments is broken into three rules, T-Assign, T-Assign-Call, and T-Assign-Gateway-Call.

Rule T-Declassify ensures that only trusted data is allowed to be declassified $\ell \sqsubseteq \langle \mathbb{S}, \mathbb{T} \rangle$ and declassification can only happen in public and trusted context $pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle$. This prevents attacker-controlled untrusted data from influencing the decision to declassify secret information. The security label of variable x will be $\langle \mathbb{P}, \mathbb{T} \rangle$ and its flag will be true T.

Rule T-Return sets the security label of the returned value to the join of the security labels of program context and the expression. This is to prevent the implicit flows that may happen when returning in a secret context. Additionally, if the expression e's flag is true, return can only be executed in a trusted context. This is to prevent attacker from affecting the decision to return a declassified value.

Rule T-Store-Gateway-Call handles the type checking of gateway methods call from a store. The security labels of gateway parameters are explicitly defined in Γ^- and must have the $\langle \mathbb{P}, \mathbb{U} \rangle$ security label ($\forall p \in \overline{p}.\ \Gamma^-(p) = \langle \mathbb{P}, \mathbb{U} \rangle$), and their flag (defined in Π^-) should be false $\forall p \in \overline{p}.\Pi^-(p) = F$. The rule ensures that the gateway can only return public values ($\ell_2 \sqsubseteq \langle \mathbb{P}, \mathbb{U} \rangle$), the labels of actual parameters are less restrictive than the predefined labels of formal parameters $\Gamma(q_i) \sqsubseteq \Gamma^-(p_i)$, and the method's typing environment is satisfied ($\forall y \in \mathit{dom}(\Gamma^-).\Gamma(y) \sqsubseteq \Gamma^-(y)$). Additionally, the typing environment after type checking the method body should respect the method's predefined post typing environment ($\forall y \in \mathit{dom}(\Gamma^+).\Gamma^+(y) \sqsubseteq \Gamma_{out}(y)$), while ensuring that the type of identifiers other than those used by the method remains unchanged ($\forall y \in \mathit{dom}(\Gamma) \setminus \mathit{dom}(\Gamma^+).\Gamma(y) = \Gamma_{out}(y)$). Similar conditions apply to Π environment. At last, the rule updates the flag of field $C.f$ to false, so even a declassified value, after returning from a gateway method has a false flag.

T-Store

$$\dfrac{\begin{array}{c}\Gamma, \Pi \vdash_\delta C.f : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta e : (\ell_2, d_2) \qquad \ell_2 \sqcup pc_\ell \sqsubseteq \ell_1 \\ \ell_1 \sqcup \ell_2 \sqcup pc_\ell = \langle \mathbb{S}, - \rangle \Rightarrow \delta(C) = E \qquad pc_d \vee d_2 = T \Rightarrow pc_\ell \sqcup \ell_1 = \langle -, \mathbb{T} \rangle \qquad d' = d_2 \vee pc_d\end{array}}{pc, \Gamma, \Pi \vdash_\delta C.f := e : \Gamma, \Pi\Big[C.f \mapsto d', C \mapsto \Pi(C) \vee d'\Big]}$$

T-Declassify

$$\dfrac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad \ell \sqsubseteq \langle \mathbb{S}, \mathbb{T} \rangle \qquad pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle \qquad \delta(x) = E}{pc, \Gamma, \Pi \vdash_\delta x := \texttt{declassify}(e) : \Gamma\Big[x \mapsto \ell \sqcap \langle \mathbb{P}, \mathbb{T} \rangle\Big], \Pi\Big[x \mapsto T\Big]}$$

T-Return

$$\dfrac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad d = T \Rightarrow pc_\ell = \langle -, \mathbb{T} \rangle}{pc, \Gamma, \Pi \vdash_\delta \texttt{return}(e) : \Gamma, \Pi \triangleright (pc_\ell \sqcup \ell, d)}$$

T-Store-Gateway-Call

$$\dfrac{\begin{array}{c}\Gamma, \Pi \vdash_\delta C.f : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta C'.m(\overline{p}) : \left(\Gamma^-, \Pi^- \ \frac{}{pc'}\ \Gamma^+, \Pi^+\right)_{(\ell_2, d_2)} \\ \delta(C) = N \qquad \delta(C') = E \qquad \ell_2 \sqsubseteq \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Gamma^-(p) = \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Pi^-(p) = F \\ \ell_2 \sqcup pc_\ell \sqsubseteq \ell_1 \qquad \Gamma(q_i) \sqsubseteq \Gamma^-(p_i) \quad i = 1...|\overline{p}| \qquad \Pi(q_i) = \Pi^-(p_i) \quad i = 1...|\overline{p}| \\ \forall y \in \mathrm{dom}(\Gamma^-).\Gamma(y) \sqsubseteq \Gamma^-(y) \qquad \forall y \in \mathrm{dom}(\Gamma^+).\Gamma^+(y) \sqsubseteq \Gamma_{out}(y) \\ \forall y \in (\mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^+).\Gamma(y) = \Gamma_{out}(y) \qquad \forall y \in \mathrm{dom}(\Pi^-).\Pi(y) \sqsubseteq \Pi^-(y) \\ \forall y \in \mathrm{dom}(\Pi^+).\Pi^+(y) \sqsubseteq \Pi_{out}(y) \qquad \forall y \in (\mathrm{dom}(\Pi) \setminus \mathrm{dom}(\Pi^+).\Pi(y) = \Pi_{out}(y)\end{array}}{pc, \Gamma, \Pi \vdash_\delta C.f := C'.m(\overline{q}) : \Gamma_{out}, \Pi_{out}\Big[C.f \mapsto F\Big]}$$

Figure A.5: Excerpt of typing rules for commands

To illustrate our type system, we revisit the example of Program A.3. The type system rejects this program because the declassify operator can only be used in a trusted program context *pc*. Rules T-Op and T-If-Else set the *pc* security label to the join of the security labels of `time` and `releaseTime`. Since `time` has security label $\langle \mathbb{P}, \mathbb{U} \rangle$ (as it comes from outside of the enclave), and `releaseTime` has security label $\langle \mathbb{P}, \mathbb{T} \rangle$ (as it comes from the enclave), we have that $pc_\ell = \langle \mathbb{P}, \mathbb{U} \rangle \sqcup \langle \mathbb{P}, \mathbb{T} \rangle = \langle \mathbb{P}, \mathbb{U} \rangle$. Next, rule T-DECLASSIFY does not allow the declassification since $pc_\ell \not\sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle$.

We prove that our security type system enforces robustness for the HAA attacker.

Theorem A.1

If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HAA.*

We now show how it can be extended with minimal changes to enforce robustness for HRAA attackers. The additional power of HRAA attacks comes from the ability to control the order and frequency of gateway calls. Whenever a gateway call modifies trusted identifiers inside the enclave, this can be used to influence declassification operations that are performed by another method. This can be prevented by computing the set of *shared identifiers* Σ containing all global variables and fields that are *assigned* to in at least one method and *accessed* in at least one method. (i.e. Σ contains all global variables and fields that are used and modified in one or more gateways methods) We compute Σ in a preprocessing step up to reaching a fixed point. We can then use the typing rules of Figures A.4 and A.5 under the constraint that the initial environment considers the integrity label of every variable and field in Σ as *untrusted.* i.e. $\forall\ x \in \Sigma.\ \Gamma_0[x \mapsto \langle \Gamma_0(x)_C, \mathbb{U} \rangle]$ and $\forall\ C.f \in \Sigma.\ \Gamma_0[C.f \mapsto \langle \Gamma_0(C.f)_C, \mathbb{U} \rangle]$.

The intuition is that by considering shared identifiers as untrusted, we enable the security type system to reject programs that use these identifiers to declassify information (explicitly or implicitly) and return it via a gateway method. For example, Program A.4 is rejected by our type system. After the preprocessing phase, $\Sigma = \{$`releaseTrigger`$\}$ since `releaseTrigger` is assigned to in one method call and accessed in another. By assigning the security label $\langle \mathbb{P}, \mathbb{U} \rangle$ to `releaseTrigger`, rule T-DECLASSIFY (Line 16) will fail since declassification is not allowed in an untrusted context.

Recall from last section that if different program executions call different gateways and those gateways return *declassified values*, the HRAA attacker can learn more by calling the gateways that were not called in the original program. In order to prevent this, we add another step to the type checking process to ensure that all gateways which declassify secret values are called in all possible executions of the (non-enclave) program.

This process is performed in several steps:

1. Identify the set of all the *gateway* methods in $S[\overrightarrow{\bullet}]$ denoted by G^D such that their return value's flag is true (i.e. $\Gamma, \Pi \vdash_\delta C.m(\overline{p}) : (-)_{(-,T)}$).
2. Enumerate the paths of the program and extract the set of gateway calls along those paths. This is achieved by a depth-first traversal of the program's graph. We use $pathsGW(S[\overrightarrow{\bullet}])$ to denote the set of all possible paths of $S[\overrightarrow{\bullet}]$, and $pathGW_i$ for the set of gateway calls in path i.
3. Check that G^D is a subset of all of the possible paths of $S[\overrightarrow{\bullet}]$. In other words:

$$G^D \subseteq pathGW_i \quad \forall i \in pathsGW(S[\overrightarrow{\bullet}])$$

This process is performed after calculating Σ and type checking methods in isolation, but before type checking the program itself. If the above process fails, we reject the program as not robust against HRAA attacker. This requirement reflects the power of the HRAA attacker outside the enclave, rejecting programs that do not call all declassifying gateways. To improve permissiveness and security, such programs can be moved to the enclave and exposed to the non-enclave environment as a single gateway method.

We remark that our enforcement accepts programs with noninterfering runs that do not always declassify information. Program A.6 presents a program with both interfering and noninterrfering runs. The program satisfies robustness (Definition A.6) and is also accepted by the type system. In fact, if variable `trustedLow` (Line 4, a public and trusted variable) is set to true, only noninterfering computations will be executed and an HRAA attacker program such as `[choice=v];ComputeArray.compute(choice);` will observe the result `"Done with computation" + choice`. Otherwise, if `trustedLow` is set to false, an interfering run will be executed and the attacker will observe the declassified average value.

```java
 1  @Enclave
 2  class ComputeArray {
 3
 4      static boolean trustedLow;
 5      @Secret static int[] array;
 6
 7      @Gateway
 8      public static String compute(int choice) {
 9          int avg = computeAvg(array);
10          String res = 0;
11          if (trustedLow) {
12              if (choice == 1) {
13                  computation1(array); //noninterfering
14              } else if (choice == 2) {
15                  computation2(array); //noninterfering
16              } else if (choice == 3) {
17                  computation3(array); //noninterfering
18              }
19              res = "Done with computation" + choice;
20              }
21          }
22          else {
23              res = String.valueOf(declassify(avg));
24          }
25          return res;
26      }
27  }
```

Program A.6: Noninterfering runs under HRAA

This example demonstrates that even though our type system requires calling all of the declassifying gateways along all program's executions, it does not mean that all of the runs of the program are necessarily interfering.

We prove soundness of the security type system with respect to the HRAA attacker.

Theorem A.2

If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *with regard to* Σ *and* G^D*, then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HRAA.*

We refer to Appendix A.1 for the proofs of theorems.

Use case for the HRAA attacker Our enforcement mechanism for the HRAA attacker requires a program to call all gateways that declassify secret information in any execution of a program. While this condition may seem restrictive, it is necessary in order to ensure that an attacker as powerful as HRAA cannot manipulate the program to tamper with declassification in unintended ways.

We identify the setting of IoT app platforms as a promising use case for enforcing security against the HRAA attacker [157]. IoT apps allow users to run simple trigger-action apps in cloud-based IoT platforms to seamlessly connect their IoT services and devices. Upon the triggering of an event, e.g. “EZVIZ camera senses motion at home”, the app executes code to perform an action, e.g. “Send an email with the camera stream”. Currently, the users have to trust the cloud provider with the sensitive information of their services and devices to run apps on their behalf. TEEs can help executing user apps securely in an untrusted IoT cloud platform and protect against the HRAA attacker. Specifically, the user can leverage enclaves to securely connect their services and smart devices, e.g. Email and EZVIZ camera, via authentication tokens (using code patterns similar to Program A.10) and use these tokens to execute trigger-action automations via gateway methods that simply transfer data between services and devices, and do not declassify sensitive information. In this setting, only the authentication gateway declassifies sensitive information, thus making these programs amenable to verification by our type system.

A.5 Endorsement and Nonmalleable Attacks

Endorsement

We extend our security framework to accommodate explicit endorsement of untrusted information coming from the non-enclave environment via gateway calls. This enables a developer to mark untrusted expressions as trusted, explicitly, to indicate that the security policy should be insensitive to their value. Following the approach of Askarov and Myers [80], we extend the syntax and semantics of J_E with command $x := \texttt{endorse}_\eta(e)$. Each endorsement command has a unique label η, and produces an endorsement event $\mathit{endorse}(\eta, v)$, which records the label η along with the endorsed value v.

$$\textsc{Endorse}\frac{E \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle \qquad \delta(x) = E \qquad \mathcal{M}' = \mathcal{M}[x \mapsto v]}{E \vdash_\delta \langle x := \texttt{endorse}_\eta(e), \mathcal{M}, \mathcal{H}\rangle \Downarrow_{\mathit{endorse}(\eta, v)} \mathcal{M}', \mathcal{H}}$$

We define *irrelevant* attacks as the set of attacks that are endorsed, and therefore can be excluded from the set of attacks used in Definition A.5. Using this concept, we argue that Definition A.5 should only hold for relevant attacks. Given a program $S[\vec{\bullet}]$, starting state σ, and attacker vector $\vec{a}$ which produces trace t (i.e. $\langle S[\vec{a}], \sigma\rangle \Downarrow_t$), relevant attacks, denoted by $\Omega(S[\vec{a}], \sigma)$, are the attacks that lead to the same sequence of endorsement events as in t.

We define relevant attacks by using irrelevant traces, which given trace t, are the set of all traces that agree with t on some prefix of *endorsement events* until they necessarily disagree on some endorsement. Formally:

Definition A.7 (Irrelevant traces)

Given a trace t, where endorsements are marked as $endorse(\eta_j, v_j)$, define a set of irrelevant traces based on the number of endorsements in t

$$\psi_i(t) = \{t' \mid t' = k.endorse(\eta_i, v_i').k'\}$$

where k is a prefix of t' with $i-1$ events all of which agree with **endorse events** in t, and $v_i \neq v_i'$. We define $\psi(t) \triangleq \bigcup_i \psi_i(t)$ as a set of irrelevant traces w.r.t. trace t.

Now, we can define the relevant attacks as the set of attacks that *do not* lead to irrelevant traces.

Definition A.8 (Relevant attacks)

Given a program $S[\vec{\bullet}]$, starting state σ, and attacker vector $\vec{a}$ such that $\langle S[\vec{a}], \sigma\rangle \Downarrow_t$, relevant attacks $\Omega(S[\vec{a}], \sigma)$ are defined as:

$$\Omega(S[\vec{a}], \sigma) = \{a' \mid \langle S[\vec{a'}], \sigma\rangle \Downarrow_{t'} \wedge t' \notin \psi(t)\}$$

Using the definition relevant attacks, we can redefine the robustness property. This new security condition accounts for the fact that the active effect on endorsed expressions does not matter.

Definition A.9 (Robustness with endorsement)

Program $S[\vec{\bullet}]$ is robust w.r.t an HAA attacker A if for all $\sigma_1, \sigma_2, \vec{a}_1$ and for all $\vec{a}_2 \in \Omega(S[\vec{a}_1], \sigma_1)$

$$N \vdash_\delta \langle S[\vec{a}_1], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\vec{a}_1], \sigma_2\rangle \Rightarrow N \vdash_\delta \langle S[\vec{a}_2], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\vec{a}_2], \sigma_2\rangle$$

This definition is similar to the original robustness property, except that instead of ensuring indistinguishability for all possible attacks, it only ensures indistinguishability for the relevant attacks, effectively ignoring the influence of irrelevant attacks.

Figure A.6 presents the typing rule for endorsement. Similar to the T-Declassify, T-Endorse ensures that $endorse_\eta(e)$ can only be used inside the enclave environment, and endorsement can only occur in a public and trusted context $pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T}\rangle$. This is to

$$\text{T-Endorse}\ \frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle \qquad \delta(x) = E}{pc, \Gamma, \Pi \vdash_\delta x := \texttt{endorse}_\eta(e) : \Gamma\left[x \mapsto \ell \sqcap \langle \mathbb{S}, \mathbb{T} \rangle\right], \Pi\left[x \mapsto d\right]}$$

Figure A.6: Typing rule for endorse command

prevent attacker-controlled untrusted data to influence the decision to endorse untrusted information.

Now, we present type soundness to prove that any well-typed program under the extended type system satisfies the new robustness property of Definition A.9.

Theorem A.3

If $pc, \Gamma, \Pi \vdash_\delta S[\vec{\bullet}] : \Gamma', \Pi'$ *then* $S[\vec{\bullet}]$ *satisfies robustness with endorsement.*

We can use a similar approach to model endorsement of HRAA attacker. However, we have to modify the definition of relevant attacks to account for the difference between the order and frequency of endorsement events in $S'[\vec{\bullet}]$ and $S[\vec{\bullet}]$.

In the new definition of relevant attacks, we use the unique endorsement label η to ensure that the value of an endorsement used in $S[\vec{\bullet}]$ is equal to the value of that same endorsement in $S'[\vec{\bullet}]$ independently of the order and frequency of that endorsement's event. We lift the definition of irrelevant traces and relevant attacks to this new setting.

Definition A.10 (HRAA irrelevant traces)

Given a trace t, where endorsements are marked as $endorse(\eta_j, v_j)$, define the set of irrelevant traces as:

$$\psi_R(t) = \{t' \mid \exists\ endorse(\eta_j, v_j) \in t, \exists\ endorse(\eta_i, v_i) \in t'.\ \eta_j = \eta_i\ and\ v_j \neq v_i\}$$

Observe that we need Definition A.10 because, if we used Definition A.7, there might be cases where there is an irrelevant trace t', but there exist no k with $i-1$ events such that all of its endorse events agree with t, thus, t' cannot be marked as irrelevant.

Definition A.11 (HRAA relevant attacks)

Given programs $S[\vec{\bullet}]$ and $S'[\vec{\bullet}]$, starting state σ, and attacker vector $\vec{a}$ such that $\langle S[\vec{a}], \sigma\rangle \Downarrow_t$, the set of relevant attacks w.r.t HRAA attacker $\Omega_R(S[\vec{a}], S'[\vec{\bullet}], \sigma)$ are defined as:

$$\Omega_R(S[\vec{a}], S'[\vec{\bullet}], \sigma) = \{a' \mid \langle S'[\vec{a'}], \sigma\rangle \Downarrow_{t'} \wedge t' \notin \psi(t)\}$$

Now, using this new definition of relevant attacks, we can redefine the robustness property w.r.t HRAA attacker.

Definition A.12 (Robustness under HRAA with endorsement)

Program $S[\vec{\bullet}]$ is robust w.r.t an HAA attacker A if for all $\sigma_1, \sigma_2, \vec{a}_1$, and for all $S'[\vec{\bullet}]$ such that $\forall \vec{a}_2 \in \Omega_R(S[\vec{a_1}], S'[\vec{\bullet}], \sigma_1)$

$$N \vdash_\delta \langle S[\vec{a}_1], \sigma_1 \rangle \approx_A N \vdash_\delta \langle S[\vec{a}_1], \sigma_2 \rangle \Rightarrow N \vdash_\delta \langle S'[\vec{a}_2], \sigma_1 \rangle \approx_A N \vdash_\delta \langle S'[\vec{a}_2], \sigma_2 \rangle$$

This definition ensures that in a robust program w.r.t HRAA attacker, as long as the endorsed values are equal, the result of execution for all possible orders and frequencies of gateway calls is indistinguishable.

We can augment the security type system of HRAA attacker with rule of Figure A.6 to enforce robustness. The following theorem proves that any well-typed program under the extended type system w.r.t HRAA attacker satisfies the robustness property of Definition A.12.

Theorem A.4

If $pc, \Gamma, \Pi \vdash_\delta S[\vec{\bullet}] : \Gamma', \Pi'$ *with regard to* Σ *and* G^D, *then* $S[\vec{\bullet}]$ *satisfies robustness under HRAA with endorsement.*

We refer the readers to Appendix A.1 for the proofs of Theorems A.3 and A.4.

Nonmalleable Attacks

Transparent endorsement [134] was introduced as a dual to robust declassification to prevent attacks arising from trusted agents endorsing information that their provider could not have seen. It is a common practice in TEE to have encrypted data passed to the enclave from outside, if these encrypted data are labeled *secret*, then the type system will reject the program and prevent the malleable attacks. Otherwise, if these encrypted ciphertexts are labeled *public* even though they contain secret plaintexts, then it is possible to receive an input which its provider cannot read, thus opening up the system to malleable attacks. While from a technical perspective transparent endorsement can be accommodated into our framework along the lines of Cecchetti et al. [134], we postpone it to future work.

A.6 Implementation

In this section, we describe the implementation of J_E for Intel SGX and discuss the gap with our formal model.

The J_E runtime operates with two separate execution environments, namely the non-enclave environment and enclave environment, which correspond to two separate Java Virtual Machines. The communication between the two environments is achieved using Java RMI [203].

The J_E compilation process involves multiple steps. A J_E program is first partitioned, then translated to Jif to check security, and finally transformed to use RMI communication. A detailed description of the code transformations implemented by the J_E compiler with a complete step-by-step example is in the technical report [185].

In addition to the abstractions supported in the formal model, i.e. static classes and static fields, the implementation supports additional Java features, including objects and generics. We convert J_E programs that include objects and generics into equivalent Jif programs. Because the implementation leverages Jif for label propagation, J_E security analysis is flow-insensitive, in contrast to the security type system, where the analysis is flow-sensitive on program variables. In the implementation, we do not assign any default Jif label to the local variables, hence, local variables get the security label of the expression they are initialized with. We remark that this decision is sound with respect to the language subset considered in our formal model. In fact, if a (declassification-free) program is deemed secure by Jif's flow-insensitive analysis, then it is also secure with respect to our flow-sensitive analysis. However, soundness may come at the expense of additional manual annotations due to the flow insensitivity of Jif's analysis.

Another difference with our formal model is the enforcement of delayed declassification. We not do implement the propagation of the declassify flag (d) used to track delayed declassification in section A.4. Instead, we require that every `declassify` operator is used only as a parameter of a `return` statement. Since any declassifications are performed within the scope of the return statement, it simply disallows the programmer to write code subject to delayed declassification leaks. We implement this check during the static analysis.

We use the SGX-LKL [204] framework to run the JAR corresponding to the enclave partition. SGX-LKL uses Linux Kernel Library [77] to handle system calls from the application within the enclave. In SGX-LKL, the application JAR, a JVM, and the necessary LKL binaries are compiled to a single image file. SGX-LKL loads the image and runs it inside the enclave.

A.7 Evaluation

In this section, we present the evaluation of J_E. We implemented case studies to demonstrate how the design of J_E can address the security requirements of distributed applications. The goal of the case studies is to exhibit the features of the language. Additional case studies are provided in the technical report [185].

Password Checker

Program A.7 shows a password checker. The class `PasswordChecker` is annotated with the `@Enclave` annotation. As a result, during run time, the `PasswordChecker` class is placed inside the enclave. The field `password` is annotated with the `@Secret` annotation. The method `checkPassword` returns a boolean which is the result of the comparison of hashes of the field `password` and parameter `guess`. The code snippet without any annotations

```
1  @Enclave
2  class PasswordChecker {
3      @Secret static String password;
4
5      @Gateway
6      public static boolean checkPassword(String guess) {
7          return (getHash(guess) == declassify(getHash(password)));
8      }
9  }
```

Program A.7: Password checker

is a valid Java code. The case study demonstrates the HAA attacker scenario. The parameter **guess** 6 to the gateway method **checkPassword** is controlled by the attacker and is considered untrusted.

Apache Spark

```
1  class MainClass {
2
3      public static void main(String[] args) {
4          JavaSparkContext sc = new JavaSparkContext(new SparkConf().setAppName("Foo"));
5          JavaRDD<EncRec> recList = sc.parallelize(getEncryptedRecords());
6          List<StatRecord> recList = recList.map(x -> StatUtil.process(x));
7      }
8  }
```

Program A.8

```
9   @Enclave
10  class StatUtil  {
11      @Secret static String key;
12
13      @Gateway
14      public static StatRecord process(EncRec rec) {
15          EncRec recE = endorse(rec);
16          Record rec = decrypt(key, recE);
17          // process the decrypted record
18          return declassify(rec);
19      }
20  }
```

Program A.9

Spark [79] is a popular processing framework for big data. It is widely used for machine learning and streaming jobs within clusters. We show a J_E implementation of secure medical data processing using the enclave and Spark. The goal is to process encrypted medical records inside the enclave to extract statistics about the records. Program A.8

represents the code running outside the enclave. We create a Spark context object `sc` (Line 4) and store the encrypted records to be processed in a `JavaRDD` (resilient distributed dataset) object `recList` (Line 5) which is a distributed data structure to store the data to be processed. `EncRec` is an encrypted medical record. The `StatUtil` class (Line 10) contains a gateway method `process` (Line 14) that accepts an encrypted record, it decrypts it with the secret key `key`, and returns the corresponding statistical information `StatRecord`. Hence, a medical record is only decrypted inside the enclave, protecting the medical data. The code without any annotations is a valid Java program, and the programmer only needs to add the annotations without any extra modifications to the original program.

Updatable Password Checker

```
 1  @Enclave
 2  class UpdatablePasswordChecker {
 3      @Secret static String password;
 4
 5      @Gateway
 6      public static void updatePassword(String currPass, String newPass) {
 7          if (checkPassword(currPass) == true) {
 8              password = newPass;
 9          }
10      }
11      @Gateway
12      public static boolean checkPassword(String guess) {
13          boolean result = false;
14          String guessE = endorse(guess);
15          if (endorse(password).equals(guessE)) {   // endorsing password
16              result = true;  // result of type <S,T>
17          }
18          return declassify(result);  // declassifying result
19      }
20  }
```

Program A.10: Updatable password checker

We consider a password checker with a provision for updating the password via a gateway method. Program A.10 shows the class `UpdatablePasswordChecker` annotated with the `@Enclave` annotation. It contains a secret field `password` and two gateway methods, namely, `checkPassword` and `updatePassword`. The `checkPassword` method (Line 12) compares the argument `guess` with the field `password` and returns the result of the comparison as a boolean. Method `updatePassword` (Line 6) assigns the value of the argument `newPass` to the secret field `password` (Line 8). This case study illustrates the HRAA attack scenario. The attacker can reorder the calling sequence of gateway methods. Since the gateway method `updatePassword` assigns an untrusted value to the secret field `password` (Line 8), the secret field can no longer be trusted. Therefore, we infer the security level of the `password` secret field as $\langle \mathbb{S}, \mathbb{U} \rangle$. The field `password` needs to be endorsed (Line 15) to allow the declassification (Line 18). J_E detects secret fields that can be modified by an attacker and the programmer needs to explicitly trust such fields to allow any dependent declassification.

A.8 Related Work

This section compares J_E with closely related work on securing distributed applications with TEEs, focusing on information flow control, frameworks for TEEs, and secure program partitioning.

Information flow control and enclaves

Recent works leverage IFC and enclaves to build applications with strong security guarantees. These works target specific challenges in the domain space, including security foundations of enclave applications, secure code partitioning for enclaves and language support for programming in TEEs. J_E pushes the boundary in several directions providing: (a) language support for Java applications in Intel SGX; (b) a security type system enabling verification of secure partitioning against realistic attacker models; (c) formalization and soundness proofs with respect to semantic characterization of security; (d) a prototype implementation supported by case studies.

Sinha et al. [119] present Moat, a system for statically verifying confidentiality properties of Intel SGX programs in the presence of passive and active attackers. They combine a flow sensitive type system with automated theorem proving to check confidentiality policies with declassification. Their active attacker is similar to our HAA attacker, assuming arbitrary modification of non-SGX code. By contrast, we focus on verifying robustness of an application against active attacker which has the advantage of using a more lightweight analysis whenever a program is already proved secure against passive attackers. Moreover, we consider more powerful attackers, and target languages with managed runtimes, thus shielding developers from SGX-specific details. Follow-up work by Sinha et al. [130] enforces Information Release Confinement, an access control policy which allows SGX code to perform arbitrary computations ensuring that it can only generate output to the non-SGX memory through encrypted channels. While this approach cannot enforce expressive information flow policies, it implements Control Flow Integrity to ensure that an active attacker does not compromise the control flow of an executions. Our approach would require similar techniques to enforce security against the HRAA attacker.

Inspired by Moat, Gollamudi and Chong [126] propose a security type system to enforce flexible information flow policies for an SGX-enabled imperative language. They abstract away the details of enclave management and develop a translation tool to automatically infer the parts of the program to be executed inside the enclaves. They consider active attackers and formalize security using knowledge-based conditions. By contrast, our approach enforces robustness for a mainstream language like Java. Recent work by Gollamudi et al. [162] uses enclaves to enforce more expressive confidentiality and integrity against passive attackers in a distributed setting. Like us, both works come with soundness proofs of security, with the key difference that our work targets robustness instead of variants of noninterference. Liu et al. [139, 164] study automated program partitioning with SGX for passive attackers. We argue that passive attackers are too weak in the context of SGX, hence our work provides the formal grounds for extending and proving their techniques in a realistic setting.

A large array of works study IFC in distributed settings [24, 27, 31, 52, 53, 138]. These works address the problem of secure program partitioning across nodes in a distributed system under the assumption that low integrity nodes may be controlled by an active attacker. Our work draws inspiration from these approaches and extends them to capture attacker models arising in TEEs.

Existing works target the foundations of IFC in presence of active attackers [34, 63, 67, 71, 80, 134] and enforce robustness via security type systems.

Programming frameworks for TEEs

To reduce the trusted computing base, in contrast to running unmodified applications inside SGX, some approaches focus on partitioning the application into components that execute within the enclave and the rest, which executes in the untrusted environment. Glamdring [137] propose the first source-level partitioning framework for securing C applications with Intel SGX. Developers annotate sensitive data and Glamdring automatically partitions the application into untrusted and enclave parts. Panoply [142] generates application binaries from the annotated source code where annotations specify the parts of the application to be run inside separate enclaves. DynSGX [143] provides tools to dynamically load, unload and execute compiled functions inside SGX enclaves efficiently. While these works leverage enclaves to enforce strong isolation properties, they do not enforce application-level policies and lack formal security guarantees.

Several works aim to ease the programming of applications that (partially) execute within an enclave. Coppolino et al. [160] present a comparative analysis of the existing approaches for securing Java applications with Intel SGX. RUST-SGX [167] provides memory-safe SGX support for Rust through a memory management scheme to control the interface between Rust and Intel's C/C++ APIs. Civet [178] is a programming framework for Java using an XML file to specify the classes that are executed inside the enclave. Civet leverages dynamic information flow control to track insecure flows within the enclave. Uranus [170] supports executing Java functions inside SGX enclaves. It provides two method-level annotations `JECall` and `JOCall` to indicate methods to be executed inside and outside the enclave respectively. Secure Routines [161] is a programming framework for SGX in the Go language. Programmers can execute Go functions (goroutine) inside an enclave, and use low-overhead channels to communicate with the untrusted environments. Like us, these works aim at developing programming models with enclaves. However they lack provable security guarantees and require security analysis of the partitioned application from scratch. Other works propose processor model calculi to capture necessary conditions for safe remote execution of enclave programs. Subramanyan et al. [146] introduce an abstract processor model to verify the security guarantees of Intel SGX under specific adversary capabilities. Sinha [144] studies confidentiality risks that can be exploited by application and infrastructure attacks in SGX applications. Autarky [174] is a controlled-channel attack resistant framework based on a modified SGX ISA.

Running unmodified applications inside SGX

Haven [113] was the first system built to run unmodified Windows legacy applications inside Intel SGX. SCONE [121] is a Docker extension that uses SGX to protect container processes. SGX-LKL [166], SGXKernel [147], Graphene-SGX [148], and Occlum [177] are library OS based frameworks designed to run unmodified Linux applications inside the SGX enclave.

Multitier programming and secure program partitioning

The J_E programming model is inspired by the multitier programming paradigm [44, 48, 156] – for a comprehensive overview of multitier programming, we refer to the survey by Weisenburger et al. [179]. In multitier programming, the code for different tiers is written as a single compilation unit and the compiler automatically splits it into the components associated to each individual tier. Different works extend the multitier programming model with information flow policies to build secure web applications via security type systems [66, 110, 122, 165] and symbolic execution [115]. In contrast to J_E, none of these approaches enforce robustness properties.

A.9 Conclusion

In this paper we present J_E, a language for confidential computing which supports enclave-enabled applications. First, J_E seamlessly integrates with a high-level, managed language, and enables programmers to develop secure enclave-enabled applications by adding annotations to Java programs. Second, J_E comes with a security model that accounts for realistic attackers, that, in the case of enclave programming, can tamper with the code and the data of the non-enclave environment. We define the notion of *robustness* of enclave-enabled programs and prove that it is correctly enforced by the J_E type system.

We evaluate our approach on several use cases from the literature, including a battleship game, a secure event processing system, and a popular processing framework for big data, showing that it can correctly handle complex cases of partitioning, information flow, declassification and trust.

We envision different avenues for future work including a generalization of our framework to multiple enclaves and nonmalleable information flow. On the practical side, we plan to extend our automated partitioning and compilation algorithms to handle Java programs beyond the core J_E fragment.

Acknowledgments

Thanks are due to Owen Arden and anonymous reviewers for their helpful feedback on this paper. This work is partially supported by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the BRF Project 1025524 from the University of St.Gallen, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

Appendices

Appendix A Proofs

Attacker Code

In this section, we present proofs for Lemma A.1 and Lemma A.2.

Lemma A.1

The attack code definition presented in A.1, captures the most powerful attack strategies available to an HAA attacker, who controls the non-enclave environment.

Proof. Because HAA attacker is in control of the non-enclave environment, he can potentially insert his code anywhere in the program. But the effects of his code is limited to what is presented in A.1. Because there are no secret values in non-enclave environment, all of the active attacker's commands executed outside of enclave can eventually have one of the following effects: modifying the parameters passed to a gateway during gateway method calls, or making the program diverge. We don't consider the case with divergence, because in this case the active attacker makes even less observations than the passive attacker.

We justify this argument by structural induction on the possible commands available to HAA attacker:

skip This case is straightforward because it has no effect on the return value of gateways.

$x := e_G$ If x is not one of the parameters passed to a gateway (i.e. $\overline{p}$), then this assignment has no effect on the values inside of enclave. The case that x is one of gateway parameters is captured by attack $q := e$. We also have to consider the cases when e_G is a method call, either to a gateway or a non-enclave method. *We limit HAA attacker from calling additional gateway methods, and this ability will be captured by HRAA attacker.* On the other hand if e_G is a non-enclave method call, we can simply replace $x := e_G$ with $x := e'$, where e' is the value returned from the method.

$C.f := e_G$ Similar to the assignment case.

$if\ e\ then\ S_1\ else\ S_2$ Because this command is outside of enclave, e is either public value from σ, or a value set by the attacker. In both cases, the attacker knows its value, thus instead of writing an if statement, he can directly write the command S_1 or S_2. Moreover, the attacker can only have an effect on the values inside enclave if S_1 or S_2 update the values of gateway parameters. Thus, this case can be reduced to a command of form $q := e$ which captures the intended effect of $if\ e\ then\ S_1\ else\ S_2$.

while e do S This case is similar to the conditional, except that by using a non-terminating loop, the attacker can make the program diverge and limit his own observations. We *ignore* this kind of active attacker because it makes even less observations than the passive attacker.

$S_1; S_2$ we have three cases:

- Neither S_1 nor S_2 modifies the gateway parameter, then this command will have no effect on the enclave, hence the attack code can be ignored.
- Only one of S_1 or S_2 modify the parameters of a gateway method. Without loss of generality, assume S_2 is the command that makes this modification. Using the above cases, we can combine $S_1; S_2$ to a command S of the form $q := e$ that has the same effect as $S_1; S_2$.
- Both of S_1 and S_2 modify gateway parameters. In this case $S_1; S_2$ can be reduced to $q_1 := e; q_2 := e'$, and this structure is presentable by the syntax of a in A.1.

□

Lemma A.2

The attacker code defined in A.1, captures the most powerful attack strategies available to the HRAA attacker.

Proof. The HRAA attacker has complete control over the non-enclave environment including code and memory, and he can potentially modify the code outside of the enclave in anyway he wants. Since sensitive data is stored inside the enclave and is only accessible via gateways, the HRAA attacker can only leverage his power by calling the gateways in any order and with any frequency, and the effects of his code is limited to the attack code presented in A.1.

The HRAA attacker can execute any J_E program that uses the gateways, but this will not make them more powerful than the attacks presented in A.1. The HRAA attacker's commands executed outside of enclave can eventually have one of the following effects: modifying the parameters passed to a gateway during gateway method calls, or making the program diverge.

Because the proof of this lemma is very similar to the proof of Lemma A.1, we only investigate some of the most interesting commands available to HRAA attacker, as the rest of them can be reconstructed straightforwardly from the proof of Lemma A.1.

$x := e_G$ If x is not one of the parameters passed to a gateway (i.e. $\overline{p}$), then this assignment has no effect on the values inside of enclave. The case that x is one of gateway parameters is captured by attack $q := e$. We also have to consider the cases when e_G is a method call, either to a gateway or a non-enclave method. HRAA attacker can call additional gateway methods, and the case that e_G is a gateway method call can be captured by Definition A.4 and the ability of HRAA attacker to call any gateway methods. On the other hand if e_G is a non-enclave method call and it does not call any gateways, we can argue that since everything outside of enclave is public, the attacker can simply replace $x := e_G$ with $x := e'$, where e' is the value returned from the method. If e_G is a non-enclave method call and it does call some other gateways, the attacker can capture the effect of running $x := e_G$ by simply calling the gateway which was inside e_G.

$\mathit{if}\ e\ \mathit{then}\ S_1\ \mathit{else}\ S_2$ Since this command is outside of the enclave, e is either a public value, or a value set by the attacker. In both cases, the attacker knows its value, therefore instead of writing the if statement, he can directly write the command S_1 or S_2, and since S_1 or S_2 can have one or a sequence of gateway calls, The behavior of the if statement can be captured by Definition A.4 as one or a sequence of gateway calls.

$\mathit{while}\ e\ \mathit{do}\ S$ In this case, similar to the if statement, the attacker knows the value of e. We should consider 4 cases:

- The loop is finite and S has no gateway calls: The attacker can only have an effect on the values inside enclave if S updates the values of gateway parameters. Thus, this case can be reduced to a command of form $q := e$ which captures the intended effect of $\mathit{while}\ e\ \mathit{do}\ S$.
- The loop is finite and S has gateway calls: Which means that it can be represented by Definition A.4 as a sequence of gateway calls.
- The loop is infinite: It does not matter whether S has gateway calls or not. Since the execution never terminates and Definition A.2 is only defined on terminating programs, we can *ignore* this case.

□

Type System Soundness

In this section, we present the proof of Theorem A.1. We have to show that if a program is well-typed, then changing the attack vector $\overrightarrow{a}$ will not change the attacker's observations. We use events and traces to keep track of these observations. In our setting, only gateway method expressions $(e.m(\overline{p}))$ emit an event, and this event is the return value of the gateway method. So, before proving Theorem A.1, we present two lemmas about gateway method calls.

Based on the syntax of our language, a gateway method can only be executed in an Assignment or a Store command. First we present a lemma for the assignment:

Lemma A.3 (Assignment gateway method calls)

Suppose that $S[\overrightarrow{\bullet}]$ *is* $[\bullet]; x := C.m(\overline{p})$. *If* $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$, *Class* C *is inside enclave, and* m *is a gateway method, then for all possible attacks* $\overrightarrow{a}$ *and* $\overrightarrow{a'}$, *if* $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ *then* $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$.

Proof. In our system, the body of a method is type-checked in isolation using the rule T-Method. Because we are in a Non-Enclave–Enclave context, method m is definitely a gateway, the body of the method no longer has any holes, and all of the parameters of the method ($\overline{p}$) are now variables with the label $(\langle \mathbb{P}, \mathbb{U}\rangle, F)$.

The event that a method call emits is determined by its return value. In other words, the last command of a method's body is $return(e)$ and the value of the expression e is the event. By induction on the structure of S:

skip This case is trivial, as it does not affect the emitted event.

$x := e_G$ By rule T-Assign, the label of variable x is updated according to the label of e_G and current context *pc*. Thus, if e_G or the context are affected by attacker's inputs, the resulting label will be untrusted $\mathbb{U}$. Note that e_G can be a method expression, this case will be an enclave–enclave method call, so it will not produce any events, and even though there are no explicit holes in this command, the attacker's inputs can still be passed as one of the method's parameters, so it is possible to have a return value that is untrusted. Additionally, if e_G is affected by some declassified value, it is going to have a True flag, and by rule T-Assign, this assignment can only happen in trusted contexts.

$C.f := e_G$ This case is very similar to the Assign case, but more limited because of the flow insensitivity of T-Store. C is a class inside of enclave, so f's security label is either $\langle \mathbb{P}, \mathbb{T}\rangle$ or $\langle \mathbb{S}, \mathbb{T}\rangle$. In any case, its value can only be updated in trusted context and by trusted values. So, changing the attack vector cannot have any effect on a trusted field's value.

$x := declassify(e)$ Using T-Declassify rule, we ensure that expression e should be trusted and it can only be declassified in a trusted context. Thus, the resulting variable x always has $\langle \mathbb{P}, \mathbb{T}\rangle$ label. However, because e is secret, its value can be different in σ_1 and σ_2. To keep track of the propagation of this declassified value, and control its delayed release, in rule T-Declassify we make sure x's flag is True.

$return(e)$ For gateway calls, the rule T-Assign-Gateway-Call uses the precondition $\ell \sqsubseteq \langle \mathbb{P}, \mathbb{U}\rangle$ to ensure the return value's label is public, but based on whether it was affected by the attacker or not, it can be trusted or untrusted. Since the assumption $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma$ implies that $S[\overrightarrow{\bullet}]$ is well-typed then e's label is indeed $\sqsubseteq \langle \mathbb{P}, \mathbb{U}\rangle$. Having this assumption, we continue by induction on the structure of e:

- v: This case is trivial. The return value is independent of the attacks. In other words

$$\begin{gathered} Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle) = \\ Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle) \\ = v \end{gathered}$$

- x: Following the semantics of ASSIGN ($x := e_G$), the value of x was determined by some expression e_G. We have to consider four cases:
  - e_G security label is $\langle \mathbb{P}, \mathbb{T}\rangle$ and its flag is False: Because e_G value was not affected by any declassified value, and because the security label is public and trusted, following the initial indistinguishability of states $\sigma_1 =_A \sigma_2$,

  and the fact that trusted values are not effected by attacker inputs, we can easily conclude e_G evaluates to the same value in $\langle S[\overrightarrow{a}], \sigma_1\rangle$, $\langle S[\overrightarrow{a'}], \sigma_1\rangle$, $\langle S[\overrightarrow{a}], \sigma_2\rangle$, and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$.

  - e_G security label is $\langle \mathbb{P}, \mathbb{T}\rangle$ and its flag is True: e_G is affected by declassified values, and because its security label is trusted, we know that it was not affected by the attacker's input. Obviously, the declassified values might be different in σ_1 and σ_2, however, by the initial indistinguishability of states $\sigma_1 =_A \sigma_2$ and the assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$, we can conclude that the declassified values were equal in σ_1 and σ_2. Thus it is straightforward to deduce that e_G will evaluate to equal values in $\langle S[\overrightarrow{a}], \sigma_1\rangle$, $\langle S[\overrightarrow{a'}], \sigma_1\rangle$, $\langle S[\overrightarrow{a}], \sigma_2\rangle$, and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$.
  - e_G security label is $\langle \mathbb{P}, \mathbb{U}\rangle$ and its flag is False: e_G value was not affected by any declassified value, it was just affected by the public values and the attacker's inputs. So following the initial indistinguishability of states $\sigma_1 =_A \sigma_2$ and the assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$, we can show that $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ will also hold. This is because the public values are unchanged, and the only difference is in the attacker's input, which is the same in both $\langle S[\overrightarrow{a'}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$. Thus, even thought the return value is *possibly* different, it is still going to be the same in $\langle S[\overrightarrow{a'}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$, hence $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$.
  - e_G security label is $\langle \mathbb{P}, \mathbb{U}\rangle$ and its flag is True: This means that e_G was effected by both attacker input and declassified values. However, by rules T-Assign and T-Store we know that after getting affected by declassified values, it was only affected by untrusted values directly (*not* through a condition or loop). Thus by assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$ we can learn that the declassified value is the same in σ_1 and σ_2. The rest is similar to the previous case, when the attacker's input changes, $\langle S[\overrightarrow{a'}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$ might produce *possibly* different yet still equal events. Thus $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ will still hold.

- $C.f$: Following the semantics of Store ($C.f := e_G$), the value of $C.f$ was determined by some expression e_G. This case is easier than the assignment because C is a class inside of enclave, so the security label of its field f can be either $\langle \mathbb{P}, \mathbb{T}\rangle$ or $\langle \mathbb{S}, \mathbb{T}\rangle$. By rule T-Assign-Gateway-Call we already know that the return value has a label $\sqsubseteq \langle \mathbb{P}, \mathbb{U}\rangle$, thus $C.f$'s label is definitely $\langle \mathbb{P}, \mathbb{T}\rangle$. This label means that the value stored in $C.f$ is not affected by the attacker's input, so following the initial indistinguishability of states $\sigma_1 =_A \sigma_2$, and the assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$, we can conclude that $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ will also hold.
- $e_1 \oplus e_2$: By induction on the structure of e_1 and e_2, we see that they can be one of the above cases. Thus it is straightforward to show that $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ will hold.

if e then S_1 *else* S_2 We have four cases:

- *e is public and trusted*: Because e is public and trusted it evaluates to equal values in $\langle S[\overrightarrow{a}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_1\rangle$ and (in case of declassification) to *possibly different yet still equal* values in $\langle S[\overrightarrow{a}], \sigma_2\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$. Therefore, by assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$, its easy to conclude that the if statement takes the same branch and $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ also holds.
- *e is public and untrusted*: In this case neither S_1 nor S_2 can have any declassification. A public value evaluates to the same values in σ_1 and σ_2, however because e is also untrusted, it can also been affected by attacker's input. However because attacker's input only changes when the attack vector changes, we can easily conclude that by having the initial indistinguishability of states $\sigma_1 =_A \sigma_2$, e will evaluate to equal values in $\langle S[\overrightarrow{a}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a}], \sigma_2\rangle$ and (based on the attacker's input) to *possibly different yet still equal* values in $\langle S[\overrightarrow{a'}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$. So it is possible for the if condition to take different branches when the attacker vector changes. Now because e is untrusted, *pc*'s security label also becomes untrusted, rules T-Assign and T-Store, prevent the assignment or store of declassified values in the body of the if statement. So the values that depend on declassified values cannot be affected by this condition. (ultimately this is what we wanted to achieve by using flags, to disallow attacker input to control the branch that can indirectly affect the final declassification value). From here on it is straightforward to show that by the initial indistinguishability of states $\sigma_1 =_A \sigma_2$ and the assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$, $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ will also hold.
- *e is secret and trusted*: By rule T-Declassify neither S_1 nor S_2 can have any declassification. Any assignment or store inside S_1 and S_2 is going to be secret, and have to be declassified in order to be able to affect the return value. We can use this intuition, the initial indistinguishability of states $\sigma_1 =_A \sigma_2$, and the assumption $Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle)$ to conclude that $Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle) = Trace(N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle)$ holds.
- *e is secret and untrusted*: This case is similar to the last one, but by rule T-Store any store command is not possible in S_1 and S_2 and because the result of any assignment inside S_1 and S_2 is going to be secret and untrusted, it cannot be declassified later on, so it can never effect the return value.

while e do S This case is similar to the case for conditionals. The only difference is that in this case, untrusted attacker inputs can cause the program to diverge, and by definition of trace indistinguishability, we consider non-terminating traces indistinguishable.

$S_1; S_2$ We have two cases:

- $S_1; return(e)$: S_1 can be any of the above commands except $return(e)$. Additionally, we can use an argument similar to the one we used for $return(e)$ case to show that the return value will be the same for programs $S[\overrightarrow{a}]$ and $S[\overrightarrow{a'}]$.

- $S_1; S_2$: This case is fairly straightforward, because neither S_1 nor S_2 have a return command, they do not affect the trace.

□

We also need another Lemma for calling a gateway method through the Store command:

Lemma A.4 (Store gateway method calls)

Suppose that $S[\overrightarrow{\bullet}]$ *is* $[\bullet]; C'.f := C.m(\overline{p})$. *If* $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$, *Class* C' *is outside the enclave,* C *is inside enclave, and* m *is a gateway method, then for all possible attacks* $\overrightarrow{a}$ *and* $\overrightarrow{a'}$, *if* $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ *then* $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$.

Proof. Similar to Lemma A.3. □

Now, using Lemma A.3 and Lemma A.4, we can prove Theorem A.1.

Theorem A.1

If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HAA.*

Proof of Theorem A.1. Following the approach of [34], suppose that for the well-typed program $S[\overrightarrow{a}]$, and states $\sigma_1 =_A \sigma_2$, we have $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$. We need to show that for all possible attacks a', the execution indistinguishability holds. Formally:

$$\forall \overrightarrow{a'}.\ N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$$

Execution is in non-enclave environment, and the security label of all variables, fields and classes outside of enclave is $\langle \mathbb{P}, \mathbb{U}\rangle$. It is worth noting that the values returned from gateways can be $\langle \mathbb{P}, \mathbb{T}\rangle$, but because the security label of pc outside of enclave is $\langle \mathbb{P}, \mathbb{U}\rangle$, the final label of the variable holding this returned value will be $\langle \mathbb{P}, \mathbb{T}\rangle \sqcup \langle \mathbb{P}, \mathbb{U}\rangle = \langle \mathbb{P}, \mathbb{U}\rangle$, furthermore, since all of the fields outside of enclave are $\langle \mathbb{P}, \mathbb{U}\rangle$, by sub-typing rule we can store a $\langle \mathbb{P}, \mathbb{T}\rangle$ inside it, but it will be treated as a $\langle \mathbb{P}, \mathbb{U}\rangle$ after that.

We present the proof by structural induction on $S[\overrightarrow{\bullet}]$. It is worth noting that we will not consider $x := \mathit{declassiy}(e)$ command, because the semantics of Declassify ensures that this command is only executed inside of enclave.

If $S[\overrightarrow{\bullet}]$ is Skip then we are done. Because it does not have a hole. So we continue by structural induction for the other cases of $S[\overrightarrow{a}]$:

$[\overrightarrow{a}]; x := e_G$**:** If e_G is not a method expression, then it will not have any effect on the trace, and we are done. If e_G is a method expression $e.m(\overline{p})$, we have to consider two cases: If m is a method outside of enclave it does not produce any events, so the trace remains unchanged. If m is a gateway method, by Lemma A.3, we already know that its returned value (event) will be indistinguishable for all attacks, thus $\forall \overrightarrow{a'}.\ N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$.

$[\overrightarrow{a}]; C'.f := e_G$**:** Similar to the previous case. If e_G is not a method expression or m is outside of enclave, we are done. Otherwise, if e_G is a method expression $e.m(\overline{p})$ and m is a gateway method, by Lemma A.4, we know that the return values (events) are indistinguishable for all attacks, thus $\forall \overrightarrow{a'}.\ N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$ will hold.

$\mathit{if}\ e\ \mathit{then}\ S_1[\overrightarrow{a_1}]\ \mathit{else}\ S_2[\overrightarrow{a_2}]$**:** If $S_1[\overrightarrow{a_1}]$ and $S_2[\overrightarrow{a_2}]$ does not have gateway calls, then executing this command will have no effect on the trace. So we only consider cases in which at least one of $S_1[\overrightarrow{a_1}]$ or $S_2[\overrightarrow{a_2}]$ has a gateway method call.

By the fact that the execution is outside of enclave we know that e is public. By using the initial indistinguishability of states $\sigma_1 =_A \sigma_2$, and the results of Lemmas A.3 and A.4, we can infer that e will evaluate to the same value v in both configurations $\langle e, \sigma_1\rangle \Downarrow \langle v, \sigma_1\rangle$ and $\langle e, \sigma_2\rangle \Downarrow \langle v, \sigma_2\rangle$. Therefore, the conditional takes the same branch in both configurations.

Without loss of generality, assume that v is non-zero, then configurations $\langle S[\overrightarrow{a}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a}], \sigma_2\rangle$ will be reduced to $\langle S_1[\overrightarrow{a_1}], \sigma_1\rangle$ and $\langle S_1[\overrightarrow{a_1}], \sigma_2\rangle$, respectively. Similarly, $\langle S[\overrightarrow{a'}], \sigma_1\rangle$ will reduce to $\langle S_1[\overrightarrow{a'_1}], \sigma_1\rangle$ and $\langle S[\overrightarrow{a'}], \sigma_2\rangle$ will reduce to $\langle S_1[\overrightarrow{a'_1}], \sigma_2\rangle$. By assumption $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ we have $N \vdash_\delta \langle S_1[\overrightarrow{a_1}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a_1}], \sigma_2\rangle$. So, by induction hypothesis we can obtain $N \vdash_\delta \langle S_1[\overrightarrow{a'_1}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a'_1}], \sigma_2\rangle$. Which in turn implies that $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$.

$\mathit{while}\ e\ \mathit{do}\ S_1[\overrightarrow{a}]$**:** We only consider the case in which $S_1[\overrightarrow{a}]$ has a gateway method call. By an argument similar to the conditional, we can show e will evaluate to the same value v in both σ_1 and σ_2. If v is zero, then both configurations $\langle S_1[\overrightarrow{a'}], \sigma_1\rangle$ and $\langle S_1[\overrightarrow{a'}], \sigma_2\rangle$ will terminate without making any changes to the trace. Thus $N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_2[\overrightarrow{a'}], \sigma_2\rangle$ will hold.

If v is non-zero, then by $N \vdash_\delta \langle \mathit{while}\ e\ \mathit{do}\ S_1[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle \mathit{while}\ e\ \mathit{do}\ S_1[\overrightarrow{a}], \sigma_2\rangle$, it is necessary that $N \vdash_\delta \langle S_1[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a}], \sigma_2\rangle$ holds. Then by induction hypothesis we have $N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_2\rangle$. If either $\langle S_1[\overrightarrow{a'}], \sigma_1\rangle$ or $\langle S_1[\overrightarrow{a'}], \sigma_2\rangle$ diverges, then its corresponding loop $\langle \mathit{while}\ e\ \mathit{do}\ S_1[\overrightarrow{a'}], \sigma_1\rangle$ or $\langle \mathit{while}\ e\ \mathit{do}\ S_1[\overrightarrow{a'}], \sigma_2\rangle$ also diverges. Then by definition of execution indistinguishability, $N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_2\rangle$ holds. On the other hand, if both $\langle S_1[\overrightarrow{a'}], \sigma_1\rangle$ and $\langle S_1[\overrightarrow{a'}], \sigma_2\rangle$ terminate, it is straightforward to show that the gateway calls inside of $S_1[\overrightarrow{a'}]$ are issued the same number of times under σ_1 and σ_2. By Lemmas A.3 and A.4, we already know that events resulted from these gateway calls are indistinguishable, thus the traces of $N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_1\rangle$ and $N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_2\rangle$ will be indistinguishable, meaning that $N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a'}], \sigma_2\rangle$ holds.

$S_1[\overrightarrow{a_1}]; S_2[\overrightarrow{a_2}]$**:** We only consider the cases in which $S_1[\overrightarrow{a_1}]$ and $S_2[\overrightarrow{a_2}]$ have gateway method calls. By induction on structure of $S_1[\overrightarrow{a'_1}]$ we can show that it either diverges or terminates. Similar to the case of while loop, if it diverges in at least one of the configurations, by definition $N \vdash_\delta \langle S[\overrightarrow{a'_1}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'_1}], \sigma_2\rangle$ holds. If it terminates in both configurations, we can use the result of Lemmas A.3 and A.4 to infer that these execution will result in indistinguishable traces. Thus $N \vdash_\delta \langle S_1[\overrightarrow{a_1'}], \sigma_1\rangle$ $N \vdash_\delta \approx_A \langle S_1[\overrightarrow{a_1'}], \sigma_2\rangle$ will hold.

If $S_1[\overrightarrow{a_1'}]$ terminates in both configurations, we can consider $S_2[\overrightarrow{a_2'}]$. Imagine $\langle S_1[\overrightarrow{a_1'}], \sigma_1\rangle \Downarrow_t \mathcal{M}'_1, \mathcal{H}'_1$ and $\langle S_1[\overrightarrow{a_1'}], \sigma_2\rangle \Downarrow_t \mathcal{M}'_2, \mathcal{H}'_2$. By the result of Lemmas A.3 and A.4 and initial state indistinguishability, it is fairly straightforward to show that $\mathcal{M}'_1 =_A \mathcal{M}'_2$ and $\mathcal{H}'_1 =_A \mathcal{H}'_2$. Now, a similar argument can be used for $S_2[\overrightarrow{a_2'}]$ to infer that $N \vdash_\delta \langle S_2[\overrightarrow{a_2'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1[\overrightarrow{a_2'}], \sigma_2\rangle$ also holds.

□

We can present the proof of Theorem A.2 similar to the proof of Theorem A.1. First, we change Lemma A.3 to account for the *untrusted* label of the values in shared state Σ.

Lemma A.5 (Gateway method calls)

Suppose that $S[\overrightarrow{\bullet}]$ *is* $[\bullet]; x := C.m(\overline{p})$. *If* $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *w.r.t HRAA attacker, Class* C *is inside enclave, and* m *is a gateway method, then for all possible attacks* $\overrightarrow{a}$ *and* $\overrightarrow{a'}$, *if* $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ *then* $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$.

Notice that the definition of Lemma A.5 is similar to Lemma A.3. This is because we are not considering any ordering of gateway calls yet, we are just using this lemma to show that the indistinguishability of gateways' return values is independent of the *attacker input* and the *shared state*.

Only Fields can act as shared states, since variables are local. In rule T-Store, $pc_d \vee d_2 = T \Rightarrow pc_\ell \sqcup \ell_1 = \langle -, \mathbb{T}\rangle$ ensures that the value of shared state (which is untrusted) cannot directly or indirectly be affected by declassified values.

To prove this lemma we just have to extend the definition of untrusted value to attacker inputs and all of the fields in Σ. Gateway return value's independence from attacker input can be shown from the change in attack vector a', and its independence of shared state can be proved from the public indistinguishability of states σ_1 and σ_2, and the untrustedness of shared state. We omit this proof because its structure is similar to the proof presented for Lemma A.3.

Now, using this lemma we can prove Theorem A.2.

Theorem A.2

If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *with regard to* Σ *and* G^D, *then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HRAA.*

Proof of Theorem A.2. Suppose that for the well-typed program with fixed order of execution $S[\overrightarrow{a}]$, and states $\sigma_1 =_A \sigma_2$, we have $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$. We need to show that in all possible ordering and frequency of gateways calls, and for all possible values for the parameters of gateways, the execution indistinguishability holds. Formally:

$$\forall S', a'.\ N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$$

Using Lemma A.5, by induction on the structure of S':

$[\overrightarrow{a'}]; x := C.m(\overline{p})$**:** If $C.m(\overline{p})$ was a gateway method originally called in $S[\overrightarrow{\bullet}]$, $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ is going to be a direct result of Lemma A.7, which shows that the indistinguishability of gateways' return values (and its resulting event) is independent of the value of parameters passed to them by the attacker and the shared state.

However, it is also possible that $C.m(\overline{p})$ is a gateway method, not originally called in $S[\overrightarrow{\bullet}]$. Because of the preprocessing step of G^D we can conclude that $C.m(\overline{p})$ does not have any *new* declassifications, by initial indistinguishability of states σ_1 and σ_2, and equality of attack vector and shared state on both sides, we can conclude that $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ holds.

$S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$**:** We know that both $S'_1[\overrightarrow{a'_1}]$ and $S'_2[\overrightarrow{a'_2}]$ have gateway calls. By Rule T-Declassify we know that untrusted values cannot have any effect on the decision to declassify. Therefore, we can conclude that changing the order or frequency of gateway calls cannot lead to any *new* declassifications. By the assumption $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$, and the fact that any of the gateway calls in $S'_1[\overrightarrow{a'_1}]$ and $S'_2[\overrightarrow{a'_2}]$ that had a declassification were also in $S[\overrightarrow{a}]$ (because G^D was a subset of it), we conclude that declassified secrets had the same value in σ_1 and σ_2. We know that the return value of gateway calls can depend on declassified secrets (which are equal in σ_1 and σ_2), attacker input (which is the same ($\overrightarrow{a'}$) for both programs), public values (that are equal in both states), and the values in shared state. Of these values, only shared state can be different between $S[\overrightarrow{\bullet}]$ and $S'[\overrightarrow{\bullet}]$. We know that they cannot effect declassifications, we also know by the definition of shared state that their value might change by the order of execution, resulting in different traces for $S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$ compared to any other order of calling these gateways. However, since their value is determined by the order of execution, we are going to get similar traces for each possible order of execution. Thus, we can conclude that $N \vdash_\delta \langle S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}], \sigma_2\rangle$ will always hold no matter the order of gateway method calls in $S'_1[\overrightarrow{a'_1}]$ and $S'_2[\overrightarrow{a'_2}]$.

Using these cases it is straightforward to conclude that the indistinguishability of traces will be preserved $\forall S', a'.\ N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$. □

Type Soundness for Endorsement

In this section, we present the proof of Theorem A.3 which follows a similar structure as the proof of Theorem A.1. We just have to extend Lemma A.3 and Lemma A.4 to account for relevant attacks. We revisit Lemma A.3:

Lemma A.6 (Assignment gateway method calls)

Suppose that $S[\overrightarrow{\bullet}]$ is $[\bullet]; x := C.m(\overline{p})$. If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$, Class C is inside enclave, and m is a gateway method, then for all possible attacks $\overrightarrow{a}$ and $\overrightarrow{a'}$ such that $\overrightarrow{a'} \in \Omega(S[\overrightarrow{a}], \sigma_1)$, if $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ then $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$.

All of the parameters of the method $(\overline{p})$ are now variables in the body of method m with the label $(\langle \mathbb{P}, \mathbb{U}\rangle, F)$. Let's continue by induction on the structure of S for some of the interesting cases:

$x := e_G$ Following the rule T-Assign, the label of variable x is updated according to the label of e_G and current context pc. Thus, if e_G or the context are affected by attacker's inputs, the resulting label will be untrusted $\mathbb{U}$. However, now it is possible that e_G or context were endorsed. In this case we can use the fact that a' is a relevant attacks to infer that the endorsed values will be the same in $S[\overrightarrow{a}]$ and $S[\overrightarrow{a'}]$. Additionally, if e_G was affected by some declassified value it is going to have a True flag, and if it eventually affects the return value of the method, we can use $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ to conclude the declassified secrets were equal in states σ_1 and σ_2.

$x := endorse_\eta(e)$ Rule T-Endorse ensures that e can only be endorsed in a trusted context. Using this rule, combined with assumption $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ and the fact the a' is an attack relevant to a, allows us to conclude that the result of endorsement will be equal under a and a'.

$x := declassify(e)$ Rule T-Declassify ensures that expression e should be trusted and it can only be declassified in a trusted context. But now that we have endorsement, it is possible that e or the context were affected by attacker's input. However, because a' is a relevant attack, we can be sure that endorsed values are the same under a and a'. Moreover, because e can be secret, its value can be different in σ_1 and σ_2. To keep track of the propagation of this declassified value, and control its delayed release, rule T-Declassify makes x's flag True. Now if e is eventually returned, we can conclude the result of declassification was equal under a and a', so the indistinguishability of traces $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$ will also hold.

We should also revisit Lemma A.4, but its proof is going to be similar to that of Lemma A.6. Utilizing these lemmas, the proof of Theorem A.3 will be completely similar to the proof of Theorem A.1.

The proof of Theorem A.4 is similar to the proof of Theorem A.2, with two main differences:

1. Variables and fields in shared state are untrusted, and therefore can be endorsed.
2. It is possible that $S'[\overrightarrow{\bullet}]$ lead to additional endorsement.

Similar to the proof of Theorem A.2, we propose a lemma to check the indistinguishability of single gateway calls.

Lemma A.7 (Gateway method calls)

Suppose that $S[\overrightarrow{\bullet}]$ *is* $[\bullet]; x := C.m(\overline{p})$ *. If* $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *w.r.t HRAA attacker, Class* C *is inside enclave, and* m *is a gateway method, then for all possible attacks* $\overrightarrow{a}$ *and* $\overrightarrow{a'}$ *such that* $\overrightarrow{a'} \in \Omega_R(S[\overrightarrow{a}], S[\overrightarrow{\bullet}], \sigma_1)$*, if* $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$ *then* $N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a'}], \sigma_2\rangle$*.*

Notice that in Lemma A.7, in the definition of relevant attacks $S'[\overrightarrow{\bullet}]$ is equal to $S[\overrightarrow{\bullet}]$. This is because we are not considering any ordering of gateway calls yet. However, the values of attacker input and *untrusted* shared state can be endorsed, and affect the return value. nevertheless, because a' is a relevant attack, we know that the endorsed value are the same in $S[\overrightarrow{a}]$ and $S[\overrightarrow{a'}]$, regardless of whether they originated from the attacker inputs or the shared state. We omit its proof because it is similar to the proof presented for Lemma A.6.

Now, using this lemma we can prove Theorem A.4.

Theorem A.4

If $pc, \Gamma, \Pi \vdash_\delta S[\overrightarrow{\bullet}] : \Gamma', \Pi'$ *with regard to* Σ *and* G^D*, then* $S[\overrightarrow{\bullet}]$ *satisfies robustness under HRAA with endorsement.*

Proof. Suppose that for the well-typed program with fixed order of execution $S[\overrightarrow{a}]$, and states $\sigma_1 =_A \sigma_2$, we have $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$. We need to show that in all possible ordering and frequency of gateways calls, and for all relevant attacks a', the execution indistinguishability holds. Formally:

$$\forall S', a' \in \Omega_R(S[\overrightarrow{a}], S'[\overrightarrow{\bullet}], \sigma_1).\ N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$$

Using Lemma A.7, by induction on the structure of S':

$[\overrightarrow{a'}]; x := C.m(\overline{p})$**:** If $C.m(\overline{p})$ was a gateway method originally called in $S[\overrightarrow{\bullet}]$, $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ is going to be a direct result of Lemma A.7.

However, it is also possible that $C.m(\overline{p})$ is a gateway method, not originally called in $S[\overrightarrow{\bullet}]$. Because of the preprocessing step of G^D we can conclude that $C.m(\overline{p})$ does not have any *new* declassifications, but it is still possible for it to have some *new* endorsements. We consider two cases:

- If $C.m(\overline{p})$ did not had new endorsement, by initial public indistinguishability of states σ_1 and σ_2, and equality of attack vector and shared state on both sides, we can conclude that $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ holds.
- If it had any *new* endorsement, this new endorsed value is either from the shared state or attacker input. Nevertheless, shared state is equal in both sides because we do not have any reordering of gateways, attacker input is also equal because both sides use a'. These facts along with the public indistinguishability of

states σ_1 and σ_2, allows us to show that $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ also holds in this case.

$S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$: We have to consider two cases:

- $S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$ calls *no new gateways*: In this case different order and frequency of execution can change the shared state. We know that values of the shared state are untrusted, but they can be endorsed. However, because a' is a relevant attack w.r.t $S'[\overrightarrow{\bullet}]$ and $S[\overrightarrow{\bullet}]$, any endorsed shared state is going to have the same value in $S[\overrightarrow{\bullet}]$ and $S'[\overrightarrow{\bullet}]$. This allows us to conclude that even though the vales of shared state can be different between $S[\overrightarrow{\bullet}]$ and $S'[\overrightarrow{\bullet}]$, the shared state values that are endorsed and can affect declassifications are equal. The rest of the proof for this case is similar to Theorem A.2.

  By Rule T-Declassify and the preprocessing step of G^D we know that changing the order or frequency of gateway calls cannot lead to any *new* declassifications, and by the assumption $N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S[\overrightarrow{a}], \sigma_2\rangle$, and the fact that any of the gateway calls in $S'_1[\overrightarrow{a'_1}]$ and $S'_2[\overrightarrow{a'_2}]$ that had a declassification were also in $S[\overrightarrow{a}]$ (because G^D was a subset of it), we can conclude that declassified secrets had the same value in σ_1 and σ_2.

  The return value of gateway calls can depend on declassified secrets (which we shown are equal in σ_1 and σ_2), attacker input (which is the same $(\overrightarrow{a'})$ for both programs), public values (that are equal in both states), and the values in shared state. Of these values, only shared state can be different between $S[\overrightarrow{\bullet}]$ and $S'[\overrightarrow{\bullet}]$, resulting in different traces for $S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$ compared to any other order of calling these gateways. However, since the value of shared state is determined by the order of execution, we are going to get similar traces for each possible order of execution.

- $S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$ calls a new gateway not originally called in $S[\overrightarrow{\bullet}]$. We have to consider two cases:

  – They have *no* new endorsements: The return value of gateway call can depend on attacker input (which is the same $(\overrightarrow{a'})$ for both programs), public values (that are equal in both states), and the values in shared state which can be different between $S[\overrightarrow{\bullet}]$ and $S'[\overrightarrow{\bullet}]$. However, since the value of shared state is determined by the order of execution, we are going to get similar traces for each possible order of execution. Thus $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ holds.

  – $S'[\overrightarrow{\bullet}]$ leads to new endorsements: By G^D we know that these new endorsements cannot lead to any new declassifications. Additionally, these new endorsed values can either be from the shared state or attacker input. Shared state is equal in both sides because we have similar ordering of gateways in $S'_1[\overrightarrow{a'_1}]; S'_2[\overrightarrow{a'_2}]$, attacker input is also equal because both sides use a'. These facts along with the public indistinguishability of states σ_1 and σ_2, allows us to show that $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ also holds in this case.

Therefore, we can conclude that $N \vdash_\delta \langle S_1'[\overrightarrow{a_1'}]; S_2'[\overrightarrow{a_2'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S_1'[\overrightarrow{a_1'}]; S_2'[\overrightarrow{a_2'}], \sigma_2\rangle$ will always hold no matter the order of gateway method calls in $S_1'[\overrightarrow{a_1'}]$ and $S_2'[\overrightarrow{a_2'}]$.

Using these cases it is straightforward to conclude that for all S' and relevant attacks a' the indistinguishability of traces will be preserved $N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_1\rangle \approx_A N \vdash_\delta \langle S'[\overrightarrow{a'}], \sigma_2\rangle$ □

$$\textsc{Int}\quad \frac{}{\alpha \vdash_\delta \langle n, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle n, \mathcal{M}, \mathcal{H}\rangle} \qquad \textsc{Unit}\quad \frac{}{\alpha \vdash_\delta \langle unit, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle unit, \mathcal{M}, \mathcal{H}\rangle}$$

$$\textsc{Class}\quad \frac{}{\alpha \vdash_\delta \langle C, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle} \qquad \text{VAR}\quad \frac{\alpha = \delta(x) \qquad v = \mathcal{M}(x)}{\alpha \vdash_\delta \langle x, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle}$$

$$\textsc{Field Access}\quad \frac{\alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle \qquad \alpha = \delta(C) \qquad v = \mathcal{H}(C, f)}{\alpha \vdash_\delta \langle e.f, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H}\rangle}$$

$$\textsc{Op}\quad \frac{\alpha \vdash_\delta \langle e_1, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle n_1, \mathcal{M}, \mathcal{H}\rangle \qquad \alpha \vdash_\delta \langle e_2, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle n_2, \mathcal{M}, \mathcal{H}\rangle \qquad n' = n_1 \oplus n_2}{\alpha \vdash_\delta \langle e_1 \oplus e_2, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle n', \mathcal{M}, \mathcal{H}\rangle}$$

$$\textsc{Method}\quad \frac{\begin{array}{c} \alpha \vdash_\delta \langle e, \mathcal{M}, \mathcal{H}\rangle \Downarrow \langle C, \mathcal{M}, \mathcal{H}\rangle \qquad method_\phi(\overline{p})\{S; return(e)\} = getMethod(C, m) \\ \Big(\big(\alpha = \delta(C)\big) \vee \big(\alpha = N \wedge \delta(C) = E \wedge \phi = G\big)\Big) \qquad \mathcal{M}^* = \mathcal{M}[p_i \mapsto \sigma(q_i)] \quad i = 1, ..., |\overline{p}| \\ \delta(C) \vdash_\delta \langle S; return(e), \mathcal{M}^*, \mathcal{H}\rangle \Downarrow \mathcal{M}^{*'}, \mathcal{H}' \triangleright v \qquad \mathcal{M}' = \mathcal{M}^{*'} \setminus [p_i] \quad i = 1, ..., |\overline{p}| \\ (\alpha = \delta(C) \Rightarrow \beta = \epsilon) \qquad (\alpha = N \wedge \delta(C) = E \wedge \phi = G \Rightarrow \beta = v) \end{array}}{\alpha \vdash_\delta \langle e.m(\overline{q}), \mathcal{M}, \mathcal{H}\rangle \Downarrow_\beta \langle v, \mathcal{M}', \mathcal{H}'\rangle}$$

Figure A.7: Large-step semantics for J_E expressions

$$\textsc{T-Unit}\quad \frac{}{\Gamma, \Pi \vdash_\delta \mathit{unit} : (\langle \mathbb{P}, \mathbb{T} \rangle, F)} \qquad \textsc{T-Int}\quad \frac{}{\Gamma, \Pi \vdash_\delta n : (\langle \mathbb{P}, \mathbb{T} \rangle, F)}$$

$$\textsc{Sub Typing}\quad \frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad \ell \sqsubseteq \ell' \qquad d \sqsubseteq d'}{\Gamma, \Pi \vdash_\delta e : (\ell', d')} \qquad \textsc{T-Class}\quad \frac{\Gamma(C) = \ell \qquad \Pi(C) = d}{\Gamma, \Pi \vdash_\delta C : (\ell, F)}$$

$$\textsc{T-Var}\quad \frac{\Gamma(x) = \ell \qquad \Pi(x) = d}{\Gamma, \Pi \vdash_\delta x : (\ell, d)} \qquad \textsc{T-Field Access}\quad \frac{\Gamma(C.f) = \ell \qquad \Pi(C.f) = d}{\Gamma, \Pi \vdash_\delta C.f : (\ell, d)}$$

$$\textsc{T-Op}\quad \frac{\Gamma, \Pi \vdash_\delta e_1 : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta e_2 : (\ell_2, d_2)}{\Gamma, \Pi \vdash_\delta e_1 \oplus e_2 : (\ell_1 \sqcup \ell_2, d_1 \vee d_2)}$$

$$\textsc{T-Method}\quad \frac{\mathrm{method}_\phi(\overline{p})\{S; \mathtt{return}(e)\} = \mathrm{getMethod}(C, m) \qquad pc', \Gamma^-, \Pi^- \vdash_\delta S; \mathtt{return}(e) : \Gamma^+, \Pi^+ \triangleright (\ell, d)}{\Gamma, \Pi \vdash_\delta C.m(\overline{p}) : \left(\Gamma^-, \Pi^- \underset{pc'}{\overline{\qquad}} \Gamma^+, \Pi^+\right)_{(\ell, d)}}$$

Figure A.8: Typing rules for expressions

Skip

$$\frac{}{\alpha \vdash_{\delta} \langle \texttt{skip}, \mathcal{M}, \mathcal{H} \rangle \Downarrow \mathcal{M}, \mathcal{H}}$$

Return

$$\frac{\alpha \vdash_{\delta} \langle e, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H} \rangle}{\alpha \vdash_{\delta} \langle \texttt{return}(e), \mathcal{M}, \mathcal{H} \rangle \Downarrow \mathcal{M}, \mathcal{H} \triangleright v}$$

Assign

$$\frac{\alpha \vdash_{\delta} \langle e_G, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{\beta} \langle v, \mathcal{M}', \mathcal{H}' \rangle \qquad \alpha = \delta(x) \qquad \mathcal{M}'' = \mathcal{M}'[x \mapsto v]}{\alpha \vdash_{\delta} \langle x := e_G, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{\beta} \mathcal{M}'', \mathcal{H}'}$$

Store

$$\frac{\alpha \vdash_{\delta} \langle e_G, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{\beta} \langle v, \mathcal{M}', \mathcal{H}' \rangle \qquad \alpha = \delta(C) \qquad \mathcal{H}'' = \mathcal{H}'[(C, f) \mapsto v]}{\alpha \vdash_{\delta} \langle C.f := e_G, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{\beta} \mathcal{M}', \mathcal{H}''}$$

Seq

$$\frac{\alpha \vdash_{\delta} \langle S_1, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t_1} \mathcal{M}_1, \mathcal{H}_1 \qquad \alpha \vdash_{\delta} \langle S_2, \mathcal{M}_1, \mathcal{H}_1 \rangle \Downarrow_{t_2} \mathcal{M}_2, \mathcal{H}_2}{\alpha \vdash_{\delta} \langle S_1; S_2, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t_1.t_2} \mathcal{M}_2, \mathcal{H}_2}$$

Seq-return

$$\frac{\alpha \vdash_{\delta} \langle S, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t} \mathcal{M}_1, \mathcal{H}_1 \qquad \alpha \vdash_{\delta} \langle \texttt{return}(e), \mathcal{M}_1, \mathcal{H}_1 \rangle \Downarrow \mathcal{M}_1, \mathcal{H}_1 \triangleright v}{\alpha \vdash_{\delta} \langle S; \texttt{return}(e), \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t} \mathcal{M}_1, \mathcal{H}_1 \triangleright v}$$

If-Else-T

$$\frac{\alpha \vdash_{\delta} \langle e, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle n, \mathcal{M}, \mathcal{H} \rangle \quad n \neq 0 \qquad \alpha \vdash_{\delta} \langle S_1, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t} \mathcal{M}', \mathcal{H}'}{\alpha \vdash_{\delta} \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t} \mathcal{M}', \mathcal{H}'}$$

If-Else-F

$$\frac{\alpha \vdash_{\delta} \langle e, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle 0, \mathcal{M}, \mathcal{H} \rangle \qquad \alpha \vdash_{\delta} \langle S_2, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t} \mathcal{M}', \mathcal{H}'}{\alpha \vdash_{\delta} \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t} \mathcal{M}', \mathcal{H}'}$$

While-F

$$\frac{\alpha \vdash_{\delta} \langle e, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle 0, \mathcal{M}, \mathcal{H} \rangle}{\alpha \vdash_{\delta} \langle \texttt{while}\ e\ \texttt{then}\ S, \mathcal{M}, \mathcal{H} \rangle \Downarrow \mathcal{M}, \mathcal{H}}$$

While-T

$$\frac{\begin{array}{c} \alpha \vdash_{\delta} \langle e, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle n, \mathcal{M}, \mathcal{H} \rangle \\ n \neq 0 \qquad \alpha \vdash_{\delta} \langle S, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t_1} \mathcal{M}', \mathcal{H}' \qquad \alpha \vdash_{\delta} \langle \texttt{while}\ e\ \texttt{then}\ S, \mathcal{M}', \mathcal{H}' \rangle \Downarrow_{t_2} \mathcal{M}^{''}, \mathcal{H}^{''} \end{array}}{\alpha \vdash_{\delta} \langle \texttt{while}\ e\ \texttt{then}\ S, \mathcal{M}, \mathcal{H} \rangle \Downarrow_{t_1.t_2} \mathcal{M}^{''}, \mathcal{H}^{''}}$$

Declassify

$$\frac{E \vdash_{\delta} \langle e, \mathcal{M}, \mathcal{H} \rangle \Downarrow \langle v, \mathcal{M}, \mathcal{H} \rangle \qquad \delta(x) = E \qquad \mathcal{M}' = \mathcal{M}[x \mapsto v]}{E \vdash_{\delta} \langle x := \texttt{declassify}(e), \mathcal{M}, \mathcal{H} \rangle \Downarrow \mathcal{M}', \mathcal{H}}$$

Figure A.9: Large-step semantics of J_E commands

T-Skip

$$\frac{}{pc, \Gamma, \Pi \vdash_\delta \texttt{skip} : \Gamma, \Pi}$$

T-Sub-Typing

$$\frac{pc_1, \Gamma_1, \Pi_1 \vdash_\delta S : \Gamma_1', \Pi_1' \qquad pc_1 \sqsubseteq pc_2 \qquad \Gamma_1 \sqsubseteq \Gamma_2 \qquad \Pi_1 \sqsubseteq \Pi_2}{pc_2, \Gamma_2, \Pi_2 \vdash_\delta S : \Gamma_2', \Pi_2'}$$

T-Assign

$$\frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad \ell \sqcup pc_\ell = \langle \mathbb{S}, - \rangle \Rightarrow \delta(x) = E \qquad d = T \Rightarrow pc_\ell = \langle -, \mathbb{T} \rangle}{pc, \Gamma, \Pi \vdash_\delta x := e : \Gamma\Big[x \mapsto pc_\ell \sqcup \ell\Big], \Pi\Big[x \mapsto d \vee pc_d\Big]}$$

T-Store

$$\frac{\begin{array}{c} \Gamma, \Pi \vdash_\delta C.f : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_\delta e : (\ell_2, d_2) \qquad \ell_2 \sqcup pc_\ell \sqsubseteq \ell_1 \\ \ell_1 \sqcup \ell_2 \sqcup pc_\ell = \langle \mathbb{S}, - \rangle \Rightarrow \delta(C) = E \qquad pc_d \vee d_2 = T \Rightarrow pc_\ell \sqcup \ell_1 = \langle -, \mathbb{T} \rangle \qquad d' = d_2 \vee pc_d \end{array}}{pc, \Gamma, \Pi \vdash_\delta C.f := e : \Gamma, \Pi\Big[C.f \mapsto d', C \mapsto \Pi(C) \vee d'\Big]}$$

T-Seq

$$\frac{pc, \Gamma, \Pi \vdash_\delta S_1 : \Gamma', \Pi' \qquad pc, \Gamma', \Pi' \vdash_\delta S_2 : \Gamma'', \Pi''}{pc, \Gamma, \Pi \vdash_\delta S_1; S_2 : \Gamma'', \Pi''}$$

T-Seq-Return

$$\frac{pc, \Gamma, \Pi \vdash_\delta S : \Gamma', \Pi' \qquad pc, \Gamma', \Pi' \vdash_\delta \texttt{return}(e) : \Gamma', \Pi' \rhd (\ell, d)}{pc, \Gamma, \Pi \vdash_\delta S; \texttt{return}(e) : \Gamma', \Pi' \rhd (\ell, d)}$$

T-If-Else

$$\frac{\begin{array}{c} \Gamma, \Pi \vdash_\delta e : (\ell, d) \\ (pc_\ell \sqcup \ell, pc_d \vee d), \Gamma, \Pi \vdash_\delta S_1 : \Gamma', \Pi' \qquad (pc_\ell \sqcup \ell, pc_d \vee d), \Gamma, \Pi \vdash_\delta S_2 : \Gamma', \Pi' \end{array}}{pc, \Gamma, \Pi \vdash_\delta \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 : \Gamma', \Pi'}$$

T-While

$$\frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad (pc_\ell \sqcup \ell, pc_d \vee d), \Gamma, \Pi \vdash_\delta S : \Gamma', \Pi'}{pc, \Gamma, \Pi \vdash_\delta \texttt{while } e \texttt{ then } S : \Gamma', \Pi'}$$

T-Declassify

$$\frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad \ell \sqsubseteq \langle \mathbb{S}, \mathbb{T} \rangle \qquad pc_\ell \sqsubseteq \langle \mathbb{P}, \mathbb{T} \rangle \qquad \delta(x) = E}{pc, \Gamma, \Pi \vdash_\delta x := \texttt{declassify}(e) : \Gamma\Big[x \mapsto \ell \sqcap \langle \mathbb{P}, \mathbb{T} \rangle\Big], \Pi\Big[x \mapsto T\Big]}$$

T-Return

$$\frac{\Gamma, \Pi \vdash_\delta e : (\ell, d) \qquad d = T \Rightarrow pc_\ell = \langle -, \mathbb{T} \rangle}{pc, \Gamma, \Pi \vdash_\delta \texttt{return}(e) : \Gamma, \Pi \rhd (pc_\ell \sqcup \ell, d)}$$

Figure A.10: Typing rules for commands

T-Assign-Call

$$\frac{\begin{array}{c}\Gamma, \Pi \vdash_{\delta} C.m(\overline{p}) : \left(\Gamma^{-}, \Pi^{-} \underset{pc'}{\overline{}} \Gamma^{+}, \Pi^{+}\right)_{(\ell, d)} \qquad pc_{\ell} \sqsubseteq pc'_{\ell} \qquad \delta(x) = \delta(C) \\ d = T \Rightarrow pc_{\ell} = \langle -, \mathbb{T} \rangle \qquad \Gamma(q_i) \sqsubseteq \Gamma^{-}(p_i) \;\; i = 1...|\overline{p}| \qquad \Pi(q_i) = \Pi^{-}(p_i) \;\; i = 1...|\overline{p}| \\ \forall y \in \mathrm{dom}(\Gamma^{-}).\Gamma(y) \sqsubseteq \Gamma^{-}(y) \qquad \forall y \in \mathrm{dom}(\Gamma^{+}).\Gamma^{+}(y) \sqsubseteq \Gamma_{out}(y) \\ \forall y \in (\mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^{+}).\Gamma(y) = \Gamma_{out}(y) \qquad \forall y \in \mathrm{dom}(\Pi^{-}).\Pi(y) \sqsubseteq \Pi^{-}(y) \\ \forall y \in \mathrm{dom}(\Pi^{+}).\Pi^{+}(y) \sqsubseteq \Pi_{out}(y) \qquad \forall y \in (\mathrm{dom}(\Pi) \setminus \mathrm{dom}(\Pi^{+}).\Pi(y) = \Pi_{out}(y)\end{array}}{pc, \Gamma, \Pi \vdash_{\delta} x := C.m(\overline{q}) : \Gamma_{out}\Big[x \mapsto pc_{\ell} \sqcup \ell\Big], \Pi_{out}\Big[x \mapsto d \vee pc_{d}\Big]}$$

T-Assign-Gateway-Call

$$\frac{\begin{array}{c}\Gamma, \Pi \vdash_{\delta} C.m(\overline{p}) : \left(\Gamma^{-}, \Pi^{-} \underset{pc'}{\overline{}} \Gamma^{+}, \Pi^{+}\right)_{(\ell, d)} \\ \delta(x) = N \qquad \delta(C) = E \qquad \ell \sqsubseteq \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Gamma^{-}(p) = \langle \mathbb{P}, \mathbb{U} \rangle \\ \forall p \in \overline{p}.\Pi^{-}(p) = F \qquad \Gamma(q_i) \sqsubseteq \Gamma^{-}(p_i) \;\; i = 1...|\overline{p}| \qquad \Pi(q_i) = \Pi^{-}(p_i) \;\; i = 1...|\overline{p}| \\ \forall y \in \mathrm{dom}(\Gamma^{-}).\Gamma(y) \sqsubseteq \Gamma^{-}(y) \qquad \forall y \in \mathrm{dom}(\Gamma^{+}).\Gamma^{+}(y) \sqsubseteq \Gamma_{out}(y) \\ \forall y \in (\mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^{+}).\Gamma(y) = \Gamma_{out}(y) \qquad \forall y \in \mathrm{dom}(\Pi^{-}).\Pi(y) \sqsubseteq \Pi^{-}(y) \\ \forall y \in \mathrm{dom}(\Pi^{+}).\Pi^{+}(y) \sqsubseteq \Pi_{out}(y) \qquad \forall y \in (\mathrm{dom}(\Pi) \setminus \mathrm{dom}(\Pi^{+}).\Pi(y) = \Pi_{out}(y)\end{array}}{pc, \Gamma, \Pi \vdash_{\delta} x := C.m(\overline{q}) : \Gamma_{out}\Big[x \mapsto pc_{\ell} \sqcup \ell\Big], \Pi_{out}\Big[x \mapsto F\Big]}$$

T-Store-Call

$$\frac{\begin{array}{c}\Gamma, \Pi \vdash_{\delta} C.f : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_{\delta} C'.m(\overline{p}) : \left(\Gamma^{-}, \Pi^{-} \underset{pc'}{\overline{}} \Gamma^{+}, \Pi^{+}\right)_{(\ell_2, d_2)} \\ \delta(C) = \delta(C') \qquad d_2 = T \Rightarrow pc_{\ell} = \langle -, \mathbb{T} \rangle \qquad \ell_1 \sqcup \ell_2 \sqcup pc_{\ell} = \langle \mathbb{S}, - \rangle \Rightarrow \delta(C) = E \\ pc_{\ell} \sqsubseteq pc'_{\ell} \qquad \ell_2 \sqcup pc_{\ell} \sqsubseteq \ell_1 \qquad \Gamma(q_i) \sqsubseteq \Gamma^{-}(p_i) \;\; i = 1...|\overline{p}| \qquad \Pi(q_i) = \Pi^{-}(p_i) \;\; i = 1...|\overline{p}| \\ \forall y \in \mathrm{dom}(\Gamma^{-}).\Gamma(y) \sqsubseteq \Gamma^{-}(y) \qquad \forall y \in \mathrm{dom}(\Gamma^{+}).\Gamma^{+}(y) \sqsubseteq \Gamma_{out}(y) \\ \forall y \in (\mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^{+}).\Gamma(y) = \Gamma_{out}(y) \qquad \forall y \in \mathrm{dom}(\Pi^{-}).\Pi(y) \sqsubseteq \Pi^{-}(y) \\ \forall y \in \mathrm{dom}(\Pi^{+}).\Pi^{+}(y) \sqsubseteq \Pi_{out}(y) \qquad \forall y \in (\mathrm{dom}(\Pi) \setminus \mathrm{dom}(\Pi^{+}).\Pi(y) = \Pi_{out}(y)\end{array}}{pc, \Gamma, \Pi \vdash_{\delta} C.f := C'.m(\overline{q}) : \Gamma_{out}, \Pi_{out}\Big[C.f \mapsto d_2 \vee pc_{d}\Big]}$$

T-Store-Gateway-Call

$$\frac{\begin{array}{c}\Gamma, \Pi \vdash_{\delta} C.f : (\ell_1, d_1) \qquad \Gamma, \Pi \vdash_{\delta} C'.m(\overline{p}) : \left(\Gamma^{-}, \Pi^{-} \underset{pc'}{\overline{}} \Gamma^{+}, \Pi^{+}\right)_{(\ell_2, d_2)} \qquad \delta(C) = N \\ \delta(C') = E \qquad \ell_2 \sqsubseteq \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Gamma^{-}(p) = \langle \mathbb{P}, \mathbb{U} \rangle \qquad \forall p \in \overline{p}.\Pi^{-}(p) = F \\ \ell_2 \sqcup pc_{\ell} \sqsubseteq \ell_1 \qquad \Gamma(q_i) \sqsubseteq \Gamma^{-}(p_i) \;\; i = 1...|\overline{p}| \qquad \Pi(q_i) = \Pi^{-}(p_i) \;\; i = 1...|\overline{p}| \\ \forall y \in \mathrm{dom}(\Gamma^{-}).\Gamma(y) \sqsubseteq \Gamma^{-}(y) \qquad \forall y \in \mathrm{dom}(\Gamma^{+}).\Gamma^{+}(y) \sqsubseteq \Gamma_{out}(y) \\ \forall y \in (\mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma^{+}).\Gamma(y) = \Gamma_{out}(y) \qquad \forall y \in \mathrm{dom}(\Pi^{-}).\Pi(y) \sqsubseteq \Pi^{-}(y) \\ \forall y \in \mathrm{dom}(\Pi^{+}).\Pi^{+}(y) \sqsubseteq \Pi_{out}(y) \qquad \forall y \in (\mathrm{dom}(\Pi) \setminus \mathrm{dom}(\Pi^{+}).\Pi(y) = \Pi_{out}(y)\end{array}}{pc, \Gamma, \Pi \vdash_{\delta} C.f := C'.m(\overline{q}) : \Gamma_{out}, \Pi_{out}\Big[C.f \mapsto F\Big]}$$

Figure A.10: Typing rules for commands

Paper B

Enclave-Based Secure Programming with $\mathbf{J_E}$

Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi

Abstract

Over the past few years, major hardware vendors have started offering processors that support Trusted Execution Environments (TEEs) allowing confidential computations over sensitive data on untrusted hosts. Unfortunately, developing applications that use TEEs remains challenging. Current solutions require using low-level languages (e.g., C/C++) to handle the TEE management process manually – a complex and error-prone task. Worse, the separation of the application into components that run inside and outside the TEE may lead to information leaks. In summary, TEEs are a powerful means to design secure applications, but there is still a long way to building secure software with TEEs alone.

In this work, we present J_E, a programming model for developing TEE-enabled applications where developers only need to annotate Java programs to define application-level security policies and run them securely inside enclaves.

B.1 Introduction

In cloud computing, cloud service providers offer their infrastructure as a service, and clients use it on an ad-hoc basis. This approach ensures on-demand computing and storage provisioning, but it comes at the price of trusting the cloud providers with potentially sensitive data. Nevertheless, the cloud computing paradigm entails many security and privacy concerns as data is inevitably processed on third-party machines. For example, the cloud could be compromised, but also, the cloud infrastructure may not have strict access control policies to rule out unauthorized access of data. Traditional privacy-preserving techniques struggle to mitigate such issues. For example, symmetric and asymmetric cryptography require encrypted data to be first decrypted to perform any computations – making plaintext data accessible to the hosting infrastructure. On the other hand, homomorphic encryption schemes [150] allow performing computations directly on the encrypted data, but their high computation time and large ciphertext size can severely affect the application's performance.

Hardware-based Trusted Execution Environments (TEEs) are hardware enclaves that protect data and code from the system software. A number of hardware vendors have introduced TEE technologies including Intel with Software Guard Extensions (SGX) [101, 108], ARM with TrustZone [60], MultiZone [175] and others [125, 129, 159, 168, 172]. In TEEs, data can be processed at native speed ensuring that it remains protected even on a third-party machine without having to encrypt it – expensive homomorphic encryption can be avoided to yield better performance.

Despite TEE implementations have been used in a number of industry products [197, 198], programming software that takes advantage of TEE functionalities remains challenging.

Figure B.1 shows the implementation of a simple password checker using the C/C++ interface for the Intel SGX enclave (in Microsoft Visual Studio with SGX add-on). With the current approach, programmers need to deal with the low-level details of enclave programming, e.g. partitioning the code into separate files that define the program running outside the enclave (main.cpp) and the program running inside the enclave (enclave.cpp), defining a separate interface between the environments with the semantics of parameter passing (enclave.edl), and setting up the enclave creation (main.cpp, lines 5 to 9) and its disposal after use (main.cpp, line 14).

Though the enclave environment is protected by the hardware, an attacker controlling the non-enclave environment can initiate various attacks on the sensitive data residing inside the enclave, thus compromising the overall application security. In Figure B.1, an attacker that controls the non-enclave environment can manipulate the parameters passed to the `checkPassword()` call to the enclave code. In such case, the compiler would not alert the programmer to report a potential security issue.

This leads us to the following key research questions: (a) How to enable seamless integration of enclaves and managed languages like Java? (b) How to check the security of enclave programs with respect to realistic enclave attackers?

In summary, this paper makes the following contributions:

- We present J_E, a language design to seamlessly support enclave programming.
- We describe the implementation of J_E and evaluate its applicability by presenting different case studies.

```
enclave {
    trusted {
        public void checkPassword([in, size=len] char* guess, [out]
            int* result, size_t len);
    };
};
```

Program B.1: enclave.edl

```
const char* password = "secret";
void checkPassword(char* guess, int* result, size_t len) {
    strcmp(guess, password) == 0) ?
        *result = 1 : *result = 0;
}
```

Program B.2: enclave.cpp

```
#include "sgx_urts.h"
#include "enclave_u.h"
#define BUF_LEN 100
int main() {
    sgx_enclave_id_t eid;
    sgx_status_t ret = SGX_SUCCESS;
    sgx_launch_token_t token = {0};
    int updated = 0;
    ret = sgx_create_enclave(ENCLAVE_FILE, SGX_DEBUG_FLAG, &token, &updated, &eid, NULL);
    if (ret != SGX_SUCCESS) { ... /* exception */ }
    char* guess = ... // read guess from stdin
    int result = 0;
    checkPassword(eid, guess, &result, BUF_LEN);
    if (SGX_SUCCESS != sgx_destroy_enclave(eid)) {...}
    return 0;
}
```

Program B.3: main.cpp

Figure B.1: Password checker, C++

B.2 J_E Design

The goal of the J_E design is twofold. (i) The design should abstract away the TEE management details allowing the programmer to easily specify the parts of the program that must run inside the TEE. (ii) The design should provide simple means to specify and enforce security policies for an application. To this end, we provide a set of security annotations and functions. The J_E compiler leverages these annotations to automatically partition the application and generate the logic for the enclave management (creation, initialization, communication). The J_E compiler uses the security annotations and functions to verify information flow policies via a security type system.

We illustrate the J_E features using the password checker routine provided in Figure B.2. In J_E, a class can be annotated with the `@Enclave` annotation (dubbed *enclave class*). Both code and data of enclave classes are stored inside the enclave. To ensure that data and computations concerning encryption take place within the enclave, the `Password Checker` class in Figure B.2 is annotated with the `@Enclave` annotation. Within an enclave class, the `@Secret` annotation identifies *secret* fields. The portions of a program influenced by a secret are also considered secret to prevent flows of sensitive data that may leak outside the enclave. The `password` field (line 3) is annotated with the `@Secret` annotation to denote that its value should not be leaked to the non-enclave environment. The static methods of enclave classes annotated with the `@Gateway` annotation (*gateway* methods) act as the interface between the enclave and the non-enclave environments. The `checkPassword` method (line 6) is annotated with the `@Gateway` annotation. The `checkPassword` method accepts a string from the non-enclave environment and compares it with the `password` field, the result of the comparison is returned to the non-enclave environment as a boolean value. The return value of the gateway methods must not be influenced by secret information.

In addition to annotations, we introduce two operators. The `declassify` is a unary operator to downgrade a secret value into a public one to release sensitive information. The result of the equality comparison of `password` and `guess` is stored in the `result` field (line 8). Since the `result` field is influenced by the `password` secret field, it is also considered as a sensitive. We apply the `declassify` operator to the `result` variable (line 9) to ensure that `result` can be released to the non-enclave environment. The `declassify` operator can only declassify the trusted values. The operator `endorse` endorses an untrusted value into a trusted one. The arguments of gateway methods come from the non-enclave environment and are considered untrusted by default. We apply the `endorse` operator to the `guess` argument (line 7). The trusted value is stored in the variable `guessE`. Hence `result` variable is not influenced by any untrusted value and is declassified successfully (line 9).

B.3 Attacker Model and Enforcement

In this section we discuss the attacker models considered in J_E, and provide an overview of the security type system used in J_E to enforce security against these attackers.

```
@Enclave
class PasswordChecker {
    @Secret static String password = ...;

    @Gateway
    public static boolean checkPassword(String guess) {
        String guessE = endorse(guess);
        boolean result = guessE.equals(password);
        return declassify(result);
    }
}
```

Program B.4: PasswordChecker.java

```
class Main {
    public static void main(String[] args) {
        String guess = ... // read guess from stdin
        PasswordChecker.checkPassword(guess);
    }
}
```

Program B.5: Main.java

Figure B.2: Password checker, J_E

Attacker Model

We assume that the application has two parts – one running inside and the other running outside the enclave. The attacker controls the non-enclave environment by: (1) controlling the non-enclave *data* memory, or (2) controlling the non-enclave *code and data* memory. These attacker capabilities induce two attacker models of interest.

Program B.6 illustrates the attacker models. The program stores a list of secret integers called `secretData`, and provides methods to access single elements of `secretData` and to release the average of these secret integers whenever the trigger `genAvg` is set. In the traditional setting without enclaves, where we trust everything in the system, this program is secure, since the secret values are written to the public variables of the main method only after declassification.

Now, consider a scenario where we need to run this code on an untrusted system. The traditional security assumptions are no longer sufficient, because the attacker can now access the system and learn the `secretData` by simply inspecting the memory.

One way to protect this data on an untrusted system is to use enclaves, thus relying on the hardware features to prevent the attacker from inspecting the enclave memory, and thus, protect the `secretData`. The naive way of achieving this would be to partition Program B.6 into secret and public parts, and put the `secretData` and all the methods that interact with it in a separate class `Storage` (Program B.7), and put it inside the enclave. The main (public) part of the program remains outside of the enclave (Program B.8).

```java
class Main {
    static int[] secretData;
    static boolean genAvg = false;

    public static void main(String[] args) {
        int data1 = getData(1);
        // ...
        releaseAvg();
        float avg = getAverage();
    }

    public static int getData(int input) {
        return declassify(secretData[input]);
    }

    public static void releaseAvg() {genAvg = true;}

    public static float getAverage() {
        if (genAvg) {
            float avg = doAverage(secretData);
            return declassify(avg);
        }
            else { return 0.0f; }
    }
}
```

Program B.6: Before partitioning

However, this naive partitioning is not enough to protect the `secretData` stored inside the enclave against different types of attacks from the non-enclave environment. In this work, we investigate two types of attackers that can exploit the enclave—non-enclave interface to learn the secrets stored inside of the enclave.

The first attacker controls the data memory outside of the enclave, hence they can manipulate the parameters passed to `getData` method, and learn all of the elements of `secretData` one by one. The second attacker controls both the data and code memory, hence they can change the order of method calls, e.g. call `Storage.releaseAvg()` in any order, and thus control the release of value `avg`. The enforcement mechanisms implemented in J_E enforce security against these types of active attackers and ensure that enclave programs do not leak secret data.

Type System

J_E uses security labels to specify application-level policies. The security labels are not part of the language but are inferred automatically by J_E. A security label is a 2-tuple consisting a confidentiality and an integrity label. We consider two labels *Public* and *Secret* for confidentiality, and two labels *Trusted* and *Untrusted* for integrity. Security labels form a standard (product) security lattice [4] and the order relation among the labels determines the allowed information flows for confidentiality and integrity.

```
// inside of enclave
class Storage {
    static int[] secretData;
    static boolean genAvg = false;

    // gateway
    public static int getData(int input) {
        return declassify(secretData[input]);
    }

    // gateway
    public static void releaseAvg() {genAvg = true;}

    // gateway
    public static float getAverage() {
        if (genAvg) {
            float avg = doAverage(secretData);
            return declassify(avg);
        }
        else { return 0.0f };
    }
}
```

Program B.7: Inside enclave

```
// outside of enclave
class Main {
    public static void main(String[] args) {
        int data1 = Storage.getData(1);
        // ...
        Storage.releaseAvg();
        float avg = Storage.getAverage();
    }
}
```

Program B.8: Outside enclave

The security type system tracks the implicit and explicit flows of information within the program by checking the security labels at each command, and propagating the security labels accordingly.

The programmer should explicitly specify the data inside the enclave that is considered secret. A secret field is labeled with a *Secret* and *Trusted* security label (2-tuple) as it contains sensitive information originating from inside an enclave class and hence, it is considered not tampered with by an attacker. The rules of the type system prevent storing secret data outside the enclave, prohibit information flow of enclave's secret data to non-enclave environment (unless secret information is intentionally declassified by the programmer in a secure manner), and ensure that gateway methods can only return values having the *Public* confidentiality level.

The type system prevents classes inside of the enclave to call into classes outside of the enclave. This is to control the flow of information and ensure that the only way for passing data to the non-enclave environment is through the return values of gateways.

The type system is mainly standard, but adds some extra safeguards to ensure security against active attackers. To enforce security against *data memory attacker*, we have to make sure that manipulating the parameters of gateway methods does not leak secret data. To achieve this, the type system assigns *Public* and *Untrusted* security label to the data coming from the non-enclave environment, and checks that the declassification of secret data is not influenced by untrusted values, thus ensuring that only the developer controls the decision to release secret data and not the active attacker.

The *data and code memory attacker* is more powerful than the *data memory attacker*. In order to enforce security against this attacker, we have to make sure that changing the order and frequency of calling gateways, or even calling new gateways, does not leak secret data (i.e. it does not lead to declassifying new secrets). To this end, the type system generates a list of all the gateways that declassify secret values and makes sure that all of these gateways are called in all possible executions of the program. This approach ensures that no new declassifying gateways can be called by the active attacker unless it has already been called in some way by the developer. Additionally, to prevent data leaks through changing the order and frequency of gateway calls, the type system marks all of the variables and fields shared between gateways as *Untrusted*. Similar to the parameters of gateway methods, these untrusted values cannot influence declassifications. The formal details of J_E's security type system are presented in [186].

B.4 Code Compilation and Implementation

The J_E compilation process involves multiple steps. Programs B.9 to B.15 shows how a J_E program (Program B.9) is partitioned (Program B.10 and B.11), translated to Jif to check security (Program B.12 and B.13), and augmented with RMI communication (Program B.14 and B.15). We now describe these individual compilation steps followed by the implementation details.

Code Partitioning

A J_E program is first analyzed and, based on the annotations, it is split into two partitions – the enclave and the non-enclave partition. All the classes annotated with the `@Enclave` annotation and all their required dependencies belong to the enclave partition. All the remaining classes belong to the non-enclave partition. Program B.9 shows a complete J_E program that encrypts string data using the secret `key` field.

The complete program includes a `Main` class (Line 1) and an `Encrypter` class (Line 8). The `Encrypter` class is annotated with the `@Enclave` annotation hence it belongs to the enclave partition. Program B.11 and B.10 show the partitioned J_E programs.

In this phase, the J_E compiler also performs some correctness checks and collects information required for conversion into an equivalent Jif program (see next section).

```java
class Main {
    public static void main(String[] args) {
        String plaintext = "message";
        String cipher = Encrypter.encrypt(plaintext);
}   }

@Enclave
class Encrypter {
    @Secret private static String key;

    @Gateway
    public static String encrypt(String plaintext) {
        String palintextE = endorse plaintext;
        String cipher = encode(plaintext, key);
        return declassify cipher;
}   }
```

Program B.9: J_E code

```java
class Main {
    public static void main(String[] args) {
        String plaintext = "message";
        String cipher = Encrypter.encrypt(plaintext);
    }
}
```

Program B.10: Partition outside the enclave

```java
@Enclave
final class Encrypter {
    @Secret private static String key;

    @Gateway
    public static String encrypt(String plaintext) {
        String plaintextE = endorse plaintext;
        String cipher = encode(plaintext, key);
        return declassify cipher;
    }
}
```

Program B.11: Partition inside the enclave

Conversion to Jif

Next, the partition to run inside the enclave is converted into an equivalent Jif [19] program. Jif extends Java with security labels to statically enforce information flow control. A Jif security label is a pair consisting of a confidentiality level and an integrity level. In the example, Program B.13 is the equivalent Jif Program B.11. The non-enclave partition remains unchanged (Program B.10 and Program B.12). Conversion of the J_E program into Jif involves the following steps. (1) J_E secret fields are converted into Jif fields with `{Enclave->*;Enclave<-*}` label (Program B.11, Line 3 and Program B.13, Line 2). Such label represents values that are *secret* and *trusted*. (2) For gateway methods, the arguments and the return value are labeled with the Jif's `{}` and `{Enclave->_;Enclave<-*}` labels respectively. The `{}` label denotes *public* and *untrusted* information while the `{Enclave->_;Enclave<-*}` label represents the *public* and *trusted* information (Program B.13, Line 5). (3) The `declassify` operator in J_E corresponds to the `declassify` operator in Jif (Program B.13, Line 8).

```
class Main {
    public static void main(String[] args) {
        String plaintext = "message";
        String cipher = Encrypter.encrypt(plaintext);
    }
}
```

Program B.12: Jif code outside the enclave

```
 1  final class Encrypter [principal Enclave] authority(Enclave) {
 2      private {Enclave->*; Enclave<-*} key;
 3
 4      public String{Enclave->_; Enclave<-*} encrypt{Enclave<-*}(String{} plaintext)
 5                                                       where authority(Enclave) {
 6          String plaintextE = endorse(plaintext, {} to {Enclave<-*});
 7          String{Enclave->*; Enclave<-*} cipher = encode(plaintext, key);
 8          return declassify (cipher, {Enclave->*; Enclave<-*} to {Enclave->_; Enclave<-*});
 9      }
10  }
```

Program B.13: Jif code inside the enclave

The obtained Jif program is compiled using the Jif compiler to ensure proper label propagation and checking.

Remote Communication

The next step introduces the code required for the communication via Java RMI [203] between the enclave and the non-enclave partition. As in RMI, remote objects are reachable only through an interface, for each class annotated with the `@Enclave` annotation, the J_E compiler generates a remote interface containing all the gateway methods. Next, the J_E compiler creates a wrapper class implementing the interface above for each enclave class. This way, all the gateway methods of an enclave class are exposed remotely to the non-enclave environment through the remote interface. Finally, the method calls to the enclave class from the non-enclave environment are replaced with an RMI lookup that returns a remote reference to the interface of the wrapper class.

```java
1  class Main{
2      public static void main(String[] args) {
3          String plaintext = "message";
4
5          Remote remoteServer = Naming.lookup("rmi://IPAddr/RemoteEncr");
6          RemoteEncr remSrvStub = (RemoteEncr) remoteServer;
7          String cipher = remSrvStub.wrapEncrypt(plaintext);
8      }
9  }
```

Program B.14: Non-enclave partition – communication

```java
1  interface RemoteEncr extends Remote {
2      public String wrapEncrypt(String plaintext);
3  }
4  class EncrypterWrapper extends UnicastRemoteObject implements RemoteEncr {
5
6      @Override
7      public String wrapEncrypt(String plaintext) {
8          return Encrypter.encrypt(plaintext);
9  }   }
10 final class Encrypter { ... }
```

Program B.15: Enclave partition – communication

Note that, we prohibit remote calls from the enclave to the non-enclave environment. For example, in Program B.15, the J_E compiler generates the `RemoteEncr` interface (Line 1) and the `EncrypterWrapper` class, which acts as a wrapper for the class `Encrypter`, and implements the `RemoteEncr` interface (Line 4). The `EncrypterWrapper` class defines a method `wrapEncrypt` which calls the static method `Encrypter.encrypt` (Line 8). In the non-enclave environment (Program B.14), the J_E compiler transforms the direct calls `Encrypter.encrypt(plaintext)` to a lookup of enclave remote class (Line 5) and to a call the `wrapEncrypt` method of the remote interface `EncrypterWrapper` (Line 7).

Packaging

All the classes to be placed inside and outside the enclave are packaged into two separate executable JAR files. Both JAR files contain an executable class, which includes code for initialization to set up the RMI registry and to publish remote objects required for communication. The user code executes after the initialization phase is complete. The compilation flow is illustrated in Figure B.3.

Implementation

We employ JavaParser [200] — a parser library for Java — to perform the code analysis and source code transformation described in section B.4. As shown in Figure B.3, the

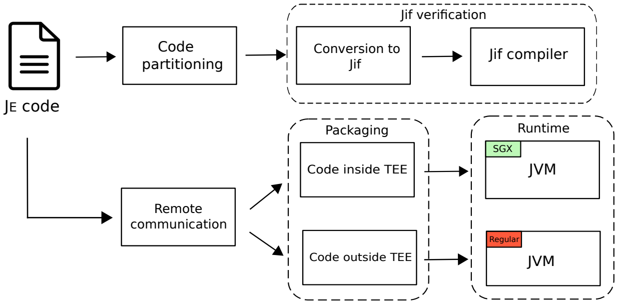


Figure B.3: J_E compilation phases

enclave program is deployed inside an Intel SGX enclave and executed using a JVM. We use the SGX-LKL framework [204] to support JVM execution inside the enclave. Running a JVM inside the enclave provides the advantages of managed languages but in comparison to running a native code, this approach suffers from a large TCB size.

B.5 Evaluation

We evaluate J_E with case studies to demonstrate that it can address the security requirements of distributed systems. The aim of the case studies is to demonstrate the core security features of J_E. The case studies consider basic Java constructs, mainly due to Jif's limited Java support and the various static constraints enforced by the Jif compiler. The case studies are of the order of tens of LOC.

We implement a **secure distributed event processing** (CEP) system based on AdaptiveCEP [149]. In CEP, event sources produce events that can carry data and are processed in a cluster of distributed nodes. AdaptiveCEP adapts the placement of the event processing operators to maximize throughput. In the secure CEP case study, every event is encrypted before it is sent to the cluster. Event processing nodes decrypt the events inside the enclave, perform the processing, and emit the corresponding output events.

Figure B.4 shows the code for secure CEP using a *filter* event processor node. The `FilterNode` class implements a filter node that can be placed on a remote machine. It contains a gateway method `filter` (Line 25) that accepts an encrypted event and an encrypted predicate. The event is decrypted inside the enclave (Line 32), the predicate is applied to the event payload, and the result is returned (Line 33).

This way, the event data are protected from an attacker that controls the OS. The attacker can only observe the encrypted events `EncEvent` passed as an argument to the `process` method.

We also implemented a **secure calculator** as described in [126]. In this case study, confidential data is placed inside the enclave and, during runtime, a user provides tasks to perform on confidential data from the non-enclave environment. The tasks are executed inside the enclave and the result is returned to the non-enclave environment.

```
class Main {
    public static void main(String[] args) {
        List<IntEvent> events =
        Generator.getIntEvents(100);
        List<EncIntEvent> encEvents =
        encrypt(events);
        List<EncIntEvent> result =
        encEvents.stream().
        filter(FilterNode.filter).
        collect(Collectors.toList());
    }
}

class EncIntEvent extends EncEvent {
    private EncInt val; // encrypted integer
    private String origin;
}
```

Program B.16

```
@Enclave
class FilterNode {

    @Secret private static String key;
    static private Predicate predicate;

    @Gateway
    public static boolean filter(EncIntEvent event) {
        if(!sanitize(event)) {
            return null;
        }
        boolean result;
        try {
            EncInt encInt = endorse(event).getVal();
            Integer val = decrypt(encInt, key);
            result = apply(predicate, val);
        } catch (Exception e) {
            return null;
        }
        return declassify(result);
    }
}
```

Program B.17

Figure B.4: Secure complex event processing

Figure B.5 illustrates tax computation on salary information using the secure calculator. The `TaskProcessor` class is annotated with the `@Enclave` annotation because it contains a secret field `salary` (Line 16) and a gateway method `process` (Line 19) that must be protected within the enclave. A user submits a list of tax-related computation tasks

```
 1  class Main {
 2      public static void main(String[] args) {
 3          List<Task> taskList = getTaskSeq("TAX");
 4          Double tax =
 5          TaskProcessor.process(taskList);
 6      }
 7  }
 8
 9  class Task {
10      public Double run (Double input) {
11          // Task computation
12      }
13  }
```

Program B.18

```
14  @Enclave
15  class TaskProcessor  {
16      @Secret static Double salary;
17
18      @Gateway
19      public static Double process(List<Task> taskList) {
20          if !(sanitize(taskList)) {
21              return null;
22          }
23          List<Task> taskListE = endorse(taskList);
24          try {
25              Double result = salary;
26              for (Task task : taskList) {
27                  result = task.run(result);
28              }
29          } catch (Exception e) {
30              return null;
31          }
32          return declassify(result);
33      }
34  }
```

Program B.19

Figure B.5: Secure calculator

(`taskList`) to the `process` method (Line 5) where the tasks are executed sequentially inside the enclave. The untrusted input `taskList` is first sanitized (Line 20) to verify that it holds certain integrity properties such as it is not null and not empty. If the sanitization check succeeds, then the `taskList` field is endorsed (23); otherwise, `null` is returned. When translating to Jif, we use Jif's *checked endorsement* construct to implement input sanitization. The input sanitization reduces the information leakage by only allowing the verified tasks to interact with the secret fields and influence the returned value.

The case study demonstrates that the code can be easily partitioned using annotations such that the sensitive information is kept inside the enclave, and a user from the non-enclave environment can only observe the result of predefined sensitive operations that are declassified explicitly by the developer.

```
class Main  {
    // opponent's grid status
    static boolean[][] gridOpp;

    public static void main(String[] args) {
        boolean gameOver = false;
        while(!gameOver) {
            // opponent's guess
            Guess g2 = getGuess();
            updateOppGrid(g2);
            int result = Grid.applyGuess(g2);
            // generate and send the guess
            // to the opponent along with
            // the result of the previous guess
        }
    }
}
```

Program B.20

```
@Enclave
class Grid  {
    @Secret private static boolean[][] grid;

    @Gateway
    public static int applyGuess(Guess guess) {
        Guess guessE = endorse(guess);
        // ship here?
        int result = apply(guessE);
        return declassify(result);
    }

    private static int apply(Guess guess) {
        // check for the presence of battleship
        return result;
    }
}
```

Program B.21

Figure B.6: Battleship game

Figure B.6 shows a J_E implementation of the **Battleship game** as in [34].

In the battleship game, each of the two players owns a secret grid. Initially, players position their battleships randomly on the grid. The game then proceeds in rounds and in each round, players guess a position on the opponent's grid. At the end of a round, players are told if the guessed location contains a battleship. The goal is to successfully guess all the battleship locations on the opponent's grid. The game ends when a player guesses all the battleship locations.

This case study demonstrates a scenario where we need to declassify some secret information depending on the external inputs. Initially, both grids are secret and in each round, a grid location needs to be made public via declassification. The location to declassify depends on the guess of the opponent; thus, we need to endorse the opponent's guess to declassify the location.

Figure B.6 shows the code run on every player's machine. The `Main` class is placed outside the enclave and it stores the status of the opponent's grid (Line 3). The `main` method consists of a while loop in which a player sends and receives guesses. The `Grid` class is placed inside the enclave. The secret variable `grid` (Line 20) stores the status of a player's grid. The gateway method `applyGuess` (Line 23) accepts the opponent's guess as an argument and checks if the guessed location contains a battleship. The argument `guess` is untrusted and we use the `endorse` operator (Line 24) to raise the integrity level to *Trusted*. The `apply` method (Line 30) extracts the array indices from the argument `guess` and checks if a battleship is present at the array location. The variable `result` has *Secret* confidentiality level as it is implicitly influenced by the secret field `grid` (inside the `apply` method, when checking for the battleship location). The subsequent declassification operation (Line 27) downgrades its confidentiality level to *Public*. The declassified value of `result` is returned to non-enclave environment (Line 27).

The case study demonstrates the use of the `endorse` and `declassify` operators to declassify secret information securely by endorsing the untrusted values explicitly.

B.6 Related Work

In this section we compare J_E to closely related works. We divide them into three broad categories.

Application partitioning for enclaves

Various works have considered partitioning an application for enclaves based on input provided by the user, such as annotations or configuration files. Glamdring [137] performs source-level partitioning of C code based on annotations. Panoply [142] creates low TCB application binaries from the annotated C applications. These works consider input sanitization checks across the enclave interface but do not employ information flow checks. Civet [178] and Uranus [170] perform Java application partitioning based on XML configuration and user annotations respectively. Secure Routines [161] extends annotation-based partitioning for Go programs. Unlike J_E, they consider only passive attackers and provide limited information-flow control guarantees.

Enclaves and information flow control

Gollamudi et al. [126] consider information flow control for enclave applications focusing on erasure policies. DFLATE [162] presents noninterference guarantees in distributed TEEs settings. In contrast to these works, J_E provides robustness guarantees against stronger active attackers. Moat [119] and its successor [130] automate confidentiality verification for enclaves programs.

Information flow control for distributed systems

Various works [24, 27, 31, 52, 53, 138, 139, 164] have employed information flow control techniques to prevent information leaks at the network boundaries in distributed settings. Inspired by these approaches, J_E uses IFC techniques to secure data flow across the enclave - non-enclave interface.

B.7 Conclusion

In this paper we presented J_E, a programming framework for enclave-enabled applications where developers use annotations to specify and guide the application partitioning and security policies. We implemented several case studies from literature showing that J_E correctly handles application partitioning while providing strong security guarantees against realistic attackers.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable feedback on this paper. We would like to thank Robert Kubicek, Jonas Seng, and Jesse-Jermaine Richter for their contribution to the J_E implementation. This work is partially supported by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the BRF Project 1025524 from the University of St.Gallen, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Digital Futures.

Paper C

Dynamic Policies Revisited

Amir M. Ahmadian and Musard Balliu

Abstract

Information flow control and dynamic policies is a difficult relationship yet to be fully understood. While dynamic policies are a natural choice in many real-world applications that downgrade and upgrade the sensitivity of information, understanding the meaning of security in this setting is challenging. In this paper we revisit the knowledge-based security conditions to reinstate a simple and intuitive security condition for dynamic policies: A program is secure if at any point during the execution the attacker's knowledge is in accordance with the active security policy at that execution point. Our key observation is the new notion of *policy consistency* to prevent policy changes whenever an attacker is already in possession of the information that the new policy intends to protect. We use this notion to study a range of realistic attackers including the perfect recall attacker, bounded attackers, and forgetful attackers, and their relationship. Importantly, our new security condition provides a clean connection between the dynamic policy and the underlying attacker model independently of the specific use case. We illustrate this by considering the different facets of dynamic policies in our framework.

On the verification side, we design and implement DynCoVer, a tool for checking dynamic information-flow policies for Java programs via symbolic execution and SMT solving. Our verification operates by first extracting a graph of program dependencies and then visiting the graph to check dynamic policies for a range of attackers. We evaluate the effectiveness and efficiency of DynCoVer on a benchmark of use cases from the literature and designed by ourselves, as well as the case study of a social network. The results show that DynCoVer can analyze small but intricate programs indicating that it can help verify security-critical parts of Java applications. We release DynCoVer publicly to support open science and encourage researchers to explore the topic further.

C.1 Introduction

Information flow control provides an appealing security framework for reasoning about dependencies between information sources and information sinks, and for ensuring that these dependencies adhere to desirable security policies. In a language-based setting, this security framework has the following ingredients: (1) an execution model which is given by the execution semantics of a program; (2) an attacker model specifying the observation power of an attacker over the attacker-visible sources and sinks; (3) a security policy specifying, for each execution point, the permitted information flows from sources to sinks, disallowing information flows from secret/high sources to public/low sinks; (4) a security condition (or security property) capturing a program's security with respect to an execution model, an attacker model, and a security policy. A classical security condition is noninterference requiring that any two executions starting with equal values on public sources yield equal values on public sinks [8].

A common trait in much recent work on information flow control has been the appeal to attacker-centric security conditions based on the concept of knowledge as a fundamental mechanism to bring out what security property is being sought and compare it with the knowledge permitted by the security policy [51, 61, 81]. Arguably, this appeal to knowledge, usually as equivalence relations on initial states, has produced clear and intuitive security conditions able to accommodate various notions of information downgrading (declassification) on which soundness arguments for enforcement mechanisms, e.g. security type systems, can be based [18, 28, 69]. In a nutshell, knowledge-based security conditions capture an intuitive requirement: A program is secure with respect to a security policy if at any point during the program's execution, the attacker's knowledge is not greater than the knowledge permitted by the active security policy at that execution point.

While this simple and elegant condition is well-understood for static multilevel security policies that only downgrade the sensitivity of information, it does not appropriately capture the security requirements of systems that change their security policies dynamically, thus both downgrading and upgrading the sensitivity of information in response to security-sensitive events. This is not satisfactory because dynamic policies are a natural choice for many real-world applications, e.g. healthcare systems, social networks, database systems, where access to information may be granted or revoked to different principals in accordance with their specific role at a given moment [88, 98, 114, 116]. Existing works address the challenge of dynamic policies by proposing security conditions that capture specific *facets* of a targeted use case [40, 49, 51, 56, 64, 68, 88, 98]. Broberg et al. [114] systematize existing research on dynamic policies illuminating the different facets exhibited by existing security conditions.

In this paper we revisit the state-of-the-art of security conditions for dynamic policies to reinstate attacker-centric knowledge-based conditions. Our starting point is the work of Askarov and Chong [88] which proposes a general framework for weakening of the attackers' observation power to accommodate dynamic policies. We further revise and develop this framework targeting three types of realistic attackers: (1) *perfect recall* attackers recalling all observations on public sinks; (2) *bounded* memory attackers recalling a bounded number of observations on public sinks; and (3) *forgetful* attackers recalling observations on public sinks up to a security policy change. While the first two attackers are standard, the third attacker, as we will see, is useful in settings where the release of knowledge is transient and it is limited to the event of a security policy change. For example, the event of changing the database policy to revoke access on a table to user A may reflect the security requirement that user A should no longer read data from that table, independently of whether or not user A accessed the table before the policy change.

Our key observation is the notion of *policy consistency* to reflect the observation power of an attacker and thus prevent a policy change whenever the attacker is already in possession of the information that the new policy intends to protect. For example, under the model of a perfect recall attacker, a policy change that revokes access to a resource that the attacker has already observed (possibly at a past time when access to that resource was granted to the attacker) should result in an inconsistent policy, since it violates the assumption on the perfect recall attacker. Unfortunately, existing works assume that policy changes are always consistent, which has often resulted in ad hoc and unintuitive security conditions. This simple but fundamental insight allows us to reestablish clear and intuitive attacker-centric knowledge-based conditions for dynamic policies. More importantly, the new security conditions are in line with the above-mentioned ingredients required by a security framework and they provide a clean separation between policy concerns and enforcement concerns. A policy designer can instantiate our framework in accordance with the security requirements for the use case at hand, by specifying the most suitable attacker model. We validate our framework by revisiting the facets of dynamic policies by Broberg et al. [114]. Moreover, in contrast to Askarov and Chong [88], we prove that, in absence of inconsistent policy changes, a perfect recall attacker is indeed stronger than a bounded memory attacker and a forgetful attacker. Finally, we discuss policy updates whenever inconsistencies are detected.

Our second contribution is the design and implementation of algorithms for verifying dynamic information-flow policies via symbolic execution and SMT solving. Our verification method operates by first extracting program dependencies and then using these dependencies to check dynamic policies for a range of attackers including perfect recall, bounded memory, and forgetful attackers. The verification algorithms adapt and extend existing approaches for checking noninterference via automated theorem proving [39, 90] and self composition [82] to the setting of dynamic policies, including the detection and repair of inconsistent policy changes. We implement [188] an open-source prototype for Java programs and evaluate the effectiveness and efficiency on a collection of benchmark from the literature and designed by ourselves, as well as the case study of a social network. The results show that DynCoVer can analyze small but intricate programs indicating that it can help verify security-critical parts of Java applications.

In summary the paper offers these contributions:

- We revisit the state-of-the-art security conditions for dynamic policies and reinstate clear and intuitive knowledge-based conditions based on the observation power of the attacker (Section C.2 and C.4).
- We show how our new framework can be used to capture the different facets of dynamic policies proposed in the literature (Section C.5).
- We design verification algorithms based on symbolic execution and SMT solving to check the security of Java programs for a range of attacker models, as well as to detect inconsistent policies (Section C.6).
- We implement DynCoVer [188] and evaluate the efficiency and effectiveness on a collection of benchmarks and the case study of a social network (Section C.7).

C.2 Problem Setting and Solution Overview

This section gives an informal overview of dynamic policies discussing the challenges, pointing out limitations of existing solutions, and arguing for revised knowledge-based security conditions. The key question is: *What is a suitable security condition for dynamic policies?*

To provide a common ground for comparing the different approaches, we borrow the notation and examples from Askarov and Chong [88] and Broberg et al. [114]. We write $A \rightarrow B$ ($A \not\rightarrow B$) to denote a security policy allowing (disallowing) information flows from security level A to security level B. We assume that no information flows between different security levels are allowed initially. For simplicity, the name of a program variable (e.g. `movie`) will match the security level of the variable (e.g. $Movie$).

We write k_i and p_i to denote the *attacker's knowledge* and the *active security policy* at program location i, respectively. By default, we assume that attackers are perfect recall, remembering any information they observe during a program's execution. A popular security condition [61, 81, 116], which is used in systems that handle only declassification of information, is given by equation (C.1) requiring that at any location i the attacker's knowledge k_i is smaller[1] than the policy knowledge (i.e. the knowledge allowed by the security policy) p_i.

$$p_i \subseteq k_i \tag{C.1}$$

[1] In this notation, knowledge corresponds to uncertainty, hence the bigger the set, the smaller the knowledge (see Section C.4).

Consider the scenario in Program C.1 where user *Alice* purchases a time-limited subscription on a streaming service to watch a `movie`. After the subscription ends, the security policy changes, however Alice still attempts to watch `movie`.

```
1  $Movie \rightarrow Alice$
2  Alice.watch(movie)
3  $Movie \not\rightarrow Alice$
4  Alice.watch(movie)
```

Program C.1

One could argue that this program should be considered insecure because *Alice* watches `movie` when she no longer has a subscription. Here, the release of knowledge is considered transient and it should satisfy the active security policy at every program location. Hence, despite being perfect recall, *Alice* should not be able to watch `movie` at a time this is disallowed by the policy (line 4). In fact, equation (C.1) holds in line 2 since *Alice* watches the movie and the policy allows her to watch `movie`, i.e. hence $p_2 \subseteq k_2$ since $\{\texttt{movie}\} \subseteq \{\texttt{movie}\}$. However, in line 4 we have that $p_4 \not\subseteq k_4$ since $\mathit{All} \not\subseteq \{\texttt{movie}\}$, where *All* denotes the set of all possible movies. Hence, the program is correctly rejected by the security condition (C.1).

Consider now Program C.2, a variation of Program C.1, displaying the message `NoSubscription!` after the second policy change. This program is also rejected by condition (C.1) since $p_4 \not\subseteq k_4$, i.e. $\mathit{All} \not\subseteq \{\texttt{movie}\}$, even though the `NoSubscription!` message does not leak anything.

```
1  $Movie \rightarrow Alice$
2  Alice.watch(movie)
3  $Movie \not\rightarrow Alice$
4  Alice.watch("NoSubscription!")
```

Program C.2

To address this case, Askarov and Chong [88] identify the power of perfect recall attacker as a key issue and present a security condition that accounts for weaker attackers. Their security condition requires that the attacker's change in knowledge should be allowed by the active policy, thus at any program location $i + 1$ the attacker's knowledge k_{i+1} should be smaller than the attacker's prior knowledge and the policy knowledge at location i:

$$p_i \cap k_i \subseteq k_{i+1} \tag{C.2}$$

Condition (C.2) now accepts Program C.2 as secure since $p_3 \cap k_3 \subseteq k_4$, i.e. $\mathit{All} \cap \{\texttt{movie}\} \subseteq \{\texttt{movie}\}$. However, surprisingly condition (C.2) also accepts Program C.1 since $p_3 \cap k_3 \subseteq k_4$, i.e. $\mathit{All} \cap \{\texttt{movie}\} \subseteq \{\texttt{movie}\}$. The root of the issue here is that perfect recall attacker is too powerful and can remember any observations made in the past, e.g. in line 2. To overcome this issue, Askarov and Chong only consider condition (C.2) for weaker attackers with bounded memory. For example, a bounded memory attacker that remembers only the last observation would now reject Program C.1 since the second observation of `movie` in line 4 reveals new information to a bounded memory attacker. In fact, now $p_3 \cap k_3 \subseteq k_4$ since $\mathit{All} \cap \mathit{All} \not\subseteq \{\texttt{movie}\}$. Similarly, Program C.2 is secure since a bounded attacker learns nothing about `movie` by observing the message `NoSubscription!`, namely $p_3 \cap k_3 \subseteq k_4$ since $\mathit{All} \cap \mathit{All} \subseteq \mathit{All}$. Condition (C.1) treats these programs similarly for a bounded memory attacker.

A key question arises at this point: *How does a policy designer choose the right attacker model, and hence security condition, for their setting*? While in principle there may always exist a bounded memory attacker that accommodates specific use cases as above, it is unclear what such attacker model should be. Askarov and Chong answer this question by requiring that condition (C.2) holds *for all attackers*, including perfect recall and bounded memory attackers. This is important because it enables compositional reasoning and facilitates enforcement by a security type system, however, security for all attackers can be too restrictive in settings where, e.g, only the perfect recall or a bounded memory attacker is realistic. In fact, Broberg et al. [114] discuss use cases where the same program can be considered either secure or insecure under different attacker models.

More importantly, condition (C.2) permits any policy changes although these changes may contradict the assumptions about the attacker.

Consider Program C.3 handling information about users' salaries. Initially, both *Alice* and *Bob* allow *Eve* to learn their salaries, however, the program displays only the average salary to *Eve*. Then *Alice* decides that her salary should no longer be visible to *Eve* and the program displays *Bob*'s salary.

```
1  Alice → Eve
2  Bob → Eve
3  output_Eve((Alice.salary + Bob.salary) / 2)
4  Alice ↛ Eve
5  output_Eve(Bob.salary)
```

Program C.3

Let As and Bs be the salary of *Alice* and *Bob*, respectively. Under a perfect recall attacker, Program C.3 satisfies condition (C.2). Indeed *Eve* can combine the average salary (line 2) and *Bob*'s salary to learn *Alice*'s salary, hence $k_5 = \{(As, Bs)\}$. Therefore, $p_4 \cap k_4 \subseteq k_5$ since $\{\mathit{All} \times Bs\} \cap \{(a, b) \mid (a + b)/2 = (As + Bs)/2\} = \{(As, Bs)\} \subseteq \{(As, Bs)\}$. The program is also accepted for a bounded memory attacker that remembers only the last output.

Under the perfect recall attacker, we argue that Program C.3 should not be accepted. The mere definition of perfect recall assumes that the attacker remembers any observations and can use these observations to infer information about the salaries of *Alice* and *Bob*. The real problem lies in the change of the policy in line 4. Because *Eve*'s knowledge in line 3 reveals some information about *Alice*'s salary (and *Eve* has perfect recall) the policy change in line 4 is *inconsistent*, trying to revoke access to a resource, i.e. *Alice*'s salary, that *Eve* has already some information about. Hence, the policy change in line 4 should be disallowed in the case of a perfect recall attacker. A similar argument applies to Program C.1 and Program C.2 under the perfect recall attacker. The reader may find this surprising, especially for Program C.2, but, again, the attacker has perfect recall, hence they remember the observation of `movie` in line 2. Therefore, restricting access to the attacker to some information they already have is meaningless and should be prevented by the security condition. In fact, by considering the intersection of the knowledge and policy ($p_i \cap k_i$), condition (C.2) effectively enforces condition (C.1) under a different policy $p = p_i \cap k_i$. This leads us to proposing a new (class of) conditions which is parameterized by an attacker A:

$$p_i^A \subseteq k_i^A \tag{C.3}$$

In contrast to condition C.1, condition C.3 makes the role of the attacker explicit in the definitions of knowledge and policy, as well as considers the consistency of a security policy. We instantiate the security condition (C.3) to characterize three attacker models: *perfect recall*, *bounded memory*, and *forgetful.* For the perfect recall attacker, our security condition corresponds to equation (C.1), ensuring that policy changes are consistent at any program location i. For a bounded memory attacker, the condition captures a weaker attacker which remembers observations up to a predefined bound m, extending the weaker attackers of Askarov and Chong [88] with policy consistency checks. Finally, the forgetful attacker captures scenarios in which the release of knowledge is transient and limited to the event of a policy change, thus ensuring that the attacker forgets (or resets) their knowledge whenever there is a policy change. This attacker model allows: (1) Reject Program C.1 since *Alice* attempts to (re-)watch the movies at a time this is prevented by the active policy; (2) Accept Program C.2 (Program C.3) since *Alice* (*Eve*) does not learn any information about `movie` (*Alice*'s `salary`) at any time this is prevented by the active policy.

C.3 Language Design

We present a simple imperative language with extended commands for policy change and outputs. We assume that outputs are performed on channels associated with security labels $X, Y, \ell \in \mathcal{L}$.

Syntax Figure C.1 presents the syntax of our language. Expression e consists of program variables x, values v, and binary operations $\oplus$. For simplicity, we restrict values to only integers n. Most of the commands are standard with the exception of `output` and `setPolicy`. Output command $\texttt{output}_\ell(e)$ evaluates expression e to

$$
\begin{array}{llll}
\mathit{Values} & v & ::= & n \\
\mathit{Expressions} & e & ::= & v \mid x \mid e_1 \oplus e_2 \\
\mathit{Commands} & c & ::= & \texttt{skip} \mid x := e \mid c_1; c_2 \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \\
 & & & \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{output}_\ell(e) \mid \texttt{setPolicy}(p)
\end{array}
$$

Figure C.1: Syntax

$$
\textsc{Skip}\ \frac{}{\langle \texttt{skip}; c, \sigma, p\rangle \xrightarrow{\epsilon} \langle c, \sigma, p\rangle}
\qquad
\textsc{Assign}\ \frac{\sigma(e) = v}{\langle x := e, \sigma, p\rangle \xrightarrow{\epsilon} \langle \texttt{skip}, \sigma[x \mapsto v], p\rangle}
$$

$$
\textsc{Seq}\ \frac{\langle c_1, \sigma, p\rangle \xrightarrow{\alpha} \langle c_1', \sigma', p'\rangle}{\langle c_1; c_2, \sigma, p\rangle \xrightarrow{\alpha} \langle c_1'; c_2, \sigma', p'\rangle}
\qquad
\textsc{If-Else-T}\ \frac{\sigma(e) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma, p\rangle \xrightarrow{\epsilon} \langle c_1, \sigma, p\rangle}
$$

$$
\textsc{If-Else-F}\ \frac{\sigma(e) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma, p\rangle \xrightarrow{\epsilon} \langle c_2, \sigma, p\rangle}
$$

$$
\textsc{While}\ \frac{}{\langle \texttt{while}\ e\ \texttt{do}\ c, \sigma, p\rangle \xrightarrow{\epsilon} \langle \texttt{if}\ e\ \texttt{then}\ (\texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip}, \sigma, p\rangle}
$$

$$
\textsc{Output}\ \frac{\sigma(e) = v}{\langle \texttt{output}_\ell(e), \sigma, p\rangle \xrightarrow{o(v,\ell)} \langle \texttt{skip}, \sigma, p\rangle}
\qquad
\textsc{Set-Policy}\ \frac{}{\langle \texttt{setPolicy}(p'), \sigma, p\rangle \xrightarrow{np(p')} \langle \texttt{skip}, \sigma, p'\rangle}
$$

Figure C.2: Semantics

some value v and then outputs v on channel ℓ. Command $\texttt{setPolicy}(p)$ sets the current security policy to p.

Semantics Figure C.2 presents the operational semantics of the language. A configuration is a tuple $\langle c, \sigma, p\rangle$ consisting of a command c, a store σ mapping variables to values (i.e. $\sigma = Vars \to Val$), and a policy p that represents the current active security policy. We use judgments of the form $\langle c, \sigma, p\rangle \xrightarrow{\alpha} \langle c', \sigma', p'\rangle$ to denote that configuration $\langle c, \sigma, p\rangle$ can take a single step to configuration $\langle c', \sigma', p'\rangle$ and optionally emit an event $\alpha \in Ev$.

Events Ev include $o(v, \ell)$ to denote the output of value v on channel ℓ, $np(p')$ to denote the activation of new policy p', or ϵ to indicate that no event was emitted.

We write $\sigma(e) = v$ to indicate that expression e evaluates to value v in store σ. We write $\sigma[x \mapsto v]$ to denote a new store that maps variable x to value v and otherwise behaves the same as σ. Most of the semantic rules are standard. Command $\texttt{setPolicy}(p')$ modifies a configuration to activate policy p' and emits the new policy event $np(p')$. Output command $\texttt{output}_\ell(e)$ evaluates e to value v, and emits the event $o(v, \ell)$.

A trace $t \in Ev^*$ is a (possibly empty) sequence of events. We write $\mid t \mid$ for the length of trace t and $t_1.t_2$ for concatenation of traces t_1 and t_2. We define projection of an event α to channel ℓ, written $\alpha\lfloor_\ell$, as: $\alpha\lfloor_\ell = \alpha$ if $\alpha = o(v, \ell)$, otherwise $\alpha\lfloor_\ell = \epsilon$. We lift projection to traces as: $(\alpha.t')\lfloor_\ell = \alpha\lfloor_\ell .t'\lfloor_\ell$ if $t = \alpha.t'$, otherwise $t\lfloor_\ell = \epsilon$.

We write $\langle c, \sigma, p\rangle \overset{t}{\Rightarrow} \langle c', \sigma', p'\rangle$ if $\langle c, \sigma, p\rangle$ takes *one or more* steps to reach configuration $\langle c', \sigma', p'\rangle$ while producing the trace t. We write $\langle c, \sigma, p\rangle \overset{t}{\Rightarrow}_n \langle c', \sigma', p'\rangle$ to denote n execution steps and omit the final configuration whenever it is irrelevant, as in $\langle c, \sigma, p\rangle \overset{t}{\Rightarrow}$. An execution point i denotes a configuration (c_i, σ_i, p_i) such that $\langle c_0, \sigma_0, p_0\rangle \overset{t}{\Rightarrow}_i \langle c_i, \sigma_i, p_i\rangle$.

C.4 Security Framework

In this section, we present knowledge-based and attacker-centric security conditions for a range of relevant attackers and explore their differences. As discussed informally in Section C.2, the security condition has the form $p_i^A \subseteq k_i^A$, meaning that in every step of the program's execution, the active security policy should be the upper bound of the attacker's knowledge. We instantiate this condition for different attacker models to consider security-relevant events such as policy changes and attacker-visible program outputs.

Security Polices

We consider a multi-user setting where a program c handles data on behalf of different users which are identified by security labels $\ell \in \mathcal{L}$. For simplicity, we assume that the users' data is read upfront and resides in the initial store of the computation. Section C.4 presents an extension to programs with arbitrary inputs.

A security policy p is a list of flows of the form $\ell_1 \rightarrow \ell_2$ and is used to define what an observer at a specific security label ℓ_2 is allowed to learn about the initial values with label ℓ_1. We assign security labels to the variables and use function Γ which is a mapping from variables to security labels (i.e. $Vars \mapsto \ell$). For simplicity, the name of a program variable (e.g. `x`) will match its security label (e.g. X), and we write $X \rightarrow A$ to denote that an observer at channel with label A can learn values with label X (i.e. X can flow to A). Throughout this paper, we use p_{init} as a predefined initial policy from which all programs start their execution. It is a simple reflexive policy that only allows a security label to flow to a corresponding channel with the same label (i.e. $X \rightarrow X$). We use $X \rightarrow A$ to add new non-reflexive flows to the list of allowed policies, and $X \not\rightarrow A$ to revert (disallow) such a flow.

A security policy induces an equivalence relation over stores. Intuitively, for the flow $X \to A$, two stores are related to each other by an equivalence relation for an observer on channel A if they have identical values for variables with security label X. Formally:

$$\sigma \equiv_A^p \sigma' \quad \text{iff} \quad \forall x \in Var:\ \Gamma(x) = X \ and\ X \to A \in p.\ \sigma(x) = \sigma'(x)$$

We write $[\sigma]_A^p$ for the set of stores in the same equivalence class as σ with respect to a policy p and an observer A. If $\sigma \equiv_A \sigma'$ (i.e. $\sigma' \in [\sigma]_A^p$), then an observer on channel A cannot distinguish between stores σ and σ'. Henceforth, we call such an observer the attacker and fix its label to A. Observe that more fine-grained policies can be defined in the expected manner by refining the definition of $[\sigma]_A^p$, e.g. by mapping each label to an equivalence relation on program stores as defined by the policy p [116, 163]. We discuss fine-grained policies in Section C.6.

In line with existing work on dynamic policies [88, 116, 163], we assume that policy changes are not observable externally, e.g. to an attacker A. In our multi-user setting, policy changes result from internal events of the underlying system itself, e.g. restricting access to a service, and these operations are typically carried out by the system administrator. This assumption applies to real-world scenarios where a user does not directly control their access rights, while the policies governing these access rights are introduced to the system by an administrator. Nevertheless, our framework can be easily extended to accommodate observable policy changes by considering such events similar to program outputs.

Security Conditions and Attacker Models

Our main focus is on the confidentiality of data, hence we consider a (logically omniscient) passive attacker that knows the program's source code and wants to deduce sensitive information about the initial store values. Our goal is to identify security conditions for dynamic policies by investigating the relationship between the attacker's knowledge and the policy knowledge for a range of attackers.

We present our knowledge-based security conditions for perfect recall and forgetful attackers. For space reasons, we refer the reader to Appendix C.3 for similar results on bounded memory attackers.

Perfect Recall Attacker

We model this attacker's knowledge of the initial store σ as a set k, which includes all of the possible stores that can produce the same observable trace. We assume the attacker with security level A is passively observing all outputs on channel with label A. When a command such as $\texttt{output}_A(v)$ executes, the attacker sees this output and learns the value v. We define attacker's knowledge as:

Definition C.1 (Perfect recall attacker knowledge at point i)

Given program c with initial store σ, and initial policy p_{init}, which produces trace t after i execution steps, i.e. $\langle c, \sigma, p_{init} \rangle \overset{t}{\Longrightarrow}_i$, the knowledge of a perfect recall attacker that observes program outputs on channel A is defined as:

$$k_i(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init} \rangle \overset{t'}{\Longrightarrow}_j \;\wedge\; t\lfloor_A = t'\lfloor_A\}$$

Intuitively, $k_i(c, \sigma, p_{init}, A)$ is the set of initial stores that the attacker at channel A believes are possible when observing the trace $t\lfloor_A$ at execution point i. Thus, the larger the knowledge set, the less certain the attacker is of the actual values in σ.

The *Perfect Recall* attacker has unlimited memory and can remember all outputs on channel A. This attacker is the most powerful attacker, because once they observe an output value they will never forget it, hence, a policy can no longer restrict the knowledge resulting from the attacker's past observations. Arguably, in presence of such an attacker, programs like C.3 should be rejected, because, as mentioned earlier, we cannot make the attacker forget what they already know. Therefore, it is not reasonable to issue a new policy that prevents this attacker from learning information which they already know.

With this intuition in mind, we need to ensure that any policy update is consistent with the current knowledge of the perfect recall attacker:

Definition C.2 (Policy consistency)

For all execution points i, and security policies p_i such that

$$\langle c, \sigma, p_{init} \rangle \overset{t}{\Longrightarrow}_{i-1} \langle \texttt{setPolicy}(p_i); c_i, \sigma_{i-1}, p_{i-1} \rangle \xrightarrow{np(p_i)} \langle c_i, \sigma_{i-1}, p_i \rangle$$

we say policy p_i is consistent with the current attacker knowledge $k_{i-1}(c, \sigma, p_{init}, A)$ if $[\sigma]_A^{p_i} \subseteq k_{i-1}(c, \sigma, p_{init}, A)$.

A security policy induces an equivalence relation over all possible stores, and $[\sigma]_A^{p_i}$ is a set of stores in the same equivalence class as initial store σ. Since by Definition C.1 attacker knowledge $k_{i-1}(c, \sigma, p_{init}, A)$ is also a set of initial stores, it is straightforward to check $[\sigma]_A^{p_i} \subseteq k_{i-1}(c, \sigma, p_{init}, A)$.

```
1  setPolicy(X → A);
2  output_A(1);
3  setPolicy(X ↛ A);
4  output_A(x);
```

Program C.4

For example in Program C.4, the new policy $X \not\rightarrow A$ in line 3 is consistent, because the attacker does not learn anything about x by observing the output in line 2, hence the new policy that disallows learning x is consistent. However, under this new policy, Program C.4 should be rejected, because the output of x in line 4 happens at a time when the policy does not allow it.

We follow this intuition to define security.

Definition C.3 (Observation security)

For all execution points i such that

$$\langle c, \sigma, p_{init} \rangle \overset{t}{\Rightarrow}_{i-1} \langle \texttt{output}_A(e); c_i, \sigma_{i-1}, p_{i-1} \rangle \xrightarrow{\alpha} \langle c_i, \sigma_i, p_i \rangle$$

program c is secure if $[\sigma]_A^{p_i} \subseteq k_i(c, \sigma, p_{init}, A)$.

This definition ensures that, whenever an output on channel A happens, the attacker's knowledge at that point is allowed by the current policy. In other words, the policy places an upper bound on the attacker's knowledge[2], and if the knowledge does not exceeds that limit, the program is secure.

Definition C.4 (Security condition for perfect recall)

A program c is secure under the perfect recall attacker if Definitions C.2 and C.3 hold.

We deliberately separate Definitions C.2 and C.3 to distinguish between policy consistency checks and security checks. A failed policy consistency check means that there is a mismatch between the attacker's power and the new policy. Therefore, a policy inconsistency can be repaired with a new policy that takes into account the attacker's knowledge. On the other hand, a failure of observation security (Definition C.3) cannot be repaired and it implies that the program is insecure.

Forgetful Attacker

We now consider an attacker that resets its knowledge after a policy change. This is specially useful for real-world applications where the release of information is not permanent and should be consistent with the active security policy at the time of the release. Program C.1 in Section C.2 is an example of the usefulness of this attacker model. Intuitively, a policy change at an execution point i means that, from the point i onward, the new policy should govern the release of information and any past knowledge should be ignored. The term "forgetful attacker" may not exactly reflect a real world attacker that suddenly forgets everything after a policy change. It is an artifact of modeling scenarios in which a policy change enforces a new

[2] Observe that the attacker's knowledge, in contrast to the policy knowledge, is precise, and it is not an upper bound.

condition on information release, independently of what an attacker may already know as result of past observations. For example, an employee may have accessed a company's information (and even stored it externally), however, no access to the same information should be allowed when they leave the company. This setting requires ignoring the attacker's knowledge prior to the policy change, essentially resulting in a forgetful attacker. We first discuss some examples illuminating the subtleties of forgetful attackers and then present our security condition.

```
1  setPolicy(X → A, Y → A);
2  output_A(y);
3  if (x > 0) then
4      output_A(1);
5  setPolicy(X ↛ A, Y ↛ A);
6  if (x <= 0) then
7      output_A(1);
8  output_A( 2 );
```

Program C.5

Consider Program C.5 as an example. When the execution reaches the `setPolicy` command in line 5, there are two possible traces that the attacker could have observed: $t_1 = y.1$ and $t_2 = y$, both leaking the value of `y`. One may think that trace t_1 leaks the sign of x but this is not the case. Because the attacker's knowledge is derived through observations and the policy changes are not observable, the attacker cannot tell which `if` statement has produced the output 1 of trace t_1. Therefore, these traces reveal nothing about `x`, and at the time of policy change, the attacker only knows `y` as stipulated by the policy in line 1. The new policy now prevents the attacker from learning `y` again, hence no outputs after the policy change should leak `y`. In fact, all executions after the policy change will output the values 1 and 2. Note that even though the output 1 on line 7 happens after the policy change, it still cannot leak the sign of `x`, therefore, Program C.5 is secure.

Program C.6 is similar except that it outputs the value of `y` at lines 4 and 7. In this case, if the execution did not take the first `if` statement, the attacker observes the trace $t = 1$ which leaks nothing about `x` or `y`, thus at the time of policy change the attacker forgets nothing. However, when `y` is outputted in line 7, the attacker learns this value and because this is not allowed by the policy, the program is insecure.

```
1  setPolicy(X → A, Y → A);
2  output_A(1);
3  if (x > 0) then
4      output_A(y);
5  setPolicy(X ↛ A, Y ↛ A);
6  if (x <= 0) then
7      output_A(y);
8  output_A(2);
```

Program C.6

Programs can leak through the progress of computation e.g. when the number of outputs depends on information that is disallowed by the policy [55]. It is our intuition that forgetful attackers should not forget these progress leaks. Once the length of a trace, i.e. the number of outputs, leaks some information, any extension of that trace will leak the same information again, thus it is not reasonable for a forgetful attacker to forget progress leaks. This can be captured by making forgetful attackers remember the number of observed outputs, including the ones that happened before a policy change. This approach captures progress leaks even when they manifest after a policy change. This is similar to the idea of counting attackers presented in van Deft et al. [116].

```
1  setPolicy(X → A);
2  if (x > 0) then
3      output_A(1);
4      output_A(1);
5  else
6      output_A(1);
7  setPolicy(X ↛ A);
8  output_A(1);
```

Program C.7

Program C.7 illustrates the effect of progress leaks on the attacker's knowledge. When the execution reaches the policy change at line 7, the attacker could have observed trace $t_1 = 1$ or $t_2 = 1.1$. Since the policy change event is not observable by the attacker and any program execution can yield at least 2 outputs (e.g. trace 1.1), the attacker learns nothing about `x`. Later, after the new policy at line 7 becomes active, the output at line 8 occurs. One of the traces that the attacker could have observed at this point is $t_2 = 1.1.1$. This trace leaks that the first `if` statement must have been executed and `x > 0`, which violates the active policy. Therefore, the number of outputs leaks the sign of x, which happens after the policy change, hence the program should be flagged as insecure.

With these intuitions in mind, we proceed to define the knowledge of forgetful attacker. We call the trace between two policy changes an epoch and use the policy events ($np(p)$) to partition the trace into multiple epochs. At each step, the observable events of the last epoch, as well as the number of events in previous epochs can affect the forgetful attacker's knowledge. To be able to separate the last epoch from the whole trace, we define the following auxiliary functions: *splitPolicy(t)* takes a trace t and returns a tuple containing all events before and after the last new policy ($np(p)$) event; *split(t, n)* takes a trace t and a number n, and returns a tuple containing the first n events and the reminder of events in the trace.

Definition C.5

Given a trace t such that $t = \alpha_1 \ldots \alpha_i \ldots \alpha_k$,

$$\text{splitPolicy(t)} = \begin{cases} (\epsilon, t) & \text{if } \alpha_r \neq np(p)\ r = 1 \ldots k \\ (\alpha_1 ... \alpha_i\ ,\ \alpha_{i+1} ... \alpha_k) & \text{if } \alpha_i = np(p) \\ & \quad \wedge\ \alpha_r \neq np(p)\ r = i+1 \ldots k \\ (t, \epsilon) & \text{if } \alpha_k = np(p) \end{cases}$$

Definition C.6

Given a trace t such that $t = \alpha_1 \dots \alpha_i . \alpha_{i+1} \dots \alpha_k$,

$$\text{split(t, n)} = \begin{cases} (t, \epsilon) & \text{if } k \leq n \\ (\alpha_1 \dots \alpha_i \ , \ \alpha_{i+1} \dots \alpha_k) & \text{if } i = n \end{cases}$$

Using these auxiliary functions, we can proceed to define the forgetful attacker's knowledge as:

Definition C.7 (Forgetful attacker knowledge at point i)

Program c with initial store σ and initial policy p_{init} produces trace t after i execution steps, i.e. $\langle c, \sigma, p_{init} \rangle \xRightarrow{t}_i$. Let $(t_1, t_2) = \text{splitPolicy(t)}$, we define for the knowledge of a forgetful attacker that observes the program outputs on channel A as:

$$\begin{aligned} k_i^{frg}(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init} \rangle \xRightarrow{t''}_j & \\ \wedge \ (t_1'', t_2'') &= \text{split}(t'' \lfloor_A, \mid t_1 \lfloor_A \mid) \\ \wedge \ t_2'' &= t_2 \lfloor_A \} \end{aligned}$$

Intuitively, for each execution point i, we identify the traces before (t_1) and after (t_2) the last policy change. The goal is to forget the knowledge induced by attacker's trace $t_1 \lfloor_A$ and compute the knowledge induced by $t_2 \lfloor_A$. We achieve this by considering any initial states that produce the same *number* of outputs as $\mid t_1 \lfloor_A \mid$ and the same outputs as $t_2 \lfloor_A$. Note that this condition (as all our conditions) is progress sensitive and accounts for progress leaks. We remark that progress leaks are never forgotten once they are revealed at some execution point.

We can now use the definition of knowledge from Definition C.7 to obtain the security condition for forgetful attackers.

Definition C.8 (Security condition for forgetful attacker)

For all execution points i such that

$$\langle c, \sigma, p_{init} \rangle \xRightarrow{t}_{i-1} \langle \texttt{output}_A(e); c_i, \sigma_{i-1}, p_{i-1} \rangle \ \xrightarrow{\alpha} \langle c_i, \sigma_i, p_i \rangle$$

program c is secure if $[\sigma]_A^{p_i} \subseteq k_i^{frg}(c, \sigma, p_{init}, A)$.

Appendix C.2 exercises the definition for Programs C.5–C.7 to investigate their security.

We can now show that if a program is secure against the perfect recall attacker, it is also secure against less powerful attackers such as bounded memory attackers and forgetful attackers. Here, we present a theorem and prove this claim for the forgetful attackers.

Theorem C.1

Given a program c, initial store σ, and initial policy p_{init}, if for all execution points i, c is secure against perfect recall attacker A_{per}, it is also secure against forgetful attacker A_{frg}. Formally:

$$[\sigma]_A^{p_i} \subseteq k_i(c, \sigma, p_{init}, A_{per}) \implies [\sigma]_A^{p_i} \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

Appendix C.3 contains the proof of Theorem C.1 and the corresponding theorem for bounded memory attackers.

Repairing Inconsistent Policies

An inconsistent policy means that the policy is incompatible with the current knowledge of the attacker. One approach is to always reject the programs with inconsistent polices, because $p_i \not\subseteq k_i$. Alternatively, we can suggest the user a new policy that is consistent with the attacker's current knowledge. Generally, a policy change is inconsistent if it restricts access to information that has already been learned by the attacker. Our goal is to relax these restrictions and add what has been learned by the attacker to the new policy to achieve a consistent policy. The intersection of the new policy and the attacker's knowledge ($p_i \cap k_i$) is a good candidate because it includes all of the new flows introduced by the new policy, and uses the knowledge of the attacker to relax the restrictions of the new policy. Intuitively, the consistent policy $p_i \cap k_i$ corresponds to the most adequate policy that meets the intention of the policy change p_i while being in line with the attacker's current knowledge k_i.

Definition C.9 (Consistent policy repair)

For all execution points i such that

$$\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_{i-1} \langle setPolicy(p_i); c_i, \sigma_{i-1}, p_{i-1}\rangle \xrightarrow{np(p_i')} \langle c_i, \sigma_{i-1}, p_i'\rangle$$

the repaired policy p_i' is induced by $[\sigma]_A^{p_i} \cap k_{i-1}(c, \sigma, p_{init}, A)$.

While previous approaches [88, 116, 163] use intersection as part of the security conditions, here we emphasize that it corresponds to a new consistent policy, thus making it explicit for the user that security of the program is checked against a different (repaired) policy.

Generalization to Programs with Inputs

In a framework of dynamic policies, it is natural to model new information arriving into the system via input channels. We show how our framework can accommodate programs with inputs with minimal changes. We extend the syntax of the language with an input command $\mathtt{input}_\ell(\mathtt{x})$ which reads a value from the input channel with label ℓ and assigns it to variable x. Clark and Hunt [57] have shown that for deterministic interactive systems, streams are sufficient to model arbitrary interactive input strategies. An input stream is an infinite sequence of values representing the pending inputs on a channel. We assume there is one input channel for each security level ℓ and an input environment ω mapping labels to input streams. We extend the configurations $\langle c, \sigma, p, \omega\rangle$ with the input environment ω and the evaluation steps as expected. We write $v : vs$ for a input stream with the first element v and remaining elements vs, and $\omega[\ell \mapsto vs]$ for the input environment that maps input stream with label ℓ to vs and otherwise behaves the same as ω. The semantics of input command is defined as:

$$\textsc{Input}\ \frac{\omega(\ell) = v : vs \quad \omega' = \omega[\ell \mapsto vs]}{\langle \mathtt{input}_\ell(\mathtt{x}), \sigma, p, \omega\rangle \xrightarrow{i(v,\ell)} \langle \mathtt{skip}, \sigma[x \mapsto v], p, \omega'\rangle}$$

This command updates the store σ with value v for variable x, and continues with vs as the reminder of the input stream of label ℓ, while emitting the input event $i(v, \ell)$. Events and traces are extended with input events in the expected manner.

We can now define security policies as equivalence relations over input environments. Two input environments are equivalent for a policy p and an attacker on channel A, i.e. $\omega \equiv_p^A \omega' \quad \text{iff } \forall \ell \to A \in p \,.\, \omega(\ell) = \omega'(\ell)$. We write $[\omega]_A^p$ for the equivalence class of ω with respect to the policy p and attacker A. With these definitions at hand, we can easily redefine the attacker knowledge over input environments and use the same security conditions adapted with the new definitions of attacker knowledge and security policies. This extension are straightforward and we omit them here in the interest of space.

C.5 Facets of Dynamic Policies

In this section we revisit the facets of dynamic policies, introduced by Broberg et al. [114], and discuss them in our framework from an attacker-centric perspective. Our goal is to show how these facets can be accommodated in our framework, illuminating on the different types of flows. We have modified and adapted the use cases to fit our language model with explicit outputs.

Time-transitive flows A flow is time-transitive if it moves information from level X to level Z via a third level Y, while a direct flow from X to Z is never allowed by the policy. Program C.8 illustrates such a flow. It reveals information about *Patient* to *DrPhil* who joined the hospital after *Patient* had left.

```
1  setPolicy(Patient → Hospital, Hospital ↛ DrPhil);
2  hospital := patientData;
3  setPolicy(Patient ↛ Hospital, Hospital → DrPhil);
4  drPhil := hospital;
5  output_{DrPhil}(drPhil);
```

Program C.8

According to Broberg et al. [114] time-transitive flows should be considered insecure in scenarios where a data flow such as $Patient \rightarrow Hospital$ is only allowed temporary for as long as *Patient* is under treatment in the hospital. Our framework can identify the insecurity of such scenarios; when `output`$_{DrPhil}$`(drPhil)` happens it indirectly reveals the value of `patientData` which is not allowed by the active policy. A similar argument applies to bounded memory and forgetful attackers. The main reason for rejecting these flows is that the observer *DrPhil* did not see the data at the time he was allowed to and the actual flow has happened at a time when patient had already left the hospital.

Broberg et al. [114] also interpret time-transitive flows as secure by considering the assignment in line 2 as a *permanent declassification*. Using permanent declassification means changing the label of `patientData` to *Hospital* permanently, however, this means that the policy $Patient \not\rightarrow Hospital$ becomes irrelevant, since `patientData` no longer has the label of *Patient* and hence it not affected by its policy. In our framework, this interpretation amounts to a program that allows flows from *Hospital* to *DrPhil* and subsequently outputs the data to *DrPhil*.

Replaying flows model scenarios in which when a piece of information is released, it can be released again, regardless of the active policy. Program C.9 illustrates such a flow. When `creditcard` is written to a log file, it should be available until the log is cleared.

```
setPolicy(Creditcard → Log);
output_{Log}(creditcard);
setPolicy(Creditcard ↛ Log);
output_{Log}(creditcard);
```

Program C.9

Replaying flows should be considered secure when the release of information is *permanent* [114]. Permanent release of information means that an observer can access any information they had learned before, irregardless of the active policy. For example in Program C.9, if the output in line 2 permanently releases the `creditcard` information to *Log*, the observer at level *Log* can always access it later. This definition can be captured by our framework, by adapting the Definition C.9 for inconsistent policies. This means that the new policy will be the intersection of the knowledge of the attacker k and new the policy p, and since the output in line 2 adds `creditcard` to the knowledge of the attacker, $k \cap p$ will always include that knowledge, hence permanently releasing it.

However, considering information as permanently released is not always the natural choice in every situation; for example in Program C.1, Alice should not have access to the `movie` after her subscription ended. Forgetful attackers in our framework are good candidates for dealing with such scenarios where we want to ignore the effects of the earlier release and accept or reject programs only based on the current active policy. This intuition is similar to the *insecure* time-limited subscription example of Broberg et al. [114].

```
1  setPolicy(Salary → Screen);
2  output_Screen(0);
3  setPolicy(Salary ↛ Screen);
4  output_Screen(salary);
```

Program C.10

Direct Release means that information is considered released as soon as the current policy permits the flow. Program C.10 illustrates such a flow where `salary` is not printed to the screen when the flow is allowed, but it is printed when the flow is no longer permitted.

Broberg et al. [114] argue that these flows are insecure when the attacker can only observe information that is actively provided (through for example an output channel). In Program C.10 nothing about `salary` has been printed to the screen, hence it makes sense to assume that an observer does not know this information. Our framework follows this intuition and rejects this flow under all attacker types and policy checks; because an attacker learns nothing from output of line 2 and the output on line 4 always violates the active policy.

However, this type of flow can be considered secure if we model attackers as constantly observing, directly in the memory, all the information which they are allowed to learn. We can model such a behavior by outputting all the variables with label *Salary* as soon as the policy *Salary* → *Screen* is activated. However, doing so effectively changes the nature of this flow to a replaying flow, and as it was discussed earlier, replaying flows can be secure only if we consider permanent release of information.

Whitelisting flows A flow is allowed whenever there is some part of the policy that allows for it. Program C.3 in Section C.2 is an example of whitelisting flows, where the observer *Eve* can use her knowledge of `(Alice.salary + Bob.salary) / 2` and `Bob.salary` to learn the salary of Alice. For perfect recall attackers, our framework rejects this class of programs because they have inconsistent policy changes. This is inline with the insecure example presented by Broberg et al. [114] which argue that information belonging to two entities *Alice* and *Bob* (in this case the average of their salaries) should be available only when both of them allow it.

```
setPolicy(Secret → Public, Key → Public);
output_Public(secret XOR key);
setPolicy(Secret ↛ Public, Key → Public);
output_Public(key);
```

Program C.11

Broberg et al. [114] presents Program C.11 as a secure example for whitelisting flows, where first the encrypted value (`secret XOR key`) is released and then later the `key`. Broberg et al. [114] argue for the security of this example on the grounds that `key` is an encryption key, and "with the release of this key an observer learns the secret information that was earlier released encrypted under that key, even though part of the policy does not allow the secret to be released". This intuition is inline with inconsistent policy repair of Definition C.9, since we know that the encrypted values have already been outputted and publishing the key releases them as well, we should either explicitly add $Secret \rightarrow Public$ to the policy, or use Definition C.9 to update the policy.

C.6 Verification of Dynamic Policies

This section discusses the precise verification of dynamic policies by symbolic execution and automated theorem proving. Our verification approach operates in two phases: (1) it extracts the dependencies of a source program by means of symbolic execution and (2) it verifies the security conditions for dynamic policies under different attacker models by relying on an SMT solver.

We impose some restrictions on the source program to make the analysis in (1) feasible. First, we assume a bounded model of runtime behavior, hence programs always terminate. Second, we assume all inputs from external environments can be read at the beginning. Hence, to support programs with inputs, one can assign fresh variables to each of the elements of a (finite) input stream. The output of phase (1) is a graph capturing dependencies between program inputs and outputs. We refer to existing works for details on symbolic execution [35].

Specifically, we analyze source programs symbolically to extract precise dependencies between program inputs and outputs. Observe that this information is sufficient to reason about security because security policies refer to program inputs and attacker observations are made though program outputs.

For each program output, our symbolic analysis stores a path condition $Pc \in PC$ and an output expressions $e \in Exp$ which are defined over the program inputs. The path condition is a predicate that represents the set of initial concrete values that trigger the execution of an output expression e. In particular, any *satisfying assignment*[3] δ of path condition Pc determines a concrete program output as computed by $\delta(e)$.

We represent these dependencies in the form of symbolic output trees (SOT) consisting of: (a) a set of nodes B labeled with output expressions $e \in Exp$; (b) a set of control flow edges $E \subseteq B \times B$; (c) a set of path conditions PC; (d) a mapping from nodes to output expressions $O : B \mapsto Exp$; (e) a mapping from edges to path conditions $L : E \mapsto PC$; and (f) a root node *Start*.

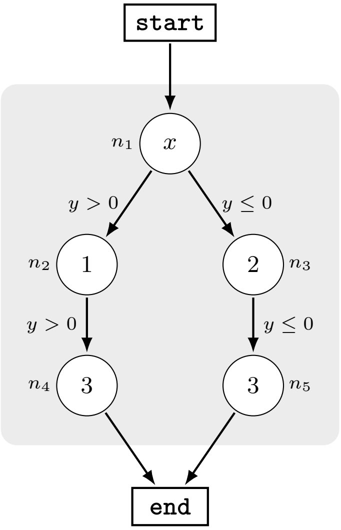


Figure C.3: SOT of Program C.12

```
setPolicy(X → A, (Y > 0) → A);
output_A(x);
if (y > 0) then
    output_A(1);
else
    output_A(2);
output_A(3);
```

Program C.12

We also extend the SOT with a special node *End*, in order to make terminal states explicit in the construction. Figure C.3 illustrates the SOT of Program C.12. Node $n3$ indicates that for all initial values of variable y such that $y \leq 0$, the second output of the program is the expression 2.

We define security policies with respect to an attacker at security level A. For a program variable x such that $X \rightarrow A$, we denote its initial value by l (for low) to reflect that variable x can be observed by A, otherwise we denote it by h (for high). We lift this notation to tuples of input variables $\vec{l}$ and $\vec{h}$ in the expected manner. Moreover, we support fine-grained policies modeled by predicates ϕ over initial values of variables. We define the policy P over $\vec{l}$, $\vec{h}$, written as $P(\vec{l}, \vec{h})$ by the predicate:

$$\vec{l} = \vec{l}\,' \wedge \phi \tag{C.4}$$

where $\vec{l}\,'$ stands for the renames of low variables, and predicate ϕ represents the leaked (i.e. declassified) expressions which is defined over low and high identifiers,

[3] A satisfying assignment is a mapping from the free variables of Pc to values, which makes the predicate Pc evaluate to true.

and their renames. The policy predicate $P(\overrightarrow{l}, \overrightarrow{h})$ induces an equivalence relation $[\sigma]_A^P$ over initial stores σ, which corresponds to the policy knowledge (cf. Section C.4). The relation can be constructed as follows: Let σ and σ' be two program stores over program variables and their renames, respectively, $\mathit{unprime}(S)$ be an operator "undoing" the variable renames over a set S, and $q(\sigma)(\sigma')$ be the evaluation of a predicate q over σ and σ'. Then the equivalence relation $[\sigma]_A^P = \mathit{unprime}(\{\sigma' | P(\overrightarrow{l}, \overrightarrow{h})(\sigma)(\sigma')\})$ defines the policy knowledge induced by the predicate $P(\overrightarrow{l}, \overrightarrow{h})$.

For example, the policy in line 1 of Program C.12 means that variable x is low, variable y is high, and expression $y > 0$ is leaked. Following equation (C.4), we write the policy $P(x, y)$ as $x = x' \wedge (y > 0 = y' > 0)$.

We remark that the policy $P(\overrightarrow{l}, \overrightarrow{h})$ corresponds to a global static policy encoding of a standard declassification policy ϕ [29]. In fact, prior work by Balliu et al. [90] shows how a static policy $P(\overrightarrow{l}, \overrightarrow{h})$ can be verified against an SOT by means of an SMT solver. Definition C.10 presents the process of generating such a formula.

Definition C.10

An SOT S is secure wrt. a security policy $P(\overrightarrow{l}, \overrightarrow{h})$ iff the following formula is unsatisfiable:

$$P(\overrightarrow{l}, \overrightarrow{h}) \wedge \bigvee_{n \in N(S)} \Bigg(Pc_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(\bigwedge_{n' \in N(S)} \neg\Big(Pc_{n'}(\overrightarrow{l}, \overrightarrow{h}') \wedge \overrightarrow{O}_n(\overrightarrow{l}, \overrightarrow{h}) = \overrightarrow{O}_{n'}(\overrightarrow{l}, \overrightarrow{h}') \Big) \Bigg) \Bigg)$$

where $\overrightarrow{O}_n$ is the tuple of output expressions along the SOT path to node n, $\overrightarrow{O}_n = \overrightarrow{O}_{n'}$ denotes the component-wise equality of two tuples, and $N(S)$ is the nodes of S.

Definition C.10 presents a logical encoding of our security condition $p \subseteq k_i$ of Section C.4 for the perfect recall attacker and a static policy p. Specifically, the condition is true if all initial stores that satisfy policy p are contained in the attacker's knowledge set k_i at each program point i. We focus only on program outputs, since non-observable commands do not affect knowledge. The non-satisfiability of the formula above implies that for any initial state $(\overrightarrow{l}, \overrightarrow{h})$ that satisfies the policy $P(\overrightarrow{l}, \overrightarrow{h})$ and reaches some output node n (i.e. $n \in N(S)$ and $Pc_n(\overrightarrow{l}, \overrightarrow{h})$) yielding an output sequence $\overrightarrow{O}_n$, it is impossible to find another state $(\overrightarrow{l}, \overrightarrow{h}')$ that satisfies the policy and reaches some output node n' (i.e. $n \in N(S)$ and $Pc_{n'}(\overrightarrow{l}, \overrightarrow{h}')$) yielding a different output sequence $\overrightarrow{O}_{n'}$. Consequently, for any initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P(\overrightarrow{l}, \overrightarrow{h})$, we have that $(\overrightarrow{l}, \overrightarrow{h}) \in k_i$, thus verifying the security condition. By contrast, if the formula is satisfiable, there exist two stores that satisfy the policy $P(\overrightarrow{l}, \overrightarrow{h})$, but either one store does not reach the output node or the two stores produce different output sequences. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_i$, thus violating the security condition.

For example, the SOT of Figure C.3 is secure wrt. the above-mentioned policy $P(x, y)$, as witnessed by the following unsatisfiable formula:

$$x = x' \wedge (y > 0 = y' > 0) \wedge \bigvee_{n \in N(S)} \Bigg(Pc_n(x, y) \wedge \\ \Bigg(\bigwedge_{n' \in N(S)} \neg \Big(Pc_{n'}(x, y') \wedge \overrightarrow{O}_n(x, y) = \overrightarrow{O}_{n'}(x, y') \Big) \Bigg) \Bigg)$$

We revise this condition to verify deterministic programs with dynamic policies for our attacker models. In a dynamic setting, the active policy at each node of S might be different from its parent or children. Therefore, instead of generating a single formula for the whole SOT, we need to generate a formula for every node n corresponding to its policy $P_n(\overrightarrow{l}, \overrightarrow{h})$. To this end, we modify the SOT generation algorithm and enrich each node with an additional attribute to store the active policy at the time of its creation.

Perfect Recall Attacker

We use Definition C.11 to check the security of a program wrt. the perfect recall attacker.

Definition C.11

An SOT S secure iff for all $n \in N(S)$ with active policy $P_n(\overrightarrow{l}, \overrightarrow{h})$, the following formula is unsatisfiable:

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n}) \wedge \Bigg(\bigwedge_{n' \in N(S)} \\ \neg \Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_n(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \Big) \Bigg) \Bigg)$$

In contrast to Definition C.11 here the active policy can be different in each node (as denoted by $Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n})$). For a node n the formula is unsatisfiable only if there is no other node with a satisfiable path condition $Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}')$ that can produce a different output. Unsatisfiability of the formula for a node n means that the program is secure wrt. the active policy at that node. To ensure security for the SOT we repeat this process for all nodes, regenerate the formula at each node and check its satisfiability. If none of the formulas are satisfiable, we can conclude that the SOT S is secure.

We use a similar approach to verify the policy consistency. However, because we do not have specific nodes for policy changes, we mark all of the nodes that appear right after a policy change and only check the policy consistency on those nodes using the following definition:

Definition C.12

Given an SOT S, active policy $P_n(\overrightarrow{l}, \overrightarrow{h})$, and $parent(n)$ which returns the parent of node n, a policy change is consistent iff for all $n \in N(S)$ such that n comes right after a policy change, the following formula is unsatisfiable:

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n}) \; \wedge \Bigg(\bigwedge_{n' \in N(S)}$$

$$\neg \Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \Big) \Bigg) \Bigg)$$

If a node was marked as an output after a policy change, before checking its security using Definition C.11, we first use Definition C.12 to check the policy consistency. The process is similar to Definition C.11, except that here instead of using the path condition and output of node n, we use the path condition and output of its parent ($Pc_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n})$ and $\overrightarrow{O}_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n})$, respectively).

Following Definition C.2 in Section C.4, we check that the attacker knowledge is allowed by the new policy. If node n is marked by the policy change it means that a policy change has happened between n and $parent(n)$, so we use the output of the node before the policy change (parent node) $\overrightarrow{O}_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n})$ and the new policy (policy of the current node) $P_n(\overrightarrow{l}, \overrightarrow{h})$ to generate the formula, and check that the new policy is in line with the observations up to n's parent.

The unsatisfiability of this formula means that the policy change between nodes $parent(n)$ and n is consistent.

Figure C.4 illustrates the SOT of Program C.13. As in the previous example, a node with expression x represents outputting the initial value of x (with $Pc = true$), while $y > 0$ and $y \leq 0$ are path conditions. The nodes following a policy change are shown with dashed lines (nodes $n1$ and $n5$).

```
setPolicy(X → A, Y ↛ A);
output_A(x);
output_A(1);
output_A(1);
if (y > 0) then
    output_A(2);
setPolicy(X ↛ A, Y ↛ A);
if (y <= 0) then
    output_A(2);
output_A(3);
```

Program C.13

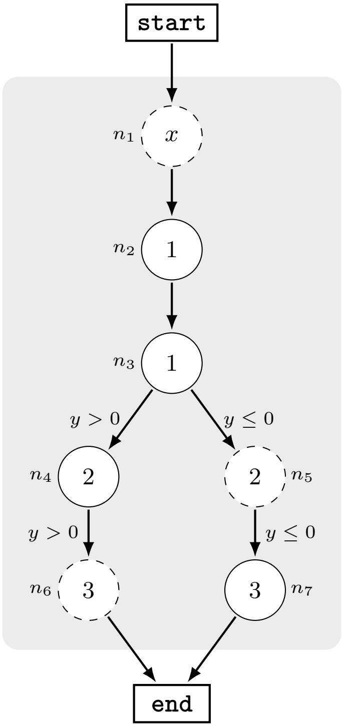


Figure C.4: SOT of Program C.13

This program is rejected by the Definition C.12, because the policy change between nodes $n3$ and $n5$ is inconsistent. The generated formula for node $n5$ is:

$$\Bigg(Pc_{n3}(\emptyset, \{x, y\}) \ \wedge \Bigg(\bigwedge_{n' \in N(S)} \neg\Big(Pc_{n'}(\emptyset, \{x', y'\}) \wedge \overrightarrow{O}_{n3}(\emptyset, \{x, y\}) = \overrightarrow{O}_{n'}(\emptyset, \{x', y'\})\Big)\Bigg)\Bigg)$$

The path condition of node $n3$ is *true* and its output sequence is $O_{n3} = (x, 1, 1)$. The formula is satisfiable if there exists a value for y or x where $Pc_{n3}(y)$ holds and *for all nodes* falsifies either the path conditions or the equality between outputs. The only node on the same level as $n3$ is $n3$ itself, which clearly means that the path condition is also *true*. However, since the output sequences are $(x, 1, 1)$ and $(x', 1, 1)$, it is sufficient to pick any value for x and x' such that $x' \neq x$ to satisfy the following formula. This implies that the policy change at node $n5$ is inconsistent.

$$\Big(true \ \wedge \neg\Big(true \wedge (x, 1, 1) = (x', 1, 1)\Big)\Big)$$

The following theorems show soundness of Definition C.11 and Definition C.12 wrt. the security conditions of Definition C.3 and Definition C.2, respectively.

Theorem C.2

Given a SOT S, if the formula in Definition C.11 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition C.3.

Theorem C.3

Given a SOT S, if the formula in Definition C.12 is unsatisfiable for all nodes $n \in S$ such that n follows a policy change, then S satisfies Definition C.2.

Forgetful Attacker

In line with the definitions of forgetful attackers in Section C.4, we ignore the actual values of the output expressions occurring before the last policy change. Therefore the output tuple $\overrightarrow{O}_n^{frg}$ for the forgetful attacker replaces all of the outputs that occurred before the last policy change with the constant value `1`. Additionally, while generating the inner conjunction of the formula, we only consider the nodes that have the same number of policy changes as n, using the auxiliary function $sameNP(S, n)$. Definition C.13 adapts Definition C.11 for forgetful attackers:

Definition C.13

Given an SOT S, policy $P_n(\overrightarrow{l}, \overrightarrow{h})$ at node n, S is secure wrt. forgetful attacker iff for all $n \in N(S)$, the following formula is unsatisfiable:

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n}) \ \wedge \Bigg(\bigwedge_{n' \in sameNP(S,n)} \neg\Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_n^{frg}(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}^{frg}(\overrightarrow{l_n}, \overrightarrow{h_n}') \Big) \Bigg) \Bigg)$$

It uses $\overrightarrow{O}_n^{frg}$ to compute output sequences, ignoring the values leaked before the policy change. Additionally, it only generates the formula for the nodes with the same number of policy changes. This is because the only relevant nodes for a forgetful attacker are the ones that are on the same epoch as the current node. Progress leaks are captured by the number of constant values in $\overrightarrow{O}_n^{frg}$ and the actual values leaked in the other epochs are ignored.

To illustrate this process, we revisit Program C.7 and its SOT in Figure C.5, and check the security for the forgetful attacker using Definition C.13. For example, at node $n5$ the generated formula is:

$$\left(Pc_{n5}(\emptyset, x) \ \wedge \left(\bigwedge_{n' \in sameNP(S,n)} \neg\left(Pc_{n'}(\emptyset, x') \wedge \overrightarrow{O}^{frg}_{n5}(\emptyset, x) = \overrightarrow{O}^{frg}_{n'}(\emptyset, x') \right)\right)\right)$$

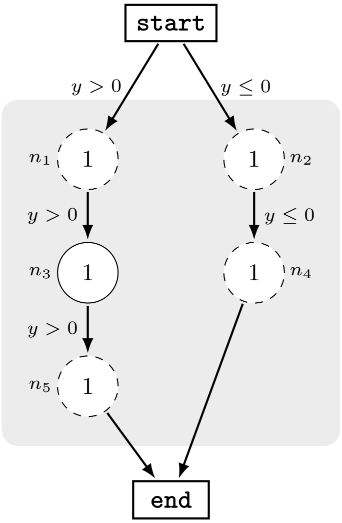


Figure C.5: SOT of Program C.7

The path condition of node $n5$ is $x > 0$ and its output sequence is $\overrightarrow{O}^{frg}_{n5} = (true, true, 1)$. The formula is satisfiable if there exists a value for x where $Pc_{n5}(x)$ holds, and *for all nodes* it falsifies either the path condition or the equality between outputs. The only node with the same number of policy changes and at the same level as $n5$ is $n5$ itself. This clearly means that the output sequences are equal. Therefore, to satisfy the formula we need a value x' that falsifies the path condition $x' > 0$, which can be any non-positive value. Thus, the formula is satisfiable and the program is insecure.

Theorem C.4 shows the soundness of Definition C.13 wrt. the security condition for the forgetful attacker.

Theorem C.4

Given a SOT S, if the formula in Definition C.13 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition C.3 for the forgetful attacker.

We refer to Appendix C.3 for proof sketches of the theorems, and Appendix C.1 for the verification algorithm of bounded memory attackers.

Policy Repair

This section we discuss an approach for generating repair policies. As discussed in Section C.4, a repair policy should ideally be the intersection between attacker knowledge and the new inconsistent policy. The approach presented here leverages the information provided by the SOT to calculate the attacker's knowledge at node n as a combination of the direct (outputs) and indirect (Pc values) observations made by the attacker. However, this approach does not always result in the intersection as defined by Definition C.9 as it sometimes over-approximates the attacker's knowledge. Developing a precise approach for generating consistent policies is left for future work.

When the policy consistency check fails at some node n, we can calculate the knowledge by traversing the SOT from n up to node *start* and collecting the attacker's explicit observations (i.e. the output expressions) and implicit observation (i.e. Pcs). It is important to note that not all output expressions and Pcs leak information, therefore we should only collect the ones that are *leaking* some information. Here we achieve this by applying a preprocessing step to the SOT, which uses a bounded memory attacker with memory capacity of $m = 1$ to determine the leaked expressions at each level of SOT. The addition of these leaked expressions to the new policy gives a candidate repaired policy.

The intuition behind this approach is that a new policy can only be inconsistent if it is more restrictive than the current policy. Thus a repaired policy should ease some of those restrictions. To this end, we extract already leaked expressions and add them to the inconsistent policy, which gives us the most restrictive version of policy that is consistent with the knowledge. After generating the repaired policy, we should also update the policy field of n's children up to the next policy change. This is because all of the nodes between n and the ones with a new policy are affected by the inconsistent policy.

C.7 Implementation and Evaluation

We implemented the algorithms presented in Section C.6 by extending ENCoVer [91] and creating a prototype dubbed DynCoVer [188]. Like ENCoVer, DynCoVer relies on Symbolic PathFinder (SPF) [102], an extension of Java PathFinder (JPF) [30], to concolically execute programs and extract the symbolic output trees from Java bytecode.

DynCoVer analyzes the program by means of concolic testing and does the following in a loop to explore all execution paths of the program and generate the SOT: it starts with concrete and symbolic values for input variables and executes the program concolically to collect each step's path condition. These conditions are then passed to a constraint solver to generate new inputs that explore different paths. Upon reaching an output statement, the output expression is evaluated in the symbolic state and a new node representing the result of that evaluation is added to the SOT. The path condition that directed the program to this output statement is also saved in the node.

After generating the SOT, DynCoVer traverses the tree using a depth-first search (DFS) strategy, and for each node, depending on the attacker, it generates the formulas described in Section C.6. Then, DynCoVer feeds the generated formula to a satisfiability modulo theory (SMT) solver (Z3 in the current implementation). If the SMT solver answers that the formula is satisfiable, then the analyzed program is deemed insecure. DynCoVer repeats this process for all of the nodes in SOT and if the SMT solver's answer was unsatisfiable for all nodes, the program is accepted as secure.

For the perfect recall and bounded memory attackers, DynCoVer also checks the policy consistency for all nodes that are marked as "nodes following a policy change". DynCoVer uses Definition C.12 for generating the policy consistency check formula, feeds it to the SMT solver, and if the result was unsatisfiable, it deems the policy change as consistent, and moves on to checking the security on that node.

DynCoVer also supports policy repair by relying on the heuristic of Section C.6. If configured in repair mode, DynCoVer performs preprocessing on the SOT and identifies leaking expressions. Upon reaching an inconsistent policy, it shows a warning message, proceeds to generate the repaired policy, and prints it to the user.

DynCoVer uses ENCoVer [91] as a basis and extends it with support for dynamic polices, policy consistency checks, and policy repair. DynCoVer is approximately 6 KLOC as computed by CLOC and includes nearly 86 classes/interfaces. Like ENCoVer, the class of programs that DynCoVer can handle is indirectly limited by the class of programs SPF (JPF core and its symbolic extension) can handle and the class of constraints Z3 can solve.

Case Studies

To evaluate the effectiveness and efficiency of DynCoVer, we carried out two different experiments. First, we created a micro benchmark suite to facilitate checking and understanding dynamic policy scenarios and different types of attackers. Second, we implemented and verified the core of a social network to demonstrate the effectiveness of our security conditions in a real-world scenario.

These case studies target three objectives: (1) to validate that the results of DynCoVer are in line with the conditions in Section C.4; (2) to ensure that the policy repair heuristic works as expected; (3) to evaluate the performance of DynCoVer.

Benchmark

Our benchmark consists of 25 programs, including programs from the paper, to demonstrate different aspects of dynamic policies. It includes programs with various constructs, loops, and implicit leaks. This benchmark is implemented in Java, and

Table C.1: Benchmark evaluation results

	DynCoVer	Attacker	Inconsistent	JPF	**Time (ms)**					SOT
	Result	Type	Policy Mode	Inst	OA	ME	FG	FS	PR	Nodes
Program 3	↯	Perfect	Reject	2940	219.5	8.7	1.5	31.6	–	2
Program 3	✓	Perfect	Repair	2940	231.6	8.8	0.6	31.5	20.4	2
Program 3	✓	Forgetful	–	2940	217.4	8.7	1.6	31.4	–	2
Program 7	×	Perfect	Reject	2964	316.4	38.4	2.6	84.1	–	5
Program 7	×	Bounded	Reject	2964	301.0	38.6	2.5	84.3	–	5
Program 7	×	Forgetful	–	2964	298.2	38.5	2.3	69.0	–	5
Program 13	↯	Perfect	Reject	2982	292.7	49.5	1.9	63.3	–	7
Program 13	✓	Perfect	Repair	2982	367.1	49.0	1.9	91.5	47.7	7
Program 13	✓	Bounded	Reject	2982	341.6	47.0	3.6	111.5	–	7
Program 13	✓	Forgetful	–	2982	300.0	48.6	2.9	68.2	–	7

DynCoVer Results: ✓ the program is secure; × the program is insecure; ↯ the program has an inconsistent policy change.
Inconsistent Policy: What to do when facing an inconsistent policy: *Reject* the inconsistent policies; *Repair* the policy.
JPF Inst: total number of instructions executed by JPF
Time: *OA*: overall; *ME*: model extraction ; *FG*: interference formula generation; *FS*: interference formula satisfiability checking; *PR*: policy repair (only if applicable)
SOT Nodes: Number of nodes in the generated Symbolic Output Tree

the only use of non-standard command is the `setPolicy` method, indicating a policy change. Each program has a configuration file which defines the attacker type, its memory capacity, and the method used to deal with inconsistent policies. These `.jpf` configuration files are used by JPF's virtual machine and DynCoVer to verify the program.

Table C.1 shows an excerpt of programs from the benchmark, Table C.3 in the Appendix contains more programs. As we can see in column "DynCoVer Result", DynCoVer rejects insecure programs and accepts secure ones, in line with the definitions in Section C.4. In addition to the results for security and policy consistency, Table C.1 also reports some information about the efficiency and performance of DynCoVer. For simple programs with small number of outputs, the SOT size and the evaluation time of DynCoVer is low. But when testing more complex programs with multiple loops and outputs, the performance decreases. Memory usage of DynCoVer is around 235 MB for simple programs, and starts to increase when the number of loops and instructions increases.

Social Network

In this case study, we implemented a social network that simulates the interactions between users, and contains some of the main functionalities of a social network, such as following, unfollowing, blocking other users, sending DMs, and creating groups and events. Users also have privacy settings and change their setting to hide their sensitive information such as phone number. The high level of interactivity

Table C.2: Social network evaluation results

	DynCoVer	Attacker	Inconsistent	JPF	Time (ms)					SOT
	Result	Type	Policy Mode	Inst	OA	ME	FG	FS	PR	Nodes
postForFollowers	↯	Perfect	Reject	13953	659.4	270.5	11.5	168.0	–	24
postForFollowers	✓	Forgetful	–	13953	795.6	293.3	12.4	160.3	–	24
blockingUser	↯	Perfect	Reject	14133	656.3	274.5	10.5	163.5	–	25
blockingUser	✓	Forgetful	–	14133	599.4	226.6	11.5	160.1	–	25
forwardingDM	×	Perfect	Reject	13037	519.0	169.5	4.2	154.1	–	12
forwardingDM	×	Bounded	Reject	13037	499.5	166.8	4.4	145.0	–	12
forwardingDM	×	Forgetful	–	13037	495.9	164.4	4.1	130.0	–	12
phoneNumberPrivacy	✓	Perfect	Reject	13135	656.6	181.0	14.0	261.5	–	21
phoneNumberPrivacy	✓	Forgetful	–	13135	680.7	212.7	9.8	242.9	–	21
leakMembership	✓	Perfect	Reject	18569	747.0	286.8	10.1	232.0	–	19
leakMembership_leave	↯	Perfect	Reject	18850	802.2	314.9	8.7	283.8	–	21
leakMembership_leave	✓	Forgetful	–	18850	769.0	301.8	13.0	231.7	–	21
leakEventInfo	✓	Perfect	Reject	10676	427.5	159.5	2.0	73.9	–	4

between entities in a social networks makes it a good candidate for dynamic policy analysis.

A social network is naturally an interactive program, whose behavior is determined by the actions of different users of the system. To model these behaviors and make them amenable to extract the SOT of the program with DynCoVer, we implemented an additional program which simulates the behavior of different users involved in the execution of the interactive program.

The Java implementation of the social network has 5 classes and 659 LOCs. There is one class for each of the entities: server, user, group, event, and post. To evaluate this case study, 6 different scenarios have been implemented and examined. The results of these experiments are reported in Table C.2.

In the `postForFollowers` scenario, users interact by following each other and creating posts. The goal here is to ensure that a post is only visible to the followers of a certain user. This scenario is secure for a forgetful attacker, and inconsistent for a perfect recall attacker. This is because the ex-follower has already seen some of the unfollowed user's posts. The second scenario is similar, but this time a user can block their followers. Similarly, this program is secure for a forgetful attacker and inconsistent for a perfect recall attacker. The policy repair mode does not make sense for these scenarios, because after unfollowing or getting blocked, the observer should no longer be able to see the user's old posts.

The next scenario simulates forwarding a user's DM to another user. This scenario is insecure for all types of attackers, because a third user should not be able to see other users' DMs. The `phoneNumberPrivacy` scenario checks the privacy settings of a user. Initially, the user's information such as the phone number is private. However, a user can changes their privacy setting to make the phone number public. The goal here is to make sure that users cannot see other users' private information. This scenario is secure for both perfect recall and forgetful attackers.

Next, we consider a scenario in which a user's membership in a group should be kept secret from all users that are not in that group. A user cannot see the group members until they are added to that group. This scenario is secure for all three types of attackers. Now, if we change this scenario in such a way that a users leaves the group after learning the names of its members, then the program is inconsistent for perfect recall and secure for forgetful attackers. In the last scenario we consider events. Here a user should not be able to see an event's information such as its title or date unless they are invited to it. The program is secure for this scenario.

C.8 Related Work

This section discusses closely related works targeting dynamic policies and information flow control. We refer to Broberg et al. [114] for a survey on dynamic policies.

Our security framework is inspired by the work of Askarov and Chong [88] on knowledge-based security conditions for dynamic policies. They propose a general framework for capturing the semantics of dynamic policies for all attackers, showing that secure program under perfect recall attacker can be insecure under a weaker attacker. We revisit and extend their framework to accommodate three realistic attacker models and point out the challenges with policy consistency. Because the notions of perfect recall attacker and bounded memory attackers have well-defined interpretations in applications and epistemic logics [16], security conditions should be specific about the attacker model and uncover inconsistent policies. This allows us to show that the security of a program under a stronger attacker implies security under weaker attackers. Moreover, we propose a security condition for forgetful attackers to capture transient release of sensitive information, instantiating the framework of Askarov and Chong. On the enforcement side, DynCoVer uses automated theorem proving while Askarov and Chong design a security type system, each targeting well-known trade-offs between precision and scalability. Van Deft et al. [116] improve the framework of Askarov and Chong with regards to progress-insensitive security, showing how a type system enforces security against all attackers. By contrast, our conditions are progress sensitive, while progress insensitivity can be accommodated following Van Deft et al. [116]. Broberg et al. [114] illuminate the different facet of dynamic policies proposed in the literature [22, 40, 49, 56, 64, 98, 163]. We revisit and discuss this facets in our framework by developing the corresponding security conditions or pointing out mismatches. Other works addressing the challenge of flexible policies include Chudnov and Naumann [152] framework for downgrading policies in reactive programs, Lu and Zhang's [173] framework for non-transitive policies, and Kozyri and Schneider's [171] reactive labels.

Recently, Li and Zhang [184] propose a general-purpose framework for dynamic policies. Their approach categorizes dynamic policies into persistent and transient policies, and uses the notion of effective traces to define a unified knowledge-based security condition. By contrast, our work departs from existing works on dynamic policies, by focusing on an attacker-centric approach and tries to address the issues of dynamic policies through policy consistency and the relation between attacker power and policy.

Our enforcement techniques build on the line of work on verifying static information flow policies by automated theorem proving [39, 82, 90, 95, 135]. To our best knowledge, none of these works addresses either dynamic policies or the issues of policy consistency and policy repair. A precursor of our approach is the work of Balliu et al. [90] which also uses Java Pathfinder to extract program dependencies and verify static noninterference policies by symbolic execution. Our verification conditions are similar to Balliu et al. [90] for static policies, and develop them further to accommodate dynamic policies. Paragon [133] extends Java with support for dynamic policies using a security type system [64, 72] and it enforces security for the perfect recall attacker. By contrast, DynCoVer additionally supports bounded memory and forgetful attacker models with policy consistency and repair. Other languages and tools supporting information flow control for perfect recall attackers include JIF [19], LIO [87], Jeeves [96], and JOANA [111].

C.9 Conclusion

We have revised knowledge-based security conditions for dynamic policies and proposed attacker-centric conditions for perfect recall, bounded memory, and forgetful attackers. Drawing on the notion of policy consistency, we studied the relationship with the different facets of dynamic policies as well as policy repair. To verify and repair dynamic policies under different attacker models, we designed, implemented, and evaluated DynCoVer, an open source tool based on symbolic execution and SMT solving. An interesting avenue for future work is to investigate the interplay between integrity and confidentiality for dynamic policies [34, 71, 80, 134].

Acknowledgments

We thank Roberto Guanciale and anonymous reviewers for the feedback. This work is partially supported by the JointForce project funded by Swedish Research Council (VR), Swedish Foundation for Strategic Research (SSF), Facebook, and Digital Futures.

Appendices

Appendix A Bounded Memory Attacker

The unlimited memory of perfect recall attacker means that it can remember everything it once observed. A bounded memory attacker is a variant of perfect recall attacker with a limited memory (called m hereafter). After observing m outputs, its memory will become full and in order to capture any new outputs the oldest observation should be removed from it (in a FIFO manner).

Security Policies

The security condition used for bounded memory attacker is similar to Definition C.4, except that when computing the knowledge of a bounded memory attacker, we have to consider its memory capacity (m) as well as the number of outputs it has observed. If the number of outputs is less than m, the attacker is going to behave just like prefect recall, and if it is more than m, the attacker is going to only remember the last m observations.

With this intuition, we can define the auxiliary function *suffix(t,m)* which takes a trace t and returns the last m events of the trace:

Definition C.14

Given trace t as $t = \alpha_1.\alpha_2...\alpha_k$, suffix(t,m) is defined as:

$$\text{suffix(t,m)} = \begin{cases} t & \text{if } k \leq m \\ \alpha_{k-m}...\alpha_k & \text{if } k > m \end{cases}$$

Now, we can define the knowledge of a bounded memory attacker at execution point i.

Definition C.15

Program c with initial store σ, and initial policy p_{init} produces trace t after i execution steps, i.e. $\langle c, \sigma, p_{init} \rangle \xRightarrow{t}_i$. We write $k_i^{bnd}(c, \sigma, p_{init}, A, m)$ for the knowledge of an attacker that observes the outputs of this program on channel A and has a bounded memory capacity of length m, and define it as follows:

$$k_i^{bnd}(c, \sigma, p_{init}, A, m) = \{\sigma' \mid \langle c, \sigma, p_{init}\rangle \overset{t'}{\Longrightarrow}_j$$
$$\wedge\ \text{suffix}(t\lfloor_A, m) = \text{suffix}(t'\lfloor_A, m)\}$$

By adapting Definitions C.2 and C.3 to $k_i^{bnd}(c, \sigma, p_{init}, A, m)$ for attacker knowledge, we can use Definition C.4 to check security for bounded memory attackers as well.

Verification of Dynamic Policies

Since a bounded memory attacker is a special type of perfect recall with limited observations, the verification approach used for checking a program's security against a bounded attacker is also similar to the verification for a perfect recall attacker. We just have to modify Definition C.11 to account for the limited memory of the attacker.

Definition C.16

Given an SOT S, active policy $P_n(\overrightarrow{l}, \overrightarrow{h})$ at node n, S is secure wrt. bounded memory attacker with memory capacity m iff for all nodes $n \in N(S)$, the following formula is unsatisfiable:

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n}) \ \wedge \Bigg(\bigwedge_{n' \in N(S)}$$
$$\neg\Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_n^m(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}^m(\overrightarrow{l_n}, \overrightarrow{h_n}')\Big)\Bigg)\Bigg)$$

where $\overrightarrow{O}_n^m$ is a tuple of the **last** m output expressions encountered on a path in SOT from node *Start* to node n, $\overrightarrow{O}_n^m = \overrightarrow{O}_{n'}^m$ denotes the component-wise equality between two tuples, and $N(S)$ is the nodes of SOT S.

In this definition we limit the length of the attacker's observations by m, which is the capacity of his memory. In other words, $\overrightarrow{O}_n^m$ contains the last m outputs observable by the attacker. Therefore by checking $\overrightarrow{O}_n^m(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}^m(\overrightarrow{l_n}, \overrightarrow{h_n}')$ for all $n' \in N(S)$ we are looking for nodes that can – from the attacker's perspective – produce the same trace of outputs.

The process of checking a program's policy consistency for a bounded memory attacker is similar to that of a perfect recall attacker (Definition C.12); the only difference is that we should use $\overrightarrow{O}_n^m$ instead of $\overrightarrow{O}_n$ during the generation of the formula for checking consistency.

To illustrate this, we revisit Program C.13 and check its policy consistency under a bounded memory attacker with memory capacity of $m = 2$. For example, the generated formula for node $n5$ is:

$$\Bigg(Pc_{n3}(\emptyset, \{x, y\}) \ \wedge \Bigg(\bigwedge_{n' \in N(S)} \neg\Big(Pc_{n'}(\emptyset, \{x', y'\}) \\ \wedge \overrightarrow{O}_{n3}^2(\emptyset, \{x, y\}) = \overrightarrow{O}_{n'}^2(\emptyset, \{x', y'\})\Big)\Bigg)\Bigg)$$

The output sequence of node $n3$ is $O_{n3}^2 = (1, 1)$. The formula is satisfiable if there exists a value for y or x where $Pc_{n3}(y)$ holds, and *for all nodes* it falsifies either the path conditions or the equality between outputs. For this attacker, it is possible for the nodes which are not on the same level to have equal outputs, so we have to consider all $n' \in N(S)$. However, in this example, there are two nodes that can possibly produce an output sequence equal to $(1, 1)$. $n2$ with output sequence $(x, 1)$ and $n3$ with $(1, 1)$, for both of which the path condition is also *true*. We consider $n3$ first, because in this case both of the output sequences are $(1, 1)$, which means that the inner formula is true, hence the result of conjunction is false. This means that even without considering $n2$ we can be conclude that the whole formula is unsatisfiable and the policy change at node $n5$ is consistent. Similarly, we can apply Definition C.16 to all of the nodes in S and show that Program C.13 is in fact secure.

The following theorem shows soundness of Definition C.16 wrt. the security condition for a bounded memory attacker.

Theorem C.5

Given a SOT S, if the formula in Definition C.16 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition C.3 for the bounded memory attacker.

Proof. Similar to the proof of Theorem C.2. □

Appendix B Examples for the Forgetful Attacker

In this section, we revisit some of the examples presented in Section C.4 to demonstrate how a forgetful attacker's knowledge is calculated.

For Program C.5 we calculate the attacker's knowledge after the output of line 8. Without the loss of generality, let us assume $\texttt{x} = 5$ and $\texttt{y} = 7$. At this point, the attacker has observed the trace $t = 7.1.2$. There is also an unobservable new policy event between 1 and 2, thus the *splitPolicy(t)* function gives us the sub-traces $(7.1, 2)$. The length of the sub-trace $t_1 = 7.2$ is 2, so the knowledge of attacker will be all of the stores that can produce a trace ending with 2 that has exactly two other observable events (of any value) before that:

$$\begin{aligned} k_8^{frg}(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init}\rangle &\overset{t''}{\Longrightarrow}_j \\ &\wedge\ (t_1'', t_2'') = \text{split}(t''\lfloor_A, 2) \\ &\wedge\ t_2'' = 2\} \end{aligned}$$

This corresponds to all the stores with any value for x, and since the active policy p at execution point $i = 8$ is also the set of all the stores, security condition C.3 holds. If we repeat this process for all execution points, we can see that the program is accepted by Definition C.3.

Similarly, the knowledge of the attacker at line 8 of Program C.6 will be all of the stores that can produce a trace that ends with sub-trace 7.2 and have exactly one other observable event before that, which will be the stores with any value for x, but *only* value 7 for y. Since the active security policy at this point is the set of all stores with any value for both x and y, the security condition $p_8 \subseteq k_8^{frg}(c, \sigma, p_{init}, A)$ *does not* hold and Program C.6 is rejected as insecure.

In Program C.7, for a positive x, the attacker observes the trace $t = 1.1.1$ after the output on line 8. The *splitPolicy(t)* function gives us the tuple $(1.1, 1)$ and since $\mid 1.1 \mid$ is 2, the attacker knowledge will be all of the stores that can produce a trace that ends with 1, and have exactly two other observable event before that:

$$\begin{aligned} k_8^{frg}(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init}\rangle &\overset{t''}{\Longrightarrow}_j \\ &\wedge\ (t_1'', t_2'') = \text{split}(t''\lfloor_A, 2) \\ &\wedge\ t_2'' = 1\} \end{aligned}$$

The only stores that can produce such a trace are the ones with `x > 0`, which implies that the security condition *does not* hold and the program is rejected as insecure.

Appendix C Proofs

Proof of Verification Soundness

Here we present sketches for the proofs of Theorems C.2, C.3, and C.4.

Theorem C.2

Given a SOT S, if the formula in Definition C.11 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition C.3.

Proof. By Definition C.3, a program is insecure if there is a point i during the execution in which the policy is not contained in the knowledge (i.e. $p_i \not\subseteq k_i$). This means that there is a value such that it is in the policy but not in the knowledge. Similarly, in the SOT, if the formula corresponding to a node n is satisfiable, it means that there exist two stores that satisfy the policy $P(\overrightarrow{l}, \overrightarrow{h})$, but either one store does not reach the output node or the two stores produce different output sequences. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_i$, thus violating the security condition.

On the other hand if the formula at node n is unsatisfiable, it implies that for any initial state $(\overrightarrow{l}, \overrightarrow{h})$ that satisfies the policy $P(\overrightarrow{l}, \overrightarrow{h})$ and reaches node n producing the output sequence $\overrightarrow{O}_n$, it is impossible to find another state $(\overrightarrow{l}, \overrightarrow{h}')$ that satisfies the policy and reaches some output node n' (i.e. $n \in N(S)$ and $Pc_{n'}(\overrightarrow{l}, \overrightarrow{h}'))$ yielding a different output sequence $\overrightarrow{O}_{n'}$. If we repeat this process for all nodes $n \in S$, and none of their formulas are satisfiable, it means that there are no outputs in SOT S such that $p_i \not\subseteq k_i$. □

Theorem C.3

Given a SOT S, if the formula in Definition C.12 is unsatisfiable for all nodes $n \in S$ such that n follows a policy change, then S satisfies Definition C.2.

Proof. The proof of this theorem is similar to Theorem C.2. However, because in the SOT we do not have any nodes for the policy change, we have to capture the policy changes on the next output nodes.

In the policy consistency check formula (Definition C.12) the policy part of the formula $(P(\overrightarrow{l}, \overrightarrow{h}))$ is generated at node n because we want it to reflect the new policy, however, the rest of the formula is generated for n's parent node (i.e. $parent(n)$). The satisfiability of this formula means that there exist two stores that satisfy the new policy $P(\overrightarrow{l}, \overrightarrow{h})$ at node n, but either one store does not reach an output or the two stores produce different output sequences up to $parent(n)$. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P_n(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_{i-1}$, thus violating the policy consistency condition.

If we repeat this process for all nodes $n \in S$ with new policy, and none of their formulas are satisfiable, it means that there are no policy changes in SOT S such that $p_i \not\subseteq k_{i-1}$. Here we assume that there is always an output after a policy change. For programs that have a policy change as their last command, we can use the node *End* for policy consistency check and apply Definition C.12. □

Theorem C.4

Given a SOT S, if the formula in Definition C.13 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition C.3 for the forgetful attacker.

Proof. The difference between the the security condition of the forgetful attacker and the perfect recall is that the latter uses Definition C.7 to calculate the attacker's knowledge.

This definition limits the observations of the attacker to the number of outputs before the policy change and the values of outputs after a policy change. The output function $\overrightarrow{O}_n^{frg}$ used in the forgetful attacker's formula (Definition C.13) captures this behavior by ignoring the value of outputs before the last policy change (replacing them with a constant value), and only keeping the actual value of the outputs that happened after the policy change.

The rest is similar to the proof of Theorem C.2. The satisfiability of the formula of Definition C.13 means that there exist two stores that satisfy the new policy $P(\overrightarrow{l}, \overrightarrow{h})$ at node n, but if they reach an output, the output sequences up to node n wrt. $\overrightarrow{O}_n^{frg}$ will be different. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P_n(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_i^{frg}$, thus violating the security condition condition. □

Proof of Attacker Power

In this section, we present Theorem C.6 to prove the claim that in the absence of inconsistent policy changes, a program which is secure against a perfect recall attacker is also secure against a bounded memory attacker.

Theorem C.6

Given a program c with no inconsistent policy changes, initial store σ, and initial policy p_{init}, if for all execution points i, c is secure against perfect recall attacker A_{per}, it is also secure against bounded memory attacker A_{bnd}^m with memory capacity $m \in N$. Formally:

$$p_i \subseteq k_i(c, \sigma, p_{init}, A_{per}) \implies$$
$$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A_{bnd}^m, m) \quad \forall m \in N$$

Proof. We should consider all execution points i such that

$$\langle c, \sigma, p_{init} \rangle \overset{t}{\Rightarrow}_i \langle c_i, \sigma_i, p_i \rangle$$

and continue with structural induction on command c_i.

All of the commands presented in Figure C.1 should be considered here. However, not all of them have an effect on the knowledge, therefore we only investigate command $\texttt{output}_\ell(e)$. Without the loss of generality let us assume that e is visible to the attacker A^m_{bnd}.

Since knowledge is monotone, the more observations an attacker has, the smaller his knowledge set will be. We can use this fact to limit the number of cases we have to investigate for different values of m. Thus, we only consider two scenarios:

- if $\mid (t.\alpha)\lfloor_{A^m_{bnd}} \mid \leq m$ then the bounded memory attacker with memory capacity m is going to know everything that the perfect recall attacker knows. Hence

  $$k_i(c, \sigma, p_{init}, A_{per}) = k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

  and since by assumption we know that $p_i \subseteq k_i(c, \sigma, p_{init}, A_{per})$, we can conclude:

  $$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

- if $\mid (t.\alpha)\lfloor_{A^m_{bnd}} \mid > m$ then the bounded memory attacker A^m_{bnd} had less observations than the perfect recall attacker. Hence

  $$k_i(c, \sigma, p_{init}, A_{per}) \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

  and since by assumption we know that $p_i \subseteq k_i(c, \sigma, p_{init}, A_{per})$, we can conclude that

  $$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

  also holds.

As a result, the security condition:

$$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m) \quad \forall m \in N$$

holds for all values of $m \in N$.

Additionally, if c_i is $\texttt{setPolicy}(p')$ we can use the assumption that program c does not have any inconsistent policies to conclude that Definition C.2 holds for perfect recall attacker A_{per} for all execution points i. Since we already established that bounded memory attacker A^m_{bnd}'s knowledge is less than or equal to A_{per} at each execution point, it is straightforward to show that:

$$p_i \subseteq k^{bnd}_{i-1}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

which means that the policy changes are also consistent for bounded memory attacker A^m_{bnd}. □

Theorem C.1 proves a similar claim for the forgetful attackers.

Theorem C.1

Given a program c, initial store σ, and initial policy p_{init}, if for all execution points i, c is secure against perfect recall attacker A_{per}, it is also secure against forgetful attacker A_{frg}. Formally:

$$[\sigma]^{p_i}_A \subseteq k_i(c, \sigma, p_{init}, A_{per}) \implies [\sigma]^{p_i}_A \subseteq k^{frg}_i(c, \sigma, p_{init}, A_{frg})$$

Proof. The proof of this theorem is similar to Theorem C.6. We consider all execution points i such that:

$$\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_i \langle c_i, \sigma_i, p_i\rangle$$

and only investigate the case where command c_i is $\texttt{output}_\ell(e)$, and assume that e is visible to the attacker A_{frg}. Let us consider three scenarios:

- If *splitPolicy(t)* is (ϵ, t), then the observations of forgetful attacker are the same as the observations of the perfect recall attacker, hence:

  $$k_i(c, \sigma, p_{init}, A_{per}) = k^{frg}_i(c, \sigma, p_{init}, A_{frg})$$

  Since by assumption we know that $p_i \subseteq k_i(c, \sigma, p_{init}, A_{per})$, we can conclude:

  $$p_i \subseteq k^{frg}_i(c, \sigma, p_{init}, A_{frg})$$

- If *splitPolicy(t)* is (t, ϵ), then the forgetful attacker makes no observations after the policy change and only knows $\mid t \mid$. Thus,

  $$k_i(c, \sigma, p_{init}, A_{per}) \subseteq k^{frg}_i(c, \sigma, p_{init}, A_{frg})$$

  Hence, we can conclude that:

  $$p_i \subseteq k^{frg}_i(c, \sigma, p_{init}, A_{frg})$$

- If *splitPolicy(t)* is (t_1, t_2). Since knowledge is monotone and t_2 is a sub-trace of t, the knowledge set of an observer that sees t_2 is bigger that the knowledge set of the observer of t. Additionally, for all events before t_2, the attacker A_{per} observed the actual value of the event while the attacker A_{frg} only knows that an event has occurred. Therefore, it is straightforward to show that the knowledge of A_{frg} is less than the knowledge of A_{per}:

  $$k_i(c, \sigma, p_{init}, A_{per}) \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

  Therefore we can conclude that:

  $$p_i \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

□

Table C.3: Benchmark evaluation results (Extended Table)

	DynCoVer	Attacker	Inconsistent	JPF	**Time (ms)**					SOT
	Result	Type	Policy Mode	Inst	OA	ME	FG	FS	PR	Nodes
Program 1	↯	Perfect	Reject	2937	279.1	8.7	1.4	69.3	–	2
Program 1	✓	Perfect	Repair	2937	242.5	8.5	0.3	39.7	19.6	2
Program 1	×	Forgetful	–	2937	222.0	8.5	1.4	29.5	–	2
Program 2	↯	Perfect	Reject	2937	211.3	8.6	1.4	27.6	–	2
Program 2	✓	Perfect	Repair	2937	249.5	8.6	0.3	52.0	19.9	2
Program 2	✓	Forgetful	–	2937	210.8	8.5	1.4	25.7	–	2
Program 3	↯	Perfect	Reject	2940	219.5	8.7	1.5	31.6	–	2
Program 3	✓	Perfect	Repair	2940	231.6	8.8	0.6	31.5	20.4	2
Program 3	✓	Forgetful	–	2940	217.4	8.7	1.6	31.4	–	2
Program 4	×	Perfect	Reject	2937	229.2	6.8	1.6	41.3	–	2
Program 4	×	Forgetful	–	2937	216.5	7.2	1.5	31.5	–	2
Program 5	↯	Perfect	Reject	2972	252.4	46.8	1.5	32.0	–	5
Program 5	✓	Perfect	Repair	2972	342.9	48.7	1.3	74.3	47.2	5
Program 5	✓	Forgetful	–	2972	279.8	48.1	2.3	51.7	–	5
Program 6	×	Perfect	Reject	2972	294.3	48.9	2.1	48.4	–	5
Program 6	×	Bounded	Reject	2972	278.3	51.0	1.8	42.3	–	5
Program 6	×	Forgetful	–	2972	250.4	45.9	1.6	31.7	–	5
Program 7	×	Perfect	Reject	2964	316.4	38.4	2.6	84.1	–	5
Program 7	×	Bounded	Reject	2964	301.0	38.6	2.5	84.3	–	5
Program 7	×	Forgetful	–	2964	298.2	38.5	2.3	69.0	–	5
Program 8	×	Perfect	Reject	2938	214.2	6.7	1.3	14.8	–	1
Program 8	×	Perfect	Repair	2938	213.8	6.7	0.1	13.6	27.2	1
Program 8	×	Forgetful	–	2938	198.5	6.5	1.3	13.3	–	1
Program 9	↯	Perfect	Reject	2942	226.9	6.9	1.7	39.3	–	3
Program 9	✓	Perfect	Repair	2942	250.0	6.7	0.5	51.3	20.6	3
Program 9	×	Forgetful	–	2942	221.4	6.7	1.6	36.4	–	3
Program 10	×	Perfect	Reject	2937	224.2	6.4	1.6	39.0	–	2
Program 10	×	Forgetful	–	2937	211.6	7.0	1.5	28.9	–	2
Program 11	↯	Perfect	Reject	2940	218.1	8.6	2.4	23.9	–	2
Program 11	✓	Perfect	Repair	2940	330.8	9.4	0.4	39.2	29.1	2
Program 12	✓	Perfect	Reject	2954	265.5	37.8	2.0	52.3	–	4
Program 12	✓	Bounded	Reject	2954	269.3	37.8	2.0	49.3	–	4
Program 12	✓	Forgetful	–	2954	276.8	38.5	2.1	43.4	–	4
Program 13	↯	Perfect	Reject	2982	292.7	49.5	1.9	63.3	–	7
Program 13	✓	Perfect	Repair	2982	367.1	49.0	1.9	91.5	47.7	7
Program 13	✓	Bounded	Reject	2982	341.6	47.0	3.6	111.5	–	7
Program 13	✓	Forgetful	–	2982	300.0	48.6	2.9	68.2	–	7
WhileLoop_5	✓	Perfect	Reject	3393	705.6	96.2	26.7	385.9	–	30
WhileLoop_10	✓	Perfect	Reject	4093	1743.7	172.8	100.4	1226.4	–	85
WhileLoop_50	✓	Perfect	Reject	20493	198030	1797	68486	126451	–	1425

DynCoVer Results: ✓ the program is secure; × the program is insecure; ↯ the program has an inconsistent policy change.
Inconsistent Policy: What to do when facing an inconsistent policy: *Reject* the inconsistent policies; *Repair* the policy.
JPF Inst: total number of instructions executed by JPF
Time: *OA*: overall; *ME*: model extraction ; *FG*: interference formula generation; *FS*: interference formula satisfiability checking; *PR*: policy repair (only if applicable)
SOT Nodes: Number of nodes in the generated Symbolic Output Tree

Paper D

Disjunctive Policies for Database-Backed Programs

Amir M. Ahmadian, Matvey Soloviev, and Musard Balliu

Abstract

When specifying security policies for databases, it is often natural to formulate *disjunctive* dependencies, where a piece of information may depend on at most one of two dependencies P_1 or P_2, but not both. A formal semantic model of such disjunctive dependencies, the Quantale of Information, was recently introduced by Hunt and Sands as a generalization of the Lattice of Information. In this paper, we seek to contribute to the understanding of disjunctive dependencies in database-backed programs and introduce a practical framework to statically enforce disjunctive security policies. To that end, we introduce the *Determinacy Quantale*, a new query-based structure which captures the ordering of disjunctive information in databases. This structure can be understood as a query-based counterpart to the Quantale of Information. Based on this structure, we design a sound enforcement mechanism to check disjunctive policies for database-backed programs. This mechanism is based on a type-based analysis for a simple imperative language with database queries, which is precise enough to accommodate a variety of row- and column-level database policies flexibly while keeping track of disjunctions due to control flow. We validate our mechanism by implementing it in a tool, DiVerT, and demonstrate its feasibility on a number of use cases.

D.1 Introduction

Database security and information flow security have largely evolved as two disparate areas [28, 38], while sharing closely-related foundations and mechanisms to enforce security. Modern applications commonly rely on shared database backends to provide rich functionality to a multitude of mutually distrusting users. In response to frontend demands, database query languages, with features such as triggers, store procedures, and user-defined functions, have increasingly come to resemble full-fledged programming languages, thus calling into question the adequacy of the underlying access control models [104, 128]. A *security policy* describes the totality of expectations that we have of a computer system in the face of adversaries that seek to satisfy objectives that may differ from ours. In the context of database systems, whose purpose is to retain and provide information, the security policies of interest constrain who is allowed to learn what parts of that information. A class of such security policies which has proven particularly challenging to enforce with the methods of database security are *disjunctive policies*, which states that given two pieces of information, some entity may either learn one *or* the other, but not both.

A common example of disjunctive policies are databases which contain personally identifiable information, such as medical trial data. Biometric parameters of participants are important confounders that must be considered when drawing conclusions from the data, but at the same time releasing too many parameters of any one participant (such as their height, age and weight) might be sufficient to deanonymize them with high confidence [26]. Hence, a security policy for such a database may specify that the user may learn height and age, or height and weight, or age and weight, but not all three. Other examples of scenarios where disjunctive policies are useful include differential privacy [45] and secret sharing.

In this paper, we combine insights from database security and information flow research to develop a formal model for reasoning about disjunctive information in database-backed programs, and thus take a step towards reconciling the two fields. Our model makes it possible to reason about the semantic information dependencies in a program that performs queries, and compare them against a disjunctive policy. Building upon this, we propose a provably sound static enforcement mechanism that ensure that the policy is satisfied.

It is customary in information flow models to represent information as an equivalence relation on states, with the refinement order of equivalence relations corresponding to having more information. This representation can be used for both the actual information conveyed by a computational process and the bound imposed on it as part of a simple, non-disjunctive security policy. The possible equivalence relations on a given universe of states form a structure called the *Lattice of Information* (LoI) [13], in which security-relevant questions can be answered, such as whether a program reveals no more information than is allowed by the security policy, or what information is revealed by the combination of two programs. Similar questions have been addressed in the database community using an analogous object called the Disclosure Lattice [99]. We observe that this definition is actually insufficient to characterize information, which motivates us to introduce a more specific structure based on query determinacy, the *Determinacy Lattice* (DL). The formal relation between the Disclosure Lattice or our definition and LoI was hitherto unexplored, and more importantly neither of them can be used to represent disjunctions as seen in our motivating example.

Recently, Hunt and Sands [181] proposed a new information flow structure called the *Quantale of Information* (QoI), which seeks to address this shortcoming and establish a formal setting for representing, combining and comparing disjunctions of information. We build upon this work to introduce an analogous structure, the *Determinacy Quantale* (DQ), representing disjunctive dependencies in database-backed programs. As we show, this structure can be formally related to the QoI, and this relationship is analogous to that between the LoI and the DL. We then use the DQ to design a knowledge-based security condition that relates disjunctive dependencies in database-backed programs to disjunctive policies.

We are the first to address the problem of enforcing disjunctive policies. Prior works that develop language-based enforcement techniques in database-backed applications do not support disjunctive policies, while database-level dependencies are restricted to coarse approximations that incorrectly reject secure programs, such as our previous example [52, 65, 122, 163, 165].

Perhaps unsurprisingly, path sensitivity of a static analysis is key to capturing disjunctive dependencies. We show how standard flow-sensitive type-based dependency analysis [116] can be adapted to a compositional path-sensitive analysis and thus capture disjunctive dependencies in terms of database queries. To represent these dependencies in the DQ model, we introduce a sound approximation of the information disclosed by each database query which is precise enough to represent complex combinations of both row- and column-level dependencies. Finally, in the DQ, the combination of these analyses can be proven sound with respect to our security condition. We expect that the overall architecture of the resulting soundness proof, in which we relate a sequence of abstractions of the behaviour of a program to ordered elements of the DQ, can be generalized to many other enforcement mechanisms for our security condition.

To demonstrate the practicality of our approach, we implement this type-based dependency analysis and query approximation for database-backed programs and evaluate it on a test suite and some use cases which effectively illustrate the need for disjunctive dependencies and disjunctive policies.

Summary of contributions.

- We introduce a formal model for reasoning about disjunctive dependencies and policies in databases. In the process, we show how to reconcile perspectives from the database security and information flow communities.
- We introduce a database-specific model of knowledge, the Determinacy Lattice, and a disjunctive extension, called the Determinacy Quantale, and explore their relationship to established general-purpose semantic models.
- Using our model, we define an extensional security condition for database-backed programs that accommodates disjunctive policies.
- We propose a type-based program analysis to capture disjunctive dependencies in database-backed programs, combine them with a novel abstraction of queries, and prove them sound with respect to our security condition. This is presented as an instance of a generalizable architecture for such soundness proofs.
- We implement a prototype tool that uses type-based dependency analysis and query approximation to verify query-based disjunctive policies for database-backed programs, and demonstrate its feasibility on a test suite and a number of use cases.

The rest of paper is structured as follows. After reviewing preliminaries in Section D.2, we give our account of the DL and introduce the DQ in Section D.3. In Section D.4, we formalize our model of database-backed programs and the security policies we impose on them, culminating in a formal security condition. We present enforcement mechanisms in Section D.5, and their implementation and evaluation in Section E.7. In Section E.8, we contextualize our contributions with a discussion of related work, and finally summarize conclusions in Section E.9.

D.2 Background

Lattice of Information

An equivalence relation $\sim \subseteq A \times A$ on a set A is a binary relation that is reflexive, symmetric, and transitive. For example, the equivalence relation `parity` on the set $A = \{0, 1, 2, 3\}$ is defined as $\{(x, y) \mid x, y \in A \wedge x \; mod \; 2 = y \; mod \; 2\}$. An equivalence relation partitions its underlying domain into disjoint equivalence classes. Given an equivalence relation P on a set A and $a \in A$, $[a]_P$ denotes the unique equivalence class induced by P that a belongs to. We write $[P]$ to denote the set of all equivalence classes induced by P. We call $[P]$ a *partition* of A and hereafter we may also refer to each element, i.e. equivalence class, of the partition $[P]$ as a *cell*. For example, `parity` partitions A into cells $\{0, 2\}$ and $\{1, 3\}$.

Equivalence relations over states are commonly used to represent an agent's knowledge, by relating two states whenever the agent cannot distinguish between them. When an equivalence relation models knowledge, we also call the cells induced by it *knowledge sets*. These have a distinct intuitive interpretation when we consider functions f that take in some state and return an agent's *view* of it. We will write the equivalence relation induced by the output of f as $\sim_f = \{(x, y) \mid f(x) = f(y)\}$. In that case, in a state a, the knowledge set $[a]_{\sim_f}$ represents the agent's remaining uncertainty about the state, in the sense of all the states that the agent still considers possible, after observing the output of f. The agent *knows* anything that is true in all states in the knowledge set. In this paper, we use the terms knowledge and information interchangeably.

A complete lattice is a set equipped with a partial ordering (reflexive, antisymmetric, and transitive) relation, maximal and minimal elements $\top$ and $\bot$ for this relation and a join (least upper bound) for any subset of elements. The meet (greatest lower bound) of a subset can be defined as the join of the set of all lower bounds of that subset [21]. The *Lattice of Information (LoI)* [13] is a structure for representing the ordering of information with equivalence relations. Let $\mathcal{L}(A)$ be the set of all equivalence relations defined on a given domain A. The LoI ranks these equivalence relations based on the information they reveal about the underlying domain. Given two equivalence relations $P, Q \in \mathcal{L}(A)$, this ordering can be defined as follows:

$$P \sqsubseteq Q \to \forall a, a' \in A \ \ (a \ Q \ a' \Rightarrow a \ P \ a')$$

For any set $S \subseteq \mathcal{L}(A)$, the least upper bound of S is the equivalence relation R defined as:

$$\forall x, y \in A \ (x \ R \ y \leftrightarrow \forall P \in S. \ x \ P \ y).$$

Formally, $LoI(A) = \langle \mathcal{L}(A), \sqsubseteq, \bigsqcup \rangle$ denotes the LoI on domain A, with ordering relation $\sqsubseteq$ and join $\bigsqcup$. The top element $\top$ in the lattice is the most precise equivalence relation `id` such that `id` $= \{(x, y) \mid x, y \in A \wedge x = y\}$, and the bottom element $\bot$ is the least precise equivalence relation `all` $= \{(x, y) \mid x, y \in A\}$.

The join of any two equivalence relations $P \sqcup Q$, being their least upper bound, is the *least* informative equivalence relation that is at least as informative as either of P and Q (i.e. is an *upper bound* on both), and thus represents the information that is conveyed from learning both P *and* Q. We refer to this as the *conjunction* of the information in P and Q.

Quantale of Information

The LoI captures the conjunction of any two information sources P and Q as the join of their respective equivalence relations. However, it does not offer an operator that would yield a representation of their *disjunction*, that is, the information that can be obtained from having access to one of them, but not both. In fact, the disjunction can not in general be represented as a single equivalence relation, and thus an element of the LoI, at all. To address this limitation, Hunt and Sands [181] propose a generalization of the LoI called the *Quantale of Information* (QoI). A quantale is a complete lattice with an additional binary "tensor" operator $\otimes$. In the QoI, the tensor is used to represent conjunction, while the lattice join represents *dis*junction.

The core idea behind the quantale structure is to interpret the disjunction $P_1 \vee \ldots \vee P_n$ of several knowledge relations as describing all knowledge relations R in which the knowledge always comes from one of the P_i. More concretely, in any possible state $a \in A$, the agent's knowledge $[a]_R$ should equal its knowledge in the same state in one of the disjuncts, $[a]_{P_i}$. *Which* disjunct it is may depend on the state, so the agent may have knowledge from P_i in the state a but knowledge from P_j in some other state a'. Relations R that satisfy this condition are called *tilings*, based on a picture of covering (since every state needs to be in some equivalence class) the space of possible states A with knowledge sets drawn from any of the disjuncts. Following Hunt and Sands, we define the set of all tilings

$$\mathrm{mix}(\mathbb{P}) = \{R \in \mathit{LoI}(A) \mid x \in [R] \Rightarrow (\exists P \in \mathbb{P}.\, x \in [P])\},$$

where $\mathbb{P}$ is a set of equivalence relations.

We would like to think of a relation R' as describing no more knowledge than a disjunction $\bigvee \mathbb{P}$ if it's bounded above by *some* $R \in \mathrm{mix}(\mathbb{P})$ in the LoI, and more generally define the quantale ordering $\mathbb{S} \sqsubseteq \mathbb{T}$ for $\mathbb{S}, \mathbb{T} \subseteq \mathcal{L}(A)$ as $\forall S \in \mathbb{S},\ \exists T \in \mathbb{T}.\ S \sqsubseteq T$. The resulting relation is not antisymmetric on general sets of relations or even mixes of general sets, reflecting the circumstance that there may be multiple mixes representing the same knowledge. As it is standard in lattice theory [25], we use the downwards closure operator $\Downarrow$ to obtain canonical representations of the order cycles of $\sqsubseteq$ and hence construct a partial order.

$$\Downarrow\mathbb{P} = \{Q \in \mathit{LoI}(A) \mid Q \sqsubseteq \mathbb{P}\}$$

The *tiling closure* of a set of equivalence relations $\mathbb{P}$,

$$\mathrm{tc}(\mathbb{P}) = \Downarrow\mathrm{mix}(\mathbb{P}),$$

then canonically represents the knowledge permitted by the disjunction $\bigvee \mathbb{P}$. The set $\mathrm{tc}(\mathbb{P})$ can still be interpreted as a list of possible equivalence relations, now including any equivalence relation that does not reveal more information than the disjunction.

We then take the elements of the QoI on a state set A to be all tiling closures of subsets of A, with the ordering $\sqsubseteq$ being set inclusion. For the tensor $\mathbb{P} \otimes \mathbb{Q} = \mathrm{tc}(\{P \sqcup Q \mid P \in \mathbb{P}, Q \in \mathbb{Q}\})$, we rely on the join operator of the LoI $\sqcup$ to calculate the least upper bound of any possible pair of equivalence relations in $\mathbb{P}$ and $\mathbb{Q}$ and then canonicalise the result. Since the sets are interpreted disjunctively, the join $\bigvee_i \mathbb{P}_i$ can simply be defined as $\mathrm{tc}(\bigcup_i \mathbb{P}_i)$.

```
if (x <= 0) then
    out(-1  ,u);
    out(x mod 2 == 0, u);
else
    out(1, u);
    out(x div 2 == 0, u);
```

Program D.1

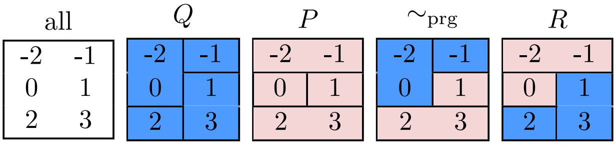


Figure D.1: Some equivalence relations on $\{-2, -1, 0, 1, 2, 3\}$

Example D.1

Program D.1 operates on a secret integer x between -2 and 3, outputting to user u whether it is greater than zero, and *either* (if it isn't) whether it is even, *or* (if it is) whether it equals 0 or 1 (by dividing by 2, rounding down and testing for 0). We expect the information released by the program ($\sim_{\mathrm{prg}}$ in Figure D.1) to be bounded by the disjunction of the knowledge relations capturing the two possible branches (resp. Q, P).
This could not be accurately expressed with LoI operations, since Q, P and $\sim_{\mathrm{prg}}$ are all incomparable, but the join of Q and P (as the only available nontrivial way of combining them) is equal to $\top$ and so would equally bound a program that directly releases x. However, $\sim_{\mathrm{prg}}$ can be tiled with equivalence classes from Q and P, and we in fact have $\mathrm{mix}(\{Q, P\}) = \{Q, P, R, \sim_{\mathrm{prg}}\}$. So in the QoI, $\mathrm{tc}(\{\sim_{\mathrm{prg}}\}) \sqsubseteq \mathrm{tc}(\{Q, P\})$, and hence $\sim_{\mathrm{prg}} \sqsubseteq Q \vee P$.

D.3 Information Ordering in Databases

Our goal is to introduce our semantic model for the information revealed by database queries, the *Determinacy Lattice*, and its extension to disjunctive dependencies, the *Determinacy Quantale*. To this end, we first review a standard formalism for reasoning about databases that we will employ.

A Primer on Relational Database Models

We use the relational model to formally define databases [15]. In this model, we distinguish between the database schema D, which specifies the structure of the database, and the database state db, which specifies its actual content.

A database schema D is a (nonempty) finite set of relation schemas t, written as $D = \{t_1, ..., t_n\}$. A relation schema (table) t is defined as a set of attributes paired with a set of constraints, where an attribute is a name paired with a domain. The number of attributes in t (written as $|t|$) is referred to as its arity. A tuple is a set of data representing a single record within a relation schema. Each tuple contains values for each attribute defined in the relation schema.

A *database state* db is a snapshot of the database schema D at a particular point in time. It represents the actual data stored in the database, consisting of a collection of tables and their respective tuples. We write $[\![t]\!]^{db}$ to represent the tuples of table t under database state db.

We write states(D) to denote the set of all database states of D. A database configuration is $\langle D, \Gamma \rangle$ where D is the database schema and Γ is a set of integrity constraints. We denote $\Omega_D = \{db \mid db \in \text{states}(D) \land \vdash db : \Gamma\}$ where $\vdash$ is an appropriate notion of constraint Γ being satisfied. An integrity constraint is an assertion about a database that must be satisfied for a database state to be considered valid. Various classes of integrity constraints exist, for instance functional dependencies which capture primary-key constraints, and inclusion dependencies which are used in foreign-key constraints [15].

Relational calculus. We rely on the Domain Relational Calculus (DRC) for our query language. In the DRC, a (non-boolean) query q over a database schema D has the form $\{\overline{x} \mid \phi\}$, where $\overline{x}$ is a sequence of variables, ϕ is a first order formula over D, and the free variables of ϕ are those in $\overline{x}$. The *evaluation* of a query q, denoted by $[\![q]\!]^{db}$, is the set of tuples that satisfy the formula ϕ with respect to db. A *boolean query* is written as $\{\ \mid \phi\}$, and its evaluation $[\![q]\!]^{db}$ is defined to be the boolean value `true` if and only if some tuple in db satisfies ϕ. We use $\mathcal{Q}$ to indicate the universe of all possible queries. The domain relational calculus employed here follows the standard convention, and we refer the reader to the relevant literature for a more comprehensive description of DRC [15].

emp :	`name`	`role`	`salary`
mng :	`division`	`manager`	

Figure D.2: Database schema for employees and managers

Example D.2

The database schema in Figure D.2 contains relations for employees emp and managers mng. A query returning the set of tuples containing the division names and the salary of the managers of each division can be written as:

$$\{(d, s) \mid \exists n, r.\ \text{emp}(n, r, s) \land \exists m.\ \text{mng}(d, m) \land n = m\}.$$

Views. In DRC, a database view is a relation defined by the result of a non-boolean query. Database views act as virtual tables and, as we will see, are useful when defining security policies. Formally, a view v defined over database schema D is a tuple $\langle id, q\rangle$, where id is the view identifier and q is the non-boolean query over schema D defining the view. The query q may refer to other views, but we assume that views do not have cyclic dependencies.

The materialization of a view v in a database state db is the evaluation of its defining query q in that state, i.e. $[\![q]\!]^{db}$. We use $v.q$ to refer to the defining query of view v. We extend relational calculus in the standard way to work with views [128].

Determinacy Lattice

Given query sets $Q, Q' \in \mathcal{P}(\mathcal{Q})$, query determinacy [76] captures whether results of the queries in Q are always sufficient to determine the result of the queries in Q'.

Definition D.1

Q determines Q' (denoted by $Q \twoheadrightarrow Q'$) iff for all database states db_1, db_2, if $[\![q]\!]^{db_1} = [\![q]\!]^{db_2}$ for all $q \in Q$, then $[\![q']\!]^{db_1} = [\![q']\!]^{db_2}$ for all $q' \in Q'$.

Intuitively, $Q \twoheadrightarrow Q'$ means that pairs of databases for which all queries in Q return the same result also give the same result under any query in Q'. This is in fact equivalent to the initial gloss that the results of queries in Q' can be computed from the results of queries in Q, as we show in detail in Appendix D.1.

Query determinacy allows us to define an ordering on sets of queries based on the information they reveal. We call this ordering *determinacy order*, denote it by $\preceq$, and define it as $\forall Q, Q' \in \mathcal{P}(\mathcal{Q})$, $Q \preceq Q'$ iff $Q' \twoheadrightarrow Q$.

Example D.3

Consider queries $q_1 = \{(n, r) \mid \exists s.\ \text{emp}(n, r, s)\}$ and $q_2 = \{(r) \mid \exists n, s.\ \text{emp}(n, r, s)\}$ defined on the relations of Figure D.2. Query q_1 discloses the name and the role of the employees while q_2 only returns their role. Intuitively, q_1 reveals more information than q_2, which means $q_2 \preceq q_1$.

This definition of determinacy order is a preorder (reflexive and transitive), but not necessarily a partial order, as it is not anti-symmetric. In other words, $q_1 \preceq q_2$ and $q_2 \preceq q_1$ does not necessarily mean that $q_1 = q_2$. As in Section D.2, this essentially means that query sets are not canonical representations of the information revealed by them. To rectify this, we form the closure $\downarrow$ under the determinacy order, so the determinacy order becomes set inclusion. Intuitively, $\downarrow Q$ will contain all the queries in $\mathcal{Q}$ whose answers can be inferred by the set of queries Q. Formally, $\downarrow Q$ is defined as:

$$\downarrow Q = \{q \in \mathcal{Q} \mid \{q\} \preceq Q\}$$

Using the definitions of determinacy order and closure $\downarrow$, we can then define the Determinacy Lattice as follows:

Definition D.2

Given a universe of queries $\mathcal{Q}$, the Determinacy Lattice $DL(\mathcal{Q})$ is a complete lattice $\langle \mathcal{L}, \sqsubseteq, \bigsqcup, \bot, \top \rangle$ such that:

- $\mathcal{L} = \{\downarrow Q \mid Q \subseteq \mathcal{Q}\}$
- $\downarrow Q_1 \sqsubseteq \downarrow Q_2$ iff $Q_1 \preceq Q_2$
- $\bigsqcup_i \downarrow Q_i = \downarrow \bigcup_i Q_i$
- $\bot = \downarrow \varnothing$, $\top = \downarrow \mathcal{Q}$,

where $\preceq$ is the determinacy order on $\mathcal{Q}$.

Disclosure order and information flow properties. Our definition of the Determinacy Lattice is similar to the definition of the Disclosure Lattice introduced by Bender et al. [99]. A Disclosure Lattice is a lattice built upon a disclosure order, which is a partial order on sets of queries satisfying additional conditions that are expected of an ordering according to the amount of information disclosed by each set of queries. Bender et al. [99] define the disclosure order as follows:

Definition D.3

Given a universe of queries $\mathcal{Q}$, a disclosure order $\preceq$ is a preorder on $\mathcal{P}(\mathcal{Q})$ that satisfies the following properties:

1. For all $Q_1, Q_2 \in \mathcal{P}(\mathcal{Q})$, if $Q_1 \subseteq Q_2$ then $Q_1 \preceq Q_2$
2. If $\mathbb{P} \subseteq \mathcal{P}(\mathcal{Q})$ and $\forall P \in \mathbb{P},\ P \preceq Q$ then $\bigcup \mathbb{P} \preceq Q$

The first property in this definition ensures that adding new elements to a set of queries only increases the amount of disclosed information and the second property allows us to derive a meaningful upper bound on the information disclosure.

The intended use of disclosure order was to order sets of queries based on the amount of information they reveal about the underlying database. However, we make the observation that this definition is not specific enough to characterize information disclosure in the information flow sense. For example, consider query containment [15], defined as:

Definition D.4

Given queries $q_1, q_2 \in \mathcal{Q}$, we say that q_1 is contained in q_2, denoted by $q_1 \subseteq q_2$, if for every database states $db \in \Omega_D$, we have $[\![q_1]\!]^{db} \subseteq [\![q_2]\!]^{db}$.

Query containment satisfies all of the requirements of a disclosure order (Def. D.3), but it is not enough to guarantee security. To illustrate this, consider a database with a single table t given in Figure D.3.

vl
0
1
$100 + s$

Figure D.3: Table t

Table t has a single column vl, and contains values 0, 1, and $100+s$, where s is a secret value that can be either 0 or 1. We thus consider two possible instances of this database, one where t contains values 0, 1, and 100 and another where it contains 0, 1, and 101. Now, consider the following queries:

$$q_1 : \{(vl_1) \mid \exists vl_2.\ t1(vl_1) \land t_2(vl_2) \land vl_1 < 100\}$$
$$q_2 : \{(vl_1) \mid \exists vl_2.\ t1(vl_1) \land t_2(vl_2) \land vl_1 < 100 \ \land vl_1 = vl_2 - 100\}$$

where t_1 and t_2 are just logical copies of table t. It is common practice to make logical copies of relation and use them in queries with self-joins [193]. The result of query q_1 is always 0 and 1. The result of query q_1 is 1 if the secret s is 1 and 0 if s is 0. As it is evident, for these queries, query containment holds and the result of query q_2 is contained in the results of q_1. However, an observer seeing the result of query q_2 can learn the value of secret s.

This example illustrates that query containment (a disclosure order) is not sufficient to guarantee the confidentiality of the secret s in an information flow setting. To ensure information flow security, we require a stronger condition, such as the notion of query determinacy order (Def. D.1) that we chose to rely on in this paper.

Relation between the DL and the LoI. There exists a close relationship between the DL and the LoI. Specifically, a query q defined over a database schema D induces an equivalence relation $q_{\sim}$ on database states db. We can formally define this equivalence relation as:

$$q_{\sim} = \{(db_1, db_2) \mid db_1, db_2 \in \Omega_D \wedge \llbracket q \rrbracket^{db_1} = \llbracket q \rrbracket^{db_2}\}$$

We write $[q_{\sim}]$ to denote the set of all equivalence classes induced by q. Given an equivalence relation $q_{\sim}$ on set Ω_D and $db \in \Omega_D$, $[db]_{q_{\sim}}$ denotes the equivalence class induced by $q_{\sim}$ to which the database state db belongs. We further lift this definition to sets of queries $Q = \{q_1, q_2, ..., q_n\}$:

$$Q_{\sim} = \{(db_1, db_2) \mid db_1, db_2 \in \Omega_D \bigwedge_{1 \leq i \leq n} \llbracket q_i \rrbracket^{db_1} = \llbracket q_i \rrbracket^{db_2}\}$$

This interpretation of database queries as equivalence relations provides a direct connection between the DL and the LoI, where the lattice elements correspond to $Q_{\sim}$, the ordering $\sqsubseteq$ to the determinacy order $\preceq$, and join and meet follow the definitions of the DL.

Lemma D.1

For all $\mathcal{Q}$, there is a complete lattice homomorphism from the Determinacy Lattice $DL(\mathcal{Q})$ to the Lattice of Information defined on $\{Q_{\sim} \mid Q \in DL(\mathcal{Q})\}$.

We prove this Lemma in Appendix D.2. To the extent that we believe $Q_{\sim}$ to accurately represent the information conveyed by the queries in Q, this lemma implies that joins and order comparisons can be performed in the DL without explicit reference to the LoI.

Determinacy Quantale

We introduce a generalization of the Determinacy Lattice, called the *Determinacy Quantale* (DQ), to represent disjunctive dependencies. Our definition of the DQ is intended as a counterpart to the QoI [181], analogously to how the DL corresponds to the LoI. To achieve this, we define a query-set counterpart of the tiling closure operator to capture the disjunction of sets of queries. Since *sets* of queries correspond to LoI elements (equivalence relations), disjunctive QoI elements (sets of equivalence relations) will be represented as *sets of sets* of queries. Each set of queries in the outer set represents a possible combination of queries that does not reveal more information than is allowed by the disjunction.

Analogously to the QoI, the tiling closure of a set of sets of queries is defined by forming the downward closure under $\sqsubseteq$ (from the DL) of their *mix*. The query-set equivalent of the *mix* operator is defined on a set of sets of queries $\mathbb{Q} = \{Q_1, ..., Q_n\}$ such that $Q_i \in DL(\mathcal{Q})$ for $i = 1, ...n$ as follows:

$$\text{mix}(\mathbb{Q}) = \{P \in DL(\mathcal{Q}) \mid x \in [P_\sim] \Rightarrow (\exists Q \in \mathbb{Q}.x \in [Q_\sim])\}$$

where $[Q_\sim]$ denotes the equivalence classes of Q as defined previously. We then define the tiling closure for a set $\mathbb{Q}$ of elements of the DL as $\text{tc}(\mathbb{Q}) = \Downarrow\text{mix}(\mathbb{Q})$. We then formally define the Determinacy Quantale $DQ(\mathcal{Q})$ as follows.

Definition D.5

Given a universe of queries $\mathcal{Q}$, let $DL(\mathcal{Q})$ be the Determinacy Lattice defined on $\mathcal{Q}$. The Determinacy Quantale $DQ(\mathcal{Q})$ is the quantale $\langle \mathcal{I}, \sqsubseteq, \bigvee, \otimes, 1\rangle$, with:

- $\mathcal{I} = \{\text{tc}(\mathbb{Q}) \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$
- $\bigvee_i \mathbb{P}_i = \text{tc}(\bigcup_i \mathbb{P}_i)$
- $\mathbb{P} \otimes \mathbb{Q} = \text{tc}\Big(\bigcup_{P\in\mathbb{P}, Q\in\mathbb{Q}}(P \sqcup Q)\Big)$
- $\sqsubseteq = \subseteq$
- $\top = DL(\mathcal{Q})$, $\bot = \varnothing$, $1 = \varnothing$,

where $\mathbb{P}, \mathbb{Q} \subseteq DL(\mathcal{Q})$.

In Appendix D.3 we show that Def. D.5 satisfies the usual quantale axioms [181]. As with the DL and LoI, the DQ embeds into a QoI by a quantale homomorphism. This QoI is defined on sets of equivalence relations derived from sets of sets of queries by the following map:

Definition D.6

Given a set of sets of queries $\mathbb{Q}$,

$$[\![\mathbb{Q}]\!] = \{Q_\sim \mid Q \in \mathbb{Q}\}.$$

We can then formally state the relationship between the DQ and QoI as follows.

Lemma D.2

For all $\mathcal{Q}$, there is a quantale homomorphism from the Determinacy Quantale $DQ(\mathcal{Q})$ to the Quantale of Information defined on $\{[\![\mathbb{Q}]\!] \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$.

The proof of Lemma D.2 is presented in Appendix D.4.

Example D.4

To illustrate the Determinacy Quantale in practice, consider Program D.2, which issues either query $q1 = \{(r, vl) \mid \exists s, n.\, \mathrm{emp}(n, r, s) \wedge r = \mathrm{Intern} \wedge vl = s\}$ or $q2 = \{(r, vl) \mid \exists s, n.\, \mathrm{emp}(n, r, s) \wedge r = \mathrm{CEO} \wedge vl = n)\}$ to the database. Query q1 returns the role and salary columns of the entry in table emp if the role of that entry is Intern. Similarly, query q2 returns the role and name columns if the role of the entry in emp is CEO.

Consider a policy defined on queries $v1 = \{(r, n) \mid \exists s.\, \mathrm{emp}(n, r, s)\}$ and $v2 = \{(r, s) \mid \exists n.\, \mathrm{emp}(n, r, s)\}$. v1 and v2, which respectively project on the name and role, and the role and salary columns of emp, are used in defining the disjunctive security policy $v1 \vee v2$.

For this example, we assume a database that has only one row in the emp table, and we also limit the domain of possible roles to $\{\mathrm{CEO}, \mathrm{Intern}\}$. These limitations are necessary in order to have a finite representation of the potential query sets and enables us to effectively depict the sets produced by the mix and tc operators.

Program D.2 depicts a disjunction that – ignoring variable y – depends either on $q1$ or $q2$ (i.e. $q1 \vee q2$), which on the DQ can be represented as a point $\mathrm{tc}(\downarrow\{q1\}) \vee \mathrm{tc}(\downarrow\{q2\})$. Similarly, the policy $v1 \vee v2$ can be represented on the DQ by $\mathrm{tc}(\downarrow\{v1\}) \vee \mathrm{tc}(\downarrow\{v2\})$.

Illustrating this point requires calculating the mix set of v1 and v2, which includes all sets of queries whose equivalence relation can be constructed from the equivalence classes of $\downarrow\{v1\}_\sim$ and $\downarrow\{v2\}_\sim$. Unfortunately, for any sufficiently rich query language, our definition of mix inevitably yields an infinite set, as infinitely many queries that are “morally equivalent” or even the same up to renaming variables represent the same knowledge set. To compactly represent such infinite sets, we will pick just one representative, and define

$$hc(\mathbb{Q}) = \{Q' \mid \exists Q \in \mathbb{Q}.\ Q_\sim = Q'_\sim\}$$

as a closure operator that adds all equivalent queries. Then $\mathrm{mix}\big(\{\downarrow\{v1\}, \downarrow\{v2\}\}\big)$ will be the set $hc(\{\downarrow\{v1\},\ \downarrow\{v2\},\ \downarrow\{p1\},\ \downarrow\{p2\}\})$, where $p1 = \{(r, vl) \mid \big(\exists s, n.\, \mathrm{emp}(n, r, s) \wedge r = \mathrm{Intern} \wedge vl = s\big) \vee \big(\exists s, n.\, \mathrm{emp}(n, r, s) \wedge r = \mathrm{CEO} \wedge vl = n\big)\}$ and $p2 = \{(r, vl) \mid \big(\exists s, n.\, \mathrm{emp}(n, r, s) \wedge r = \mathrm{CEO} \wedge vl = s\big) \vee \big(\exists s, n.\, \mathrm{emp}(n, r, s) \wedge r = \mathrm{Intern} \wedge vl = n\big)\}$.

Therefore, we can depict the policy as the point $\Downarrow(hc(\{\downarrow\{v1\},\ \downarrow\{v2\},\ \downarrow\{p1\},\ \downarrow\{p2\}\}))$ on the DQ. Similarly, the DQ point of the Program D.2 (i.e. $\mathrm{tc}(\downarrow\{q1\}) \vee \mathrm{tc}(\downarrow\{q2\})$), can also be depicted by the point $\Downarrow hc(\{\downarrow\{p1\}\})$ on the DQ. We illustrate the part of the DQ which includes these points in Figure D.4, and as it is evident from the figure, conclude that Program D.2 is inline with the policy.

```
if (y > 0) then
    x ← q1
else
    x ← q2
out(x, u);
```

Program D.2

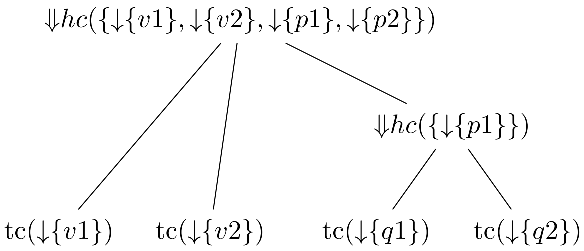


Figure D.4: A portion of the DQ for queries q1, q2, v1, v2

D.4 Security Framework

Drawing on the quantale model of dependencies for programs and databases, we develop an extensional condition that defines security for programs that interact with databases and support disjunctive security policies. We will later use the security condition to prove soundness of enforcement mechanisms in Section D.5. Specifically, we formalize the syntax and semantics of a simple imperative language with database queries. Programs read the input from the database via queries, while users receive the output through predefined output channels. We define (disjunctive) security policies as views over the database and interpret them end-to-end. We then use this model to define a knowledge-based security condition for our setting.

Language

Syntax. The syntax for the commands of our language as depicted in Figure D.5, primarily consists of standard commands such as assignment, conditionals, and loops.

$$c ::= \quad \texttt{skip} \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid x \leftarrow q \mid x := e \mid c_1; c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{out}(e, u)$$

Figure D.5: Language syntax

The command $\texttt{out}(e, u)$ outputs the result of evaluating expression e to user $u \in \mathcal{U}$. The command $x \leftarrow q$ issues the query q to the database and stores the result in variable x. For modeling the queries, we rely on conjunctive queries with comparison introduced in Section D.5.

Expressions e can be variables $x \in \text{Vars}$, values (integers) $n \in \text{Val}$, binary operations $e_1 \oplus e_2$, single tuples $tp \in \text{Val}$, and set of tuples $\overline{tp} \in \text{Val}$. For simplicity, we do not provide de-constructors for database tuples.

Semantics. As discussed in Section D.3, a database state (or simply state) $db \in \Omega_D$ is defined with respect to a schema D and a finite set of integrity constraints. A configuration $\langle c, m, db\rangle$ consists of a command c, a memory $m = \text{Var} \rightarrow \text{Val}$ mapping variables to values, and a state db.

$$\textsc{Int}\ \frac{}{\langle n, m, db\rangle \downarrow n} \qquad \textsc{Tuple}\ \frac{}{\langle tp, m, db\rangle \downarrow tp} \qquad \textsc{TupleSet}\ \frac{}{\langle \overline{tp}, m, db\rangle \downarrow \overline{tp}} \qquad \textsc{Var}\ \frac{vl = m(x)}{\langle x, m, db\rangle \downarrow vl}$$

$$\textsc{Op}\ \frac{\langle e_1, m, db\rangle \downarrow n_1 \qquad \langle e_1, m, db\rangle \downarrow n_2 \qquad n = n_1 \oplus n_2}{\langle e_1 \oplus e_2, m, db\rangle \downarrow n}$$

Figure D.6: Semantic rules for expressions

The semantics of expressions is mostly standard and its rules are presented in Figure D.6. We use judgments of the form $\langle e, m, db\rangle \downarrow vl$ to denote that an expression e evaluates to value vl in memory m and state db. For simplicity, we refrain from defining binary operations on tuples, unless the underlying database query is boolean.

We use judgments of the form $\langle c, m, db\rangle \xrightarrow{\alpha} \langle c', m', db'\rangle$ to denote that a configuration $\langle c, m, db\rangle$ in one step evaluates to memory m' and state db' and (possibly) produces an observation $\alpha \in \text{Obs}$; we write ϵ whenever a command produces no observation. We write $m[x \mapsto vl]$ to denote a memory m with variable x assigned the value vl.

Figure D.7 provides the semantic rules for commands. The query evaluation rule QueryEval is similar to assignment as it evaluates a query q into state db and stores the result in the variable x. We use the command $\texttt{out}(e, u)$ to produce an observation. Formally, an observation $\alpha \in \text{Obs}$ is a tuple $\langle o, u\rangle$, where $u \in \mathcal{U}$ is the identifier of the user observing the output and o is the result of evaluating expression e, which is either a simple value or the result set of a non-boolean query.

We write $\langle c, m, db\rangle \overset{\tau}{\Longrightarrow}_u \langle c', m', db'\rangle$ to denote when $\langle c, m, db\rangle$ takes one or more steps to reach configuration $\langle c', m', db'\rangle$ while producing the trace (sequence of observations) $\tau \in \text{Obs}^*$. We omit the final configuration whenever it is irrelevant and write $\langle c, m, db\rangle \overset{\tau}{\Longrightarrow}_u$.

Security Model

We now introduce our knowledge-based security model for disjunctive security policies. For simplicity, we denote the initial program memory by m_0 and assume it is fixed and public to all users, hence the only way to input sensitive information is through database queries. Users make observations through output channels, hence

$$\textsc{Skip}\quad \frac{}{\langle \texttt{skip}, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m, db\rangle} \qquad \textsc{Assign}\quad \frac{\langle e, m, db\rangle \downarrow vl \qquad m' = m[x \mapsto vl]}{\langle x := e, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m', db\rangle}$$

$$\textsc{IfTrue}\quad \frac{\langle e, m, db\rangle \downarrow n \qquad n \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, db\rangle \xrightarrow{\epsilon} \langle c_1, m, db\rangle} \qquad \textsc{IfFalse}\quad \frac{\langle e, m, db\rangle \downarrow n \qquad n = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, db\rangle \xrightarrow{\epsilon} \langle c_2, m, db\rangle}$$

$$\textsc{WhileTrue}\quad \frac{\langle e, m, db\rangle \downarrow n \qquad n \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, db\rangle \xrightarrow{\epsilon} \langle c; \texttt{while}\ e\ \texttt{do}\ c, m, db\rangle} \qquad \textsc{WhileFalse}\quad \frac{\langle e, m, db\rangle \downarrow n \qquad n = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m, db\rangle}$$

$$\textsc{Seq}\quad \frac{\langle c_1, m, db\rangle \xrightarrow{\alpha} \langle c_1', m', db'\rangle}{\langle c_1; c_2, m, db\rangle \xrightarrow{\alpha} \langle c_1'; c_2, m', db'\rangle} \qquad \textsc{SeqEmpty}\quad \frac{}{\langle \epsilon; c, m, db\rangle \xrightarrow{\epsilon} \langle c, m, db\rangle}$$

$$\textsc{QueryEval}\quad \frac{vl = \mathcal{E}[\![\texttt{q}]\!]\texttt{db} \qquad m' = m[x \mapsto vl]}{\langle x \leftarrow q, m, db\rangle \xrightarrow{\epsilon} \langle \epsilon, m', db\rangle} \qquad \textsc{Output}\quad \frac{\langle e, m, db\rangle \downarrow vl}{\langle \texttt{out}(e, u), m, db\rangle \xrightarrow{\langle vl, u\rangle} \langle \epsilon, m, db\rangle}$$

Figure D.7: Semantics rules for commands

their knowledge of the database is determined by what they can infer based on these observations. This model induces standard equivalence relations for database states and observation traces.

Database state equivalence. Two states db and db' are equivalent with respect to a set of tables and views V, written as $db \approx_V db'$, iff all tables and views in V have identical contents in db and db'. Formally, states db and db' are equivalent with respect to V iff for all view $v \in V$, $\mathcal{E}[\![\texttt{v.q}]\!]\texttt{db} = \mathcal{E}[\![\texttt{v.q}]\!]\texttt{db}'$ and for all table $t \in V$, $[\![t]\!]^{db} = [\![t]\!]^{db'}$. A set of tables and views V induces an equivalence relation, and for a state db, the equivalence class $[db]_V$ contains all states that are equivalent to db with respect to V.

Trace equivalence. We use trace projection to define trace equivalence. The projection of a trace τ for user u written as $\tau\lfloor_u$ is the sequence of all observations in τ that u can observe. Traces τ_1 and τ_2 are equivalent with respect to user u, written as $\tau_1 \approx_u \tau_2$, iff the projection of one of them to u is the prefix of the other, i.e. $\tau_1\lfloor_u \preceq \tau_2\lfloor_u$ or $\tau_1\lfloor_u \succeq \tau_2\lfloor_u$.

Equivalence of trace prefixes is a standard technicality needed to ignore leaks due to program's progress/termination [55], and here we adapt a definition of trace equivalence which does not differentiate between program divergence and termination [163].

User knowledge. When executing a program prg, we assume memory is always initially in the all-zero state m_0. Thus, we can view a program's execution for any user as a function from database db to user-observable output traces, $\tau_{\mathrm{prg},u}(db) = \tau\lfloor_u$ when $\langle \mathrm{prg}, m_0, db\rangle \overset{\tau}{\Longrightarrow}_u$. This function induces an equivalence relation on databases, $[\![\mathrm{prg}]\!]_u = \sim_{\tau_{\mathrm{prg},u}}$, which characterizes the knowledge of db conveyed by the output of prg to u.

Security policy. A security policy is a list of user policies (written as P_u) for each user $u \in \mathcal{U}$. User policies are defined as views and table identifiers over a database schema, and determine what a user u is allowed to observe.

$$\begin{array}{rcl} \mathrm{con} & := & \{v\} \mid \{t\} \mid \mathrm{con}_1 \cup \mathrm{con}_2 \\ \mathrm{dis} & := & \{\mathrm{con}\} \mid \mathrm{dis}_1 \cup \mathrm{dis}_2 \\ P_u & := & \mathrm{dis} \end{array}$$

Figure D.8: Syntax of user policy

Figure D.8 presents the syntax of disjunctive policies for our model. They are defined as a set of sets in order to represent a disjunction of conjunctions of simpler policies. A conjunction con is a set of view v and table t identifiers, and a disjunction dis is a set of conjunctions. For example, the policy P_u for user u who is allowed to see table t_1 and view v_1, or view v_2 but not both, is defined as $P_u = \{\{t_1, v_1\}, \{v_2\}\}$.

The overall policy of the system, written as P, is the list of user policies. Per Def. D.6, the policy P_u can be represented semantically as an element $[\![P_u]\!]$ of the Quantale of Information. Thus, we can formulate our security condition as the assertion that the knowledge of the database that the execution of the program prg conveys to u is bounded above by the disjunctive knowledge allowed by the policy, $[\![P_u]\!]$.

> **Definition D.7**
>
> The program prg is secure for the user u and policy P_u if $[\![\mathrm{prg}]\!]_u \sqsubseteq [\![P_u]\!]$.

D.5 Enforcement of Disjunctive Policies

Having formulated the security condition, we would like to prove that useful programs satisfy it. To this end, we introduce a sound static enforcement mechanism, which imposes some structural limitations on the policy and trades off some completeness for the sake of efficiency and ease of analysis.

Figure D.9 illustrates how our mechanism functions at a high level. We assume as input a program and policy in the format described in Figure D.5 and Figure D.8 respectively. The program is then subjected to a static *dependency analysis* (Section D.5), which computes an overapproximate set of possible paths of control flow through the program, along with the queries (dependencies) retrieved for each path,

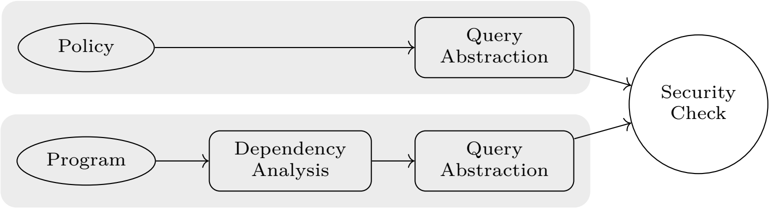


Figure D.9: Enforcement steps

giving an element of the DQ, that is a (disjunctive) set of (conjunctive) sets of queries. Per Figure D.8, the policy is also already given in this format.

We would like to verify that the program dependencies are bounded by the policy in the DQ, as by Lemma D.2, this entails the security condition (Def. D.7) that the disjunctive information that is revealed by the program is bounded above by the QoI interpretation of the policy. However, checking DQ ordering on general queries may be computationally costly. We therefore *abstract* (Section D.5) both the policy and the path dependencies into a more tractable format (symbolic tuples), which again overapproximates the information they can retrieve. To guarantee soundness, we require that the views in the policy are such that this abstraction is lossless for them. Finally, as the *security check* (Section D.5), we compute a tractable comparison on sets of sets of symbolic tuples that can be shown to imply DQ ordering.

Conjunctive Queries

While our theoretical definitions are based on the fully-general domain relational calculus as a query language, to avoid complexity, our enforcement mechanism will work with a restricted subset called *conjunctive queries with comparisons* (CQCs). This language is a subset of relational calculus that only employs conjunction ($\wedge$) and existential quantification ($\exists$) and omits disjunction ($\vee$), negation ($\neg$), and universal quantification ($\forall$). CQCs can model `SELECT-FROM-WHERE` portion of SQL, where there are only `AND` and comparisons in the `WHERE` clause.

Our language for (non-boolean) CQC q over a database schema D employs the standard notation [15, 193], and has the form *heading* $\leftarrow$ *body*:

$$\text{ans}(\overline{y}) \leftarrow R_1(\overline{x}_1), ..., R_n(\overline{x}_n), C_1, ..., C_m$$

where $R_1, ..., R_n$ are relations in D, and $\overline{x}_1, ..., \overline{x}_n$ are their variables. We use $\text{Var}(q) = \overline{x}_1 \cup ... \cup \overline{x}_n$ to denote the set of variables appearing in the body of the query q. $C_1, ..., C_m$ are formulae of the form $x_i \oplus x_j$ where $\oplus$ is the comparison operator which could be anything from $<, \leq, =, \neq, >, \geq$ and x_i and x_j are either variables in $\text{Var}(q)$ or constants.

We require that $\overline{y} \subseteq \mathrm{Var}(q)$. Without loss of generality, we assume that there are no self-joins in the query. In case of queries with self-joins, we can make logical copies of the relations to accommodate them [193]. The body of a CQC q comprises two parts, namely the relation identifiers $R_1, ..., R_n$ referred to as $\mathrm{ids}(q)$, and the conditions $C_1, ..., C_m$ denoted by $\mathrm{cnd}(q)$.

Similarly to Section D.3, the evaluation of q on the database state *db* (denoted by $\mathcal{E}[\![\mathtt{q}]\!]\mathtt{db}$) is defined by taking all tuples in the cartesian product of $\mathrm{ids}(q)$ in *db* that satisfy $\mathrm{cnd}(q)$, and projecting to the column set $\overline{y}$.

Example D.5

Consider the database schema in Figure D.2. The following query returns a set of tuples containing the names of divisions whose managers have a salary of more than 50:

$$\mathrm{ans}(d) \leftarrow \mathrm{emp}(n, r, s), \mathrm{mng}(d, m), n = m, s > 50$$

Type-based Dependency Analysis

Our static dependency analysis builds on the generic type system of van Delft et al. [116] and extends it with support for disjunctive dependencies. We intuitively expect that a disjunctive dependency analysis must be path-sensitive, so as to distinguish between different executions and also keep track of the history of observations. Both of these requirements are often challenging for type-based analyses, which do not naturally align with the execution order. We will first illustrate these challenges with examples and then present our analysis.

Program D.3 illustrates the need for path sensitivity. The analysis should distinguish between the *then* branch, where variable x depends on the set $\{y, w, z\}$, and the *else* branch where x depends on $\{y, x\}$. Our reference analysis [116] would join these two sets at the end of the if statement, ultimately yielding the dependency set $\{x, y, w, z\}$. In our analysis, these sets are never joined, but instead combined to form a set of sets, namely, $\{\{y, w, z\}, \{y, x\}\}$, where the outer set represents a disjunctive dependency and the inner sets represent conjunctive dependency.

```
if (y > 0) then
    x := w + z;
else
    x := x + 1;
out(x,u);
```

Program D.3

Program D.4 illustrates the need to keep track of the observation history. It outputs x at lines 5 and 10, and the dependency set of x in both places is $\{\{q1, z\}, \{q2, z\}\}$. However, this program will always output both $q1$ and $q2$. Now, if a policy only

$$\textsc{T-Skip}\quad \frac{}{\vdash \texttt{skip} : \Gamma_{id}} \qquad \textsc{T-Assign}\quad \frac{\Gamma = \Gamma_{id}[x \mapsto \{fv(e) \cup \{pc\}\}]}{\vdash x := e : \Gamma} \qquad \textsc{T-QueryEval}\quad \frac{\Gamma = \Gamma_{id}[x \mapsto \{\{q, pc\}\}]}{\vdash x \leftarrow q : \Gamma}$$

$$\textsc{T-If}\quad \frac{\vdash c_i : \Gamma_i \qquad \Gamma'_i = \Gamma_i ; \Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}]\ i = 1, 2 \qquad \Gamma' = (\Gamma'_1 \Cup \Gamma'_2)[pc \mapsto \{\{pc\}\}]}{\vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \Gamma'}$$

$$\textsc{T-While}\quad \frac{\vdash c : \Gamma_c \qquad \Gamma_f = (\Gamma_c ; \Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}])^* \qquad \Gamma' = \Gamma_f[pc \mapsto \{\{pc\}\}]}{\vdash \texttt{while}\ e\ \texttt{do}\ c : \Gamma'}$$

$$\textsc{T-Output}\quad \frac{\Gamma' = \Gamma_{id}[u \mapsto \{fv(e) \cup \{pc, u\}\}]}{\vdash \texttt{out}(e, u) : \Gamma'} \qquad \textsc{T-Seq}\quad \frac{\vdash c_1 : \Gamma_1 \qquad \vdash c_2 : \Gamma_2 \qquad \Gamma' = \Gamma_2 ; \Gamma_1}{\vdash c_1 ; c_2 : \Gamma'}$$

Figure D.10: Type-based dependency analysis rules

allows user u to see either query $q1$ or $q2$, the outputs at lines 5 and 10 will be incorrectly accepted. Hence, the analysis should account for all outputs to user u.

```
1   if (z == 0) then
2       x ← q1;
3   else
4       x ← q2;
5   out(x,u);
6   if (z != 0) then
7       x ← q1;
8   else
9       x ← q2;
10  out(x,u);
```

Program D.4

Figure D.10 depicts the rules of our disjunctive dependency analysis. We use judgments of the form $\vdash c : \Gamma$, where Γ is an environment mapping variables Var to set of sets of dependencies Dep. The set of variables is $\text{Var} = PV \cup \mathcal{U} \cup \{pc\}$, where PV are program variables, $\mathcal{U}$ are users, and pc is the program context. The dependencies Dep are $\text{Dep} = \text{Var} \cup \mathcal{Q}$, where Var are variables and $\mathcal{Q}$ are queries that can be issued to a database. We use $u \in \mathcal{U}$ to indicate the dependencies of all outputs to user u.

We start by introducing the operators and auxiliary functions employed within the rules, and then proceed to explain the rules themselves. The operator $\otimes$ is used to join two (or more) sets of sets, defined as:

$$\Gamma_1(x_1) \otimes ... \otimes \Gamma_n(x_n) = \{S_1 \cup ... \cup S_n \mid S_i \in \Gamma_i(x_i) \quad i = 1, \ldots, n\}$$

For example, the join of $\Gamma_1(x) = \{\{x, y\}, \{z, y\}\}$ and $\Gamma_2(y) = \{\{w\}, \{x, z\}\}$ is:

$$\Gamma_1(x) \otimes \Gamma_2(y) = \{\{x, y, w\}, \{x, y, z\}, \{z, y, w\}\}$$

Intuitively, the result of the join operator is a set of sets capturing the product of the original sets of sets under the set union operation. We use this operator to calculate all the possible combinations of two environments.

$\Gamma_2; \Gamma_1$ represents the sequential composition of two environments. Intuitively, $\Gamma_2; \Gamma_1$ is the same as Γ_2 but updated with all of the dependencies that have been previously established in Γ_1. Formally:

$$\Gamma_2; \Gamma_1(x) = \bigcup_{S_2 \in \Gamma_2(x)} \bigotimes_{y \in S_2} \Gamma_1(y)$$

For example, the sequential composition of the environments

$$\begin{aligned} \Gamma_1 &= [x \mapsto \{\{x\}, \{y\}\}, y \mapsto \{\{y\}\}, pc \mapsto \{\{y, pc\}\}] \\ \Gamma_2 &= [x \mapsto \{\{pc, x\}\}, y \mapsto \{\{pc, y\}\}, pc \mapsto \{\{pc\}\}] \end{aligned}$$

evaluates to

$$\Gamma_2; \Gamma_1 = [x \mapsto \{\{x, y, pc\}, \{y, pc\}\}, y \mapsto \{\{pc, y\}\}, pc \mapsto \{\{y, pc\}\}]$$

Finally, the operator $\Cup$ calculates the union of two environments: $\Gamma_1 \Cup \Gamma_2 = \forall x \in \text{Var},\ \Gamma_1(x) \cup \Gamma_2(x)$. This operator is used in conditionals to capture the disjunctive join of the two branches. For example, in line 5 in Program D.3, $\Gamma_1(x) = \{\{y, w, z\}\}$ and $\Gamma_2(x) = \{\{y, x\}\}$, and the result of $(\Gamma_1 \Cup \Gamma_2)(x)$ would be $\{\{y, w, z\}, \{y, x\}\}$.

For loops, we rely on the fixed-point of Γ, denoted by Γ^*, which we define as:

$$\Gamma^* = \bigcup_{n > 0} \Gamma^n$$

where $\Gamma^0 = \Gamma_{id}$ and $\Gamma^{n+1} = \Gamma^n; \Gamma$.

In these rules, Γ_{id} is the identity environment, defined as $\forall x \in \text{Var},\ \Gamma_{id}(x) = \{\{x\}\}$, and $fv(e)$ denotes the free variables of expression e.

T-Assign updates the dependency set of the assigned variable x to the set of the free variables of expression e and *pc*, otherwise it matches the identity environment. Rule T-QueryEval is similar to assignment, except that instead of $fv(e)$, it adds query q to the dependency set.

T-If sequentially composes the dependency sets of each branch with the environment $\Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}]$, thus adding variables of the branch condition to the dependency set of each branch. Finally, these environments (Γ_1 and Γ_2) are joined disjunctively using the $\Cup$ operator.

T-While uses the fixed-point operator to calculate the dependency set of the loop. To do so, it first calculates the dependency set of the loop body, which is sequentially composed with $\Gamma_{id}[pc \mapsto \{fv(e) \cup \{pc\}\}]$ to account for the dependencies to the loop condition. Finally, the fixed-point operator computes the dependency set of the while loop.

T-Output relies on the dependency set including $fv(e)$, $\{pc\}$ and $\{u\}$, where $fv(e)$ includes all the variables of the expression outputted to user u, $\{pc\}$ captures the implicit dependencies to the path conditions, and $\{u\}$ is the dependency set of user u and captures the history of dependencies that user u might have observed up to this point. Observe that by the definition of sequential composition, all the dependencies of the previous outputs will be added to u.

This analysis yields a final environment Γ_{fin}. The result of the analysis is the value of this environment for the user identifier u, which includes both queries and program variables. Since program variables do not contain sensitive information, and we are primarily concerned with queries, we refine the result of $\Gamma_{\text{fin}}(u)$ to only include queries. This refined outcome defines the ultimate result of our analysis, denoted as QL_u:

$$\text{QL}_u \triangleq \bigcup_{S \in \Gamma_{\text{fin}}(u)} \{S \cap \mathcal{Q}\}$$

The soundness proof of our enforcement relies on the circumstance that, if the set of queries on which the u-outputs of prg depend when running on a database state db are denoted by $Q_{\text{prg},u}(db)$, then this set is guaranteed to be found in the set QL_u produced by the dependency analysis. We show how to define $Q_{\text{prg},u}(db)$ using a taint-tracking semantics presented in Appendix D.5. Formally, this gives rise to the following soundness condition for the dependency analysis.

Lemma D.3

For all $db \in \Omega_D$, $Q_{\text{prg},u}(db) \in \text{QL}_u(\text{prg})$.

Query Abstraction

Even for CQCs, comparing the information revealed by sets of queries is hard in general. To define a well-behaved and more tractable determinacy order on which to build our DQ, we introduce another overapproximating abstraction, which we will use to soundly *label* queries and policies.

We define a *symbolic tuple* as $\langle T, \phi, \pi \rangle$, where $T = \{t_1, t_2 ..., t_n\}$ is a set of table identifiers, ϕ is a boolean combination of equality, inequality, and comparisons over the columns of the tables in T, and π is a subset of the columns of the tables in T. In a symbolic tuple, π denotes the query's projection on the columns of the tables in T, and ϕ defines the constraints over the rows.

Example D.6

The symbolic tuple of query $\text{ans}(d) \leftarrow \text{emp}(n, r, s), \text{mng}(d, m), n = m, s > 50$ defined on the relations of Figure D.2 would be $\langle \{\text{emp}, \text{mng}\}, s > 50 \wedge n = m, \{d\} \rangle$.

While calculating the exact set of symbolic tuples of a relational calculus query is intractable for many classes of queries, it is tractable for conjunctive queries with comparison (CQC). Given a conjunctive query $q = \text{ans}(\overline{y}) \leftarrow R_1(\overline{x}_1), ..., R_n(\overline{x}_n), C_1, ..., C_m$, the function sts computes a symbolic tuple from q as follows:

$$\text{sts}(q) = \langle \text{ids}(q'), (\bigwedge_{C \in \text{cnd}(q')} C), \overline{y} \rangle$$

where ids(q') and cnd(q') defined in Section D.5 return the relation identifiers and conditionals of q', respectively. Here, q' is the query obtained by recursively replacing views with their definitions. We lift this definition to sets of queries Q, and define sts(Q) as $\{\bigcup_{q \in Q} \text{sts}(q)\}$.

Using sts, we define the function σ_{st} for a set of sets of queries $\mathbb{Q}$ as follows:

$$\sigma_{\text{st}}(\mathbb{Q}) = \{\text{sts}(Q) \mid Q \in \mathbb{Q}\}$$

Policy Analysis. The function σ_{st} can also be used to map a disjunctive security policy to a set of labels. However, in order to ensure soundness and avoid approximation, we place some constraints on policies. (1) To make computing the set of symbolic tuples tractable we only support policies with views in the CQC form. (2) We require that the symbolic tuples of views be *well-formed*, which we define as:

Definition D.8

The symbolic tuple $\langle T, \phi, \pi \rangle$ is said to be well-formed if it satisfies:

$$\text{dep}(\phi) \subseteq \pi$$

where $\phi = C_1 \land ... \land C_n$ and $\text{dep}(\phi) = \bigcup_{i \in \{1,...,n\}} fv(C_i)$ returns the column dependency set of ϕ.

Well-formedness ensures that the symbolic tuples are precise, at the expense of limiting a view to only applying constrains on the columns which it projects on.

Furthermore, we treat the table identifiers used in policies as special views that return the whole table. For instance, a policy which allows access to table emp can be rewritten as view ans$(n, r, s) \leftarrow$ emp(n, r, s).

As discussed in Section D.4, the disjunctive security policy of user u (written as P_u) is a set of conjunctions con, interpreted as a disjunction of conjunctions of table and view identifiers. For a policy P_u that adheres to the constraints mentioned earlier, σ_{st} is defined as follows:

$$\sigma_{\text{st}}(P_u) = \{\text{sts}(\text{con}) \mid \text{con} \in P_u\}$$

Labels. In our model, a security label ℓ is defined as a set of symbolic tuples, and we define the ordering relation of two labels, written as $\ell_1 \sqsubseteq_{\text{st}} \ell_2$, as follows:

Definition D.9

$\ell_1 \sqsubseteq_{\text{st}} \ell_2$ iff for all symbolic tuples $\langle T, \phi, \pi \rangle \in \ell_1$, there are well-formed symbolic tuples $\langle T_1, \phi_1, \pi_1 \rangle, ..., \langle T_n, \phi_n, \pi_n \rangle$ in ℓ_2 such that $T \subseteq (T_1 \cup ... \cup T_n)$, $T_1, ..., T_n$ are disjoint, $\phi \models (\phi_1 \wedge ... \wedge \phi_n)$, and $dep(\phi) \cup \pi \subseteq (\pi_1 \cup ... \cup \pi_n)$.

To ensure soundness, we assume that all of the symbolic tuples in the right hand side of $\sqsubseteq_{\text{st}}$ are well-formed. This definition relies on entailment to check the ordering of ϕ, and write $\phi_1 \models \phi_2$ which means that any assignment that satisfies ϕ_1 also satisfies ϕ_2.

Example D.7

Consider symbolic tuples $\ell_1 = \{\langle \{\text{emp}\}, s = 10, \{r\} \rangle\}$ and $\ell_2 = \{\langle \{\text{emp}, \text{mng}\}, s > 5, \{r, s, m\} \rangle\}$. We have $\ell_1 \sqsubseteq_{\text{st}} \ell_2$ since $\{\text{emp}\} \subseteq \{\text{emp}, \text{mng}\}$, $\{r\} \subseteq \{r, s, m\}$, $s = 10 \models s > 5$ and $\{s\} \cup \{r\} \subseteq \{r, s, m\}$.

Enforcement

The dependency analysis of Section D.5 extracts the dependencies of program prg's outputs to user u and produces QL_u. Applying σ_{st} to QL_u yields a set of labels, each bounding the information revealed in some path, the u-knowledge of prg (denoted by $k(\text{prg})_u$). We interpret this as a disjunction, as any execution follows along one particular path.

Similarly, applying σ_{st} to the disjunctive security policy of user u (i.e. P_u) results in a set of labels. Each label faithfully captures one conjunction, and so the policy is also represented as a set of labels $ak(P_u)$, interpreted disjunctively.

By Lemma D.2, to verify that the security condition is satisfied, it is sufficient to establish that $\text{QL}_u \sqsubseteq P_u$ in the DQ. However, checking $\sqsubseteq$ in the DQ is not generally tractable. For the security check, we therefore instead perform a twofold approximation: we check ordering for the conjunctive inner sets using the approximate ordering $\sqsubseteq_{\text{st}}$, and approximate the mix-based ordering on the disjunctive outer sets in a way that loses little relative to our analysis:

Definition D.10

We say that $k(\text{prg})_u \sqsubseteq_* ak(P_u)$ iff

$$\forall \ell_k \in k(\text{prg})_u, \ \exists \ell_{ak} \in ak(P_u). \ \ \ell_k \sqsubseteq_{\text{st}} \ell_{ak}$$

$$\begin{tikzcd}
 & & \text{DQ} & & \text{QoI} \\
ak(P_u) & & P_u \arrow[ll, mapsto, "\sigma_{\text{st}}"'] \arrow[rr, mapsto] & & \llbracket P_u \rrbracket \\
\rotatebox[origin=c]{90}{$\sqsubseteq$}^{*} & \Rightarrow & \rotatebox[origin=c]{90}{$\sqsubseteq$} & \Rightarrow & \rotatebox[origin=c]{90}{$\sqsubseteq$} \\
k(\text{prg})_u & & \text{QL}_u \arrow[ll, mapsto, "\sigma_{\text{st}}"] \arrow[rr, mapsto] & & \llbracket \text{QL}_u \rrbracket \\
 & & & \Rightarrow & \rotatebox[origin=c]{90}{$\sqsubseteq$} \\
 & & \text{prg} \arrow[uu, mapsto, "\text{D.A.}"] \arrow[rr, mapsto] & & \llbracket \text{prg} \rrbracket_u
\end{tikzcd}$$

Figure D.11: Overall architecture of our proof

where ℓ_{ak} and ℓ_k are labels, and $\sqsubseteq_{\text{st}}$ is the symbolic tuple ordering of Def. D.9. To ensure faithful labeling of policies, we assume all of the symbolic tuples in ℓ_{ak} are well-formed as defined in Def. D.8. We can then formalize the relationship between $\sqsubseteq_*$ and $\sqsubseteq$ as follows.

Lemma D.4

If $\sigma_{\text{st}}(\{Q_1, ..., Q_n\}) \sqsubseteq_* \sigma_{\text{st}}(\{P_1, ..., P_m\})$, then in the DQ, $(Q_1 \vee ... \vee Q_n) \sqsubseteq (P_1 \vee ... \vee P_m)$.

We refer the readers to Appendix D.6 for the proof of this Lemma.

Soundness Proof

Figure D.11 outlines the overall architecture of our enforcement mechanism and the correctness assertion that we make of it.

The rightmost column of Figure D.11 represents a chain of information order relations in the QoI, which we establish for each enforcement step. Following the chain from bottom to top, we obtain the security condition of Def. D.7. At the same time, the "left boundary" of the figure, comprising the D.A., σ_{st} abstractions and $\sqsubseteq_*$ check, represents the computations that are actually performed to check a program.

Theorem D.1

If a program prg *satisfies Def. D.10, then it is secure in the sense of Def. D.7.*

Proof. The statement follows from establishing the implications in the diagram of Figure D.11. The top left cell is Lemma D.4; the top right cell is Lemma D.2; and the bottom cell (dependency analysis) is Appendix D.5. □

D.6 Implementation and Evaluation

In this section, we describe our prototype DiVerT [194], which implements the type-based dependency analysis of Section D.5 and query abstraction of Section D.5 to verify the security of database-backed programs. We then evaluate DiVerT's effectiveness using functional tests and an assortment of real-world-inspired use cases.

Implementation

To evaluate the feasibility and security of our approach in practice, we implemented the type-based dependency analysis of Section D.5. For the sake of practicality, instead of CQC, DiVerT uses the `SELECT-FROM-WHERE` portion of SQL, which is analogous to CQC as described in Section D.5. Following the query analysis of Section D.5, these SQL queries are then converted into symbolic tuples. For the security check, the symbolic tuples with the result of the program analysis must be compared to those representing the policy; to perform this comparison following Def. D.9, we use the Z3 SMT solver [58]. Our implementation operates on programs in the language presented in Section D.4, with the addition of two macros `@Table@` and `@Policy@` for defining the tables' schema and the security policy.

Test suite

To validate our implementation, we use a functional test suite consisting of 20 programs, designed to capture a broad variety of examples of disjunctive dependencies. This suite includes programs with row- and column-level policies of varying granularity levels, and those necessitating the use of SMT solvers for verification. Furthermore, the tests verify the behaviour of the dependency analysis by incorporating complex conditionals, loops, and implicit and explicit outputs. The tests can be found in the implementation repository [194].

Use cases

We evaluate DiVerT on four use cases inspired by real-world problems in which disjunctive policies naturally arise. The purpose of this evaluation is to validate the security analysis of DiVerT on realistic scenarios involving disjunctive policies, and ensure that its behaviour is consistent with the definitions of Section D.4. Rather than analysing complete applications for each example, we therefore focus on smaller kernels that capture the core security-critical behaviour of the respective problem.

Privacy-preserving location service. Multilateration is a technique to determine the location of a user by measuring their distance to known reference points [17]. Two distances are sufficient to narrow a user's location down to one of two points on a map, and three identify the location unambiguously. Consider a location service provider which tracks, for some number of users, not only their precise location but also their distances to certain points of interest (PoI) such as restaurants or shops. An advertiser wants to query this service to provide location-based ads. For example, if the user is close to a shop A, and A has a sale going on, the user may be enticed by this information.

Privacy and business considerations make it desirable to not reveal the precise location of the user to the advertisement company accessing the database, while still allowing for some location-based services in this vein. If the advertiser were to learn the distance of a single user to two or more PoIs at a specific time, the user's location could be inferred. However, we may still want to release the user's distance to any one PoI which they are currently closest to. This can be interpreted as a disjunctive policy, in which the information revealed for each user is bounded by the disjunction of that user's distances to some *single* PoI.

The database schema consists of a single table `Distance(id, poi, dis, loc)`, which stores the ID of each user, the name of the PoI, their distance, and the user's precise location. We implement a small example with two PoIs {'restaurant', 'mall'} and two users $\{1, 2\}$. Let the view $v_{i,j}$ for each user i and PoI j be defined as the query `SELECT id, poi FROM Distance WHERE id =` i `AND poi =` j. The disjunctive policy then covers every combination of user and PoI as a possibility:

$$\begin{gathered}\big\{\{v_{1,\text{`restaurant'}}, v_{2,\text{`restaurant'}}\}, \{v_{1,\text{`restaurant'}}, v_{2,\text{`mall'}}\},\\ \{v_{1,\text{`mall'}}, v_{2,\text{`restaurant'}}\}, \{v_{1,\text{`mall'}}, v_{2,\text{`mall'}}\}\big\}\end{gathered}$$

We test two programs against this policy. In one, the advertiser uses internal parameters identifying a target user and interest, and issues a single query requesting that user's distance from the relevant point of interest. In the other, the advertiser still targets a particular user, but queries all of that user's distances. As expected, DiVerT accepts the former program, but rejects the latter.

Privacy-preserving data publishing. Expanding upon the motivating example in the introduction, we consider the case of programs querying a database with personally identifiable information (i.e. quasi-identifiers). As discussed before, revealing too many quasi-identifiers may make it possible to identify an individual. We consider the example of a medical database [26] with a table `Patients(zip, gen, dis)` storing the ZIP code of residence, gender and disease of patients. An agent querying the database should not learn more than two of these at a time. For simplicity's sake, we only consider queries that retrieve the same data from each patient. Defining $v_1 =$ `SELECT dis, gen FROM Patients`, $v_2 =$ `SELECT zip,` gen FROM Patients, and v_3 = SELECT zip, dis FROM Patients, the disjunctive policy can then be written as $\{\{v_1\}, \{v_2\}, \{v_3\}\}$.

Once again, we validate two programs against this policy. Branching on an internal parameter, the client will issue one query to select data for either male or female patients. In the first program, all queries take the form of SELECT dis FROM Patients WHERE gen = 'F', whereas in the second one, one of the queries additionally filters on the ZIP code: SELECT dis FROM Patients WHERE gen = 'F' AND zip = 10001. Again, only the latter program is rejected by DiVerT. This reveals a potential subtlety, as data dependency and hence release of information may arise not only from what columns are selected, but also from conditions restricting the set of rows.

Secret sharing. We implement a (t, n) secret sharing schema that splits a secret value s into n shares $s_1, s_2, ..., s_n$. These shares are then distributed among n parties $p_1, p_2, ..., p_n$, each receiving a unique share. A secure secret sharing schema requires that the secret s can only be reconstructed if t or more participants combine their shares. If the number of combined shares is less than t, no information about the secret should be revealed. This requirement naturally translates to a disjunctive policy $s_1 \vee s_2 \vee ... \vee s_n$, stipulating that participants can each only learn *one* share.

We assume that the shares $s_1, s_2, ..., s_n$ are created by a secure secret sharing schema and are then stored in a database. The database schema consists of the table Shares(shareID, shareVal) which stores the ID of each share and their corresponding value.

The policy only allows a user to read one of the shares (i.e. only one row of the table). We define the view v_i for each share as SELECT shareVal, shareID FROM Shares WHERE shareID = i where $i = 1, ..., n$. The corresponding disjunctive policy is going to look like $\{\{v_1\}, \{v_2\}, ..., \{v_n\}\}$.

We implement a program that executes a subroutine for each user, issuing a database query to retrieve the user's share. For example the query for a user to retrieve the share number 5 is SELECT shareVal FROM Shares WHERE shareID = 5 and it is correctly accepted by DiVerT. If the same user issues another query to retrieve share number 6, it violates the policy and hence the program is rejected. This scenario shows that DiVerT is able to correctly enforce row-level policies precisely.

Online shop. This use case models an online shop and a user with a gift card can only use it to "buy" items that match the value of the gift card. Here we consider a scenario with an online shop that only provides digital items and they are stored in a database. The database schema consists of the items table Items(id, name, data) which stores the ID and name of each digital item. We define a view v_n for each item as SELECT data, name FROM Items WHERE name = n where n is the item's name.

Assume a database that has the items *Movie*, *CinemaTicket*, *Audiobook*, *Ebook*, and

GymMem. A policy should only allow the user to access a certain amount of items whose value adds up to value of gift card. For instance a disjunctive policy may look like:

$$\{\{v_{\text{Movie}}, v_{\text{CinemaTicket}}\}, \{v_{\text{Audiobook}}, v_{\text{Ebook}}\}, \{v_{\text{GymMem}}\}, \{v_{\text{CinemaTicket}}, v_{\text{Ebook}}\}\}$$

We model a user program that issues queries to select items, e.g. `SELECT data FROM Items WHERE name = 'Movie'`.

DiVerT accepts this query because view v_{Movie} allows the user to access *Movie*. We create two different scenarios; in one the user issues another query asking for *Audiobook*, which DiVerT rejects. In the second scenario, the user asks for *CinemaTicket* which is allowed by the policy, and hence DiVerT accepts it.

D.7 Related Work

This section puts our contributions in the context of related works in the areas of information flow security and database security, discussing security models of dependencies and tractable enforcement mechanisms. To our knowledge, we are the first to explore enforcement mechanisms for disjunctive policies, as well as to reconcile semantic models of (disjunctive) dependencies across the areas of information flow control and database access control.

Security models. Semantic models of dependencies have a long history since the introduction of the Lattice of Information (LoI) by Landauer and Redmond [13]. These models define a lattice structure to represent information as equivalence relations ordered by refinement and serve as cornerstone to justify soundness of various dependency analysis at the heart of enforcement mechanisms for security. For example, the universal lattice by Hunt and Sands [46] models dependencies between program variables such that the lattice elements are sets of variables ordered by set containment, and uses it to justify soundness against baseline security conditions, e.g. noninterference [8].

Within the database community, Bender et al. [99, 104] define the notion of Disclosure Lattice to represent the information disclosed by sets of database queries. Disclosure Lattice has been further developed by Guarnieri et al. [163] to enforce conjunctive information-flow policies for database-backed programs. We point out that not all disclosure orders are suitable to represent information disclosure in the context of information flow control: By studying its relation to LoI, we show that query determinacy and the stronger notion of equivalent query rewriting [76] provide sound abstraction, while query containment does not.

Our work builds on recent work by Hunt and Sands [181], which provides a semantic model for disjunctive dependencies, under the notion of the Quantale of Information.

We study quantale structures in the context of databases, providing support for disjunctive policies in database-backed programs. While these policies are rooted in the area of access control, cf. ethical wall policies [12], the work of Hunt and Sands [181] is the first to provide an extensional characterization as information-flow policies. Drawing on our new notion of Determinacy Quantale, we develop a security condition to capture the security of database-backed programs in presence of disjunctive database policies.

Enforcement mechanisms. The problem of enforcing disjunctive policies for programs and/or databases is completely unexplored. We study how a standard type-based program analysis [116], equipped the notion of path sensitivity, can be adapted to statically capture disjunctive program dependencies.

At the core of our analysis is a new abstraction of database queries which enables flexible enforcement of disjunctive policies by means of SMT solvers, as witnessed by our use cases. An immediate benefit of our Determinacy Quantale is that we can prove soundness of the enforcement with respect to a solid semantic baseline for disjunctive dependencies.

There exists a wide array of works enforcing conjunctive policies for database-backed programs. Guarnieri et al. [163] propose dynamic monitoring to enforce database policies. Their abstractions are limited to boolean queries and rely on the Disclosure Lattice of Bender et al. [99, 104], which may cause soundness issues when assuming query containment as the underlying lattice order.

Language-integrated queries are supported by a range of works such as SIF [52] and JsLinq [122], SeLinks [65], UrFlow [73], DAISY [163], Jacqueline [131], and LWeb [165] for row- and column-level conjunctive policies. These works apply PL-based enforcement techniques such as type systems, dependent types, refinement types, and symbolic execution to database-backed programs [59, 117, 163, 165], but lack support for expressing and enforcing disjunctive policies.

Li and Zhang [136] explore path-sensitive program analysis to improve precision of information flow analysis, yet they do not consider disjunctive policies. QAPLA [140] is a database access control middleware supporting complex security policies, such as linking and aggregation policies, with focus only on access control.

D.8 Conclusions

We presented a case for the significance of disjunctive dependency analysis to the security of database-backed programs. After reviewing recent theoretical developments in representing disjunctive information, we introduced two structures, the Determinacy Lattice and the Determinacy Quantale, as database-oriented counterparts to theoretical structures representing simple and disjunctive knowledge respectively.

Using these structures, we formulated a security condition which expresses that a database-backed program satisfies a given disjunctive policy. In order to enforce this security condition, we developed a type-based static analysis to compute a bound on the disjunctive dependencies of database-backed programs in a model language. By a series of approximations, this bound itself can be tractably compared to the representation of a static policy.

These steps constitute an enforcement mechanism for disjunctive policies, which we proved sound with respect to our security condition. To showcase this enforcement mechanism, we implemented it in our prototype tool, DiVerT. In order to validate this prototype and the overall framework, we verified the tool on a set of functional tests covering a variety of language features and disjunctive information patterns, as well as several use cases representing real-world scenarios in which we want to enforce disjunctive policies.

Acknowledgements

We are grateful to David Sands and Roberto Guanciale for fruitful discussions, and would also like to thank the anonymous reviewers for their insightful comments and feedback.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Research Council (VR), and the Swedish Foundation for Strategic Research (SSF).

Appendices

Appendix A Interpretations of Query Determinacy

We prove the following technical lemma to show that the two intuitive interpretations of the definition of query determinacy are equivalent.

Lemma D.5

If A is recursively enumerable and $f : A \to B$ and $g : A \to C$ are computable, then the following are equivalent:

(i) For all $a, a' \in A$, if $f(a) = f(a')$, then $g(a) = g(a')$.

(ii) There exists a computable $h : B \to C$ such that for all $a \in A$, $g(a) = h(f(a))$.

Proof.

(ii)⇒(i): Suppose $b = f(a) = f(a')$, and h is as in (ii). Then $g(a) = h(f(a)) = h(b)$, and $g(a') = h(f(a')) = h(b)$.

(i)⇒(ii): Let $\hat{f} : B \rightharpoonup A$ be the partial function that enumerates A and for a given $b \in B$ returns the first $a \in A$ it finds such that $f(a) = b$. This is computable, per the algorithmic description provided. This does not necessarily satisfy $\hat{f}(f(a)) = a$, but we do have $f(\hat{f}(f(a))) = f(a)$ by definition (since the enumeration of A will either encounter a or another a' such that $f(a') = f(a)$ eventually). Hence $g(\hat{f}(f(a))) = g(a)$ by (i). So defining h by $h(b) = g(\hat{f}(b))$, we find that $h(f(a)) = g(a)$ as required.

□

Instantiating Lemma D.5 with A as the set of possible databases, f as the function $r_Q(db) = \{[\![q]\!]^{db} \mid q \in Q\}$ that computes the results of the queries in Q on db, and g as the same for Q', we find that Q determining Q' indeed means that the (results of) queries in Q are always sufficient to determine (compute) the result of the queries in Q'.

Appendix B Relation Between DL and LoI

We first prove some auxiliary lemmas, and then proceed to prove Lemma D.1.

Lemma D.6

For sets of queries $Q_1, Q_2 \in DL(\mathcal{Q})$, the ordering $\downarrow Q_1 \sqsubseteq \downarrow Q_2$ on the DL implies $Q_{1\sim} \sqsubseteq Q_{2\sim}$ on the LoI defined on $\{Q_\sim \mid Q \in DL(\mathcal{Q})\}$:

$$\downarrow Q_1 \sqsubseteq \downarrow Q_2 \rightarrow Q_{1\sim} \sqsubseteq Q_{2\sim}$$

Proof. The definition of the ordering relation of the LoI (Section D.2) and $Q_{1\sim} \sqsubseteq Q_{2\sim}$ would give us:

$$Q_{1\sim} \sqsubseteq Q_{2\sim} \rightarrow \forall db, db' \in \Omega_D \;\; (db\ Q_{1\sim}\ db' \Rightarrow db\ Q_{2\sim}\ db') \tag{1}$$

By the definition of equivalence relations for query sets ($Q_\sim$), for all $db, db' \in \Omega_D$ we have:

$$\begin{aligned}&(db\ Q_{1\sim}\ db' \Rightarrow db\ Q_{2\sim}\ db') \rightarrow \\ &\qquad\qquad \Big((\llbracket q_2 \rrbracket^{db} = \llbracket q_2 \rrbracket^{db'} \forall q_2 \in Q_2) \Rightarrow (\llbracket q_1 \rrbracket^{db} = \llbracket q_1 \rrbracket^{db'} \forall q_1 \in Q_1) \Big)\end{aligned} \tag{2}$$

(1) and (2) would give us:

$$\begin{aligned}&Q_{1\sim} \sqsubseteq Q_{2\sim} \rightarrow \forall db, db' \in \Omega_D \\ &\qquad\qquad \Big((\llbracket q_2 \rrbracket^{db} = \llbracket q_2 \rrbracket^{db'} \forall q_2 \in Q_2) \Rightarrow (\llbracket q_1 \rrbracket^{db} = \llbracket q_1 \rrbracket^{db'} \forall q_1 \in Q_1) \Big)\end{aligned} \tag{3}$$

On the other hand, by the definition of the Determinacy Lattice D.2, we have $\downarrow Q_1 \sqsubseteq \downarrow Q_2 \leftrightarrow Q_1 \preceq Q_2$. From the definition of determinacy ordering, $Q_1 \preceq Q_2$ means $Q_2 \twoheadrightarrow Q_1$. By the definition of query determinacy (Def. D.1) we know that $Q_2 \twoheadrightarrow Q_1$ if:

$$\forall db, db' \in \Omega_D \, \Big((\llbracket q_2 \rrbracket^{db} = \llbracket q_2 \rrbracket^{db'} \forall q_2 \in Q_2) \Rightarrow (\llbracket q_1 \rrbracket^{db} = \llbracket q_1 \rrbracket^{db'} \forall q_1 \in Q_1) \Big) \tag{4}$$

It is evident from (3) and (4) that $\downarrow Q_1 \sqsubseteq \downarrow Q_2 \rightarrow Q_{1\sim} \sqsubseteq Q_{2\sim}$ holds. □

Relying on Def. D.6 to establish the set of equivalence relations derived from a set of sets of queries, we propose following lemma:

Lemma D.7

For any set of sets of queries $\mathbb{Q} \subseteq DL(\mathcal{Q})$, the join of $\mathbb{Q}$ on the DL implies the join of $[\![\mathbb{Q}]\!]$ on the LoI defined on $\{Q_\sim \mid Q \in DL(\mathcal{Q})\}$:

$$\bigsqcup \mathbb{Q} \to \bigsqcup [\![\mathbb{Q}]\!]$$

Proof. Assume there is a set of queries $R \in DL(\mathcal{Q})$ such that $R = \bigsqcup \mathbb{Q}$.

By the definition of the Determinacy Lattice D.3, we have $\bigsqcup \mathbb{Q} = \downarrow \bigcup \mathbb{Q}$ which would give us $R = \downarrow \bigcup \mathbb{Q}$. By the definitions of $\downarrow$ and query determinacy(Def. D.1), it is straightforward to see $(\bigcup \mathbb{Q}) \twoheadrightarrow \downarrow \bigcup \mathbb{Q}$ and $\downarrow \bigcup \mathbb{Q} \twoheadrightarrow (\bigcup \mathbb{Q})$. Replacing $\downarrow \bigcup \mathbb{Q}$ with R, by the definition of query determinacy (Def. D.1) we have $R \twoheadrightarrow (\bigcup \mathbb{Q})$:

$$\forall db, db' \in \Omega_D \left(\forall r \in R.\ [\![r]\!]^{db} = [\![r]\!]^{db'} \to \forall p \in \bigcup \mathbb{Q}.\ [\![p]\!]^{db} = [\![p]\!]^{db'} \right) \tag{1}$$

and $(\bigcup \mathbb{Q}) \twoheadrightarrow R$:

$$\forall db, db' \in \Omega_D \left(\forall p \in \bigcup \mathbb{Q}.\ [\![p]\!]^{db} = [\![p]\!]^{db'} \to \forall r \in R.\ [\![r]\!]^{db} = [\![r]\!]^{db'} \right) \tag{2}$$

(1) and (2) would give us:

$$\forall db, db' \in \Omega_D\ (\forall r \in R.\ [\![r]\!]^{db} = [\![r]\!]^{db'} \leftrightarrow \forall p \in \bigcup \mathbb{Q}.\ [\![p]\!]^{db} = [\![p]\!]^{db'}) \tag{3}$$

Assume $\bigsqcup [\![\mathbb{Q}]\!]$ is an equivalence relation R'. By the definition of the join of the LoI (Section D.2):

$$\bigsqcup [\![\mathbb{Q}]\!] = \forall db, db' \in \Omega_D\ (db\ R'_\sim\ db' \leftrightarrow \forall Q \in \mathbb{Q}.\ db\ Q_\sim\ db')$$

and by the definition of equivalence relations for query sets, for all $db, db' \in \Omega_D$ we have:

$$\begin{aligned} \bigsqcup [\![\mathbb{Q}]\!] = \\ & (\forall r \in R'.\ [\![r]\!]^{db} = [\![r]\!]^{db'} \leftrightarrow \forall Q \in \mathbb{Q}.\ \forall q \in Q.\ [\![q]\!]^{db} = [\![q]\!]^{db'}) = \\ & (\forall r \in R'.\ [\![r]\!]^{db} = [\![r]\!]^{db'} \leftrightarrow \forall p \in \bigcup \mathbb{Q}.\ [\![p]\!]^{db} = [\![p]\!]^{db'}) \end{aligned} \tag{4}$$

(3) and (4) would allow us to conclude $R = R'$, hence $\bigsqcup \mathbb{Q} \to \bigsqcup [\![\mathbb{Q}]\!]$. □

Lemma D.1

For all $\mathcal{Q}$, there is a complete lattice homomorphism from the Determinacy Lattice $DL(\mathcal{Q})$ to the Lattice of Information defined on $\{Q_\sim \mid Q \in DL(\mathcal{Q})\}$.

Proof. To prove this homomorphism, we need to show that the Determinacy Lattice's ordering and join, as well as the top and bottom elements imply their LoI counterparts. Lemmas D.6 and D.7 provide the proofs of ordering and join. The proof for top and bottom elements:

- $\downarrow\mathcal{Q} \rightarrow (\downarrow\mathcal{Q})_\sim$
- $\downarrow\varnothing \rightarrow (\downarrow\varnothing)_\sim$

follows trivially from the definition of $\downarrow$ and $_\sim$. □

Appendix C Determinacy Quantale Axioms

We follow the approach of [181] to prove that our definition of the Determinacy Quantale is indeed a quantale. We begin by defining what is a quantale.

Definition D.11

A quantale is a structure $\langle \mathcal{L}, \sqsubseteq, \vee, \otimes, 1 \rangle$ such that:

1. $\langle \mathcal{L}, \sqsubseteq, \vee \rangle$ is a complete join-semilattice
2. $\langle \mathcal{L}, \otimes, 1 \rangle$ is monoid, that is $\otimes$ is associative and $\forall x \in \mathcal{L}, x \otimes 1 = x = 1 \otimes x$
3. $\otimes$ distributes over $\vee$.

A quantale is called commutative when its $\otimes$ operator is commutative [181].

Next, we prove some lemmas that are later used in the proof of Theorem D.2.

Lemma D.8

Both mix *and* tc *are closure operators.*

Proof. A closure operator is a function $f : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ from the power set of domain A to itself that satisfies the following properties for all sets $X, Y \subseteq A$:

- f is extensive: $X \subseteq \mathrm{f}(X)$

- f is increasing: $X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$
- f is idempotent: $f(f(X)) = f(X)$

It is straightforward to show that both mix and tc satisfy these conditions. □

Definition D.12

For a closure operator $\downarrow$ defined on the domain A, and a function $F : A \to A$, say that F weakly commutes with $\downarrow$ if $F(cl(X)) \subseteq cl(F(X))$ for all $X \subseteq A$.

Lemma D.9

Let $\downarrow : A \to A$ be a closure operator and let $X, Y \subseteq A$. Suppose that $F : A \to A$ weakly commutes with $\downarrow$ and that $G : A \times A \to A$ weakly commutes with $\downarrow$ in each argument. Then:

1. $\downarrow(F(\downarrow(X))) = \downarrow(F(X))$
2. $\downarrow(G(\downarrow(X) \times \downarrow(Y))) = \downarrow(G(X \times Y))$

Proof. Routine, following the properties of closure operator. □

Lemma D.10

Let $\mathbb{P}, \mathbb{Q} \subseteq DL(\mathcal{Q})$, the union operator $\cup$ weakly commutes with tc*:*

$$\mathrm{tc}(\mathrm{tc}(\mathbb{P}) \cup \mathrm{tc}(\mathbb{Q})) = \mathrm{tc}(\mathbb{P} \cup \mathbb{Q})$$

Proof. It suffices to show $\mathbb{R} \in \mathrm{tc}(\mathrm{tc}(\mathbb{P}) \cup \mathrm{tc}(\mathbb{Q}))$ iff $\mathbb{R} \in \mathrm{tc}(\mathbb{P} \cup \mathbb{Q})$, which follows easily from the definitions of $\cup$ and tc. □

Lemma D.11

The join operator of DL weakly commutes with tc *in each argument*

Proof. Let $P, Q \in DL(\mathcal{Q})$, and let $\mathbb{S} \subseteq DL(\mathcal{Q})$. If $Q_\sim$ is tiled by $[\![\mathbb{S}]\!]$ then $P_\sim \sqcup Q_\sim$ is tiled by $\{P_\sim \sqcup R \mid R \in [\![\mathbb{S}]\!]\}$. This follows easily from the definition of the equivalence relation induced by a query (i.e. $\sim$), mix, Lemma D.7 and the fact that $[P_\sim \sqcup Q_\sim] = \{A \cap B \mid A \in [P_\sim], B \in [Q_\sim]\} \setminus \varnothing$. □

Lemma D.12

Given two sets of sets of queries $\mathbb{Q}, \mathbb{P} \subseteq DL(\mathcal{Q})$ *it holds that:*

$$\mathrm{tc}(\mathbb{Q}) \otimes \mathrm{tc}(\mathbb{P}) = \mathbb{Q} \otimes \mathbb{P}$$

Proof. By Lemma D.11 we know that the join operator of DL weakly commutes with tc in each argument. We apply this lemma to the definition of $\otimes$ operator:

$$\begin{aligned}
&\mathrm{tc}(\mathbb{Q}) \otimes \mathrm{tc}(\mathbb{P}) = \\
&\mathrm{tc}(\bigcup_{Q \in \mathrm{tc}(\mathbb{Q}), P \in \mathrm{tc}(\mathbb{P})} (Q \sqcup P)) = \\
&\mathrm{tc}(\bigcup_{Q \in \mathbb{Q}, P \in \mathbb{P}} (Q \sqcup P)) = \\
&\mathbb{Q} \otimes \mathbb{P}
\end{aligned}$$

□

Lemma D.13

Given two sets of sets of queries $\mathbb{Q}, \mathbb{P} \subseteq DL(\mathcal{Q})$ *it holds that:*

$$\mathbb{Q} \vee \mathbb{P} = \mathbb{P} \vee \mathbb{Q}$$

Proof. Follows directly from the definition of $\vee$ in the DL and the commutativity of union operator $\cup$.

$$\begin{aligned}
&\mathbb{Q} \vee \mathbb{P} = \\
&{\downarrow}(\mathbb{Q} \cup \mathbb{P}) = \\
&{\downarrow}(\mathbb{P} \cup \mathbb{Q}) = \\
&\mathbb{P} \vee \mathbb{Q}
\end{aligned}$$

□

Now, we show that DQ in Def. D.5 is a quantale.

Theorem D.2

The Determinacy Quantale is a commutative quantale.

Proof. We have to show that our definition of Determinacy Quantale respects the quantale axioms of Def. D.11.

1. Showing $\langle \mathcal{I}, \sqsubseteq, \vee \rangle$ is a complete join-semilattice is straightforward following Lemma D.8 and the fact that tc is a closure operator.

2. We should show that $\otimes$ is associative and 1 is a unit:

   a. For the associativity of $\otimes$ we need to show that $\mathbb{P} \otimes (\mathbb{Q} \otimes \mathbb{R}) = (\mathbb{P} \otimes \mathbb{Q}) \otimes \mathbb{R}$. Here we rely on Lemmas D.11 and D.12 to eliminate the nested uses of tc and the basic properties of $\cup$ operator to show that both sides of $\mathbb{P} \otimes (\mathbb{Q} \otimes \mathbb{R}) = (\mathbb{P} \otimes \mathbb{Q}) \otimes \mathbb{R}$ can be reduced to identical expressions. Left side:

   $$\begin{aligned}
   &\mathbb{P} \otimes (\mathbb{Q} \otimes \mathbb{R}) = \\
   &\mathbb{P} \otimes \mathrm{tc}(\bigcup_{Q \in \mathbb{Q}, R \in \mathbb{R}} (Q \sqcup R)) = \\
   &\mathbb{P} \otimes (\bigcup_{Q \in \mathbb{Q}, R \in \mathbb{R}} (Q \sqcup R)) = \\
   &\mathrm{tc}(\bigcup_{P \in \mathbb{P}, T \in (\bigcup_{Q \in \mathbb{Q}, R \in \mathbb{R}} (Q \sqcup R))} (P \sqcup T)) = \\
   &\mathrm{tc}(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}, R \in \mathbb{R})} (P \sqcup Q \sqcup R)) \qquad (1)
   \end{aligned}$$

   Right Side:

   $$\begin{aligned}
   &(\mathbb{P} \otimes \mathbb{Q}) \otimes \mathbb{R} = \\
   &\mathrm{tc}(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}} (P \sqcup Q)) \otimes \mathbb{R} = \\
   &(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}} (P \sqcup Q)) \otimes \mathbb{R} = \\
   &\mathrm{tc}(\bigcup_{T \in (\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}} (P \sqcup Q)), R \in \mathbb{R}} (T \sqcup R)) = \\
   &\mathrm{tc}(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}, R \in \mathbb{R})} (P \sqcup Q \sqcup R)) \qquad (2)
   \end{aligned}$$

   By (1) and (2) we can conclude that $\mathbb{P} \otimes (\mathbb{Q} \otimes \mathbb{R}) = (\mathbb{P} \otimes \mathbb{Q}) \otimes \mathbb{R}$.

   b. To show that $1 = \varnothing$ is a unit for $\otimes$ we need to show that $\forall x \in \mathcal{I}, x \otimes 1 = x = 1 \otimes x$. Using $\varnothing$ as the unit, and applying the definition of $\otimes$ will give us:

   $$\mathbb{Q} \otimes \varnothing = \mathrm{tc}(\bigcup_{Q \in \mathbb{Q}} (Q)) = \mathrm{tc}(\mathbb{Q}) = \mathbb{Q}$$

   which following the associativity of $\otimes$ gives us $\forall x \in \mathcal{I}, x \otimes \varnothing = x = \varnothing \otimes x$.

3. To establish distributivity we need to show that $\mathbb{P} \otimes (\mathbb{Q} \vee \mathbb{R}) = (\mathbb{P} \otimes \mathbb{Q}) \vee (\mathbb{P} \otimes \mathbb{R})$. We again rely on Lemmas D.11 and D.12 and basic properties of $\cup$ to show:

$$\begin{aligned}
&\mathbb{P} \otimes (\mathbb{Q} \vee \mathbb{R}) = \\
&\mathbb{P} \otimes \mathrm{tc}(\mathbb{Q} \cup \mathbb{R}) = \\
&\mathbb{P} \otimes (\mathbb{Q} \cup \mathbb{R}) = \\
&\mathrm{tc}(\bigcup_{P \in \mathbb{P}, T \in (\mathbb{Q} \cup \mathbb{R})} (P \sqcup T)) = \\
&\mathrm{tc}(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}} (P \sqcup Q) \cup \bigcup_{P \in \mathbb{P}, R \in \mathbb{R}} (P \sqcup R)) = \\
&\mathrm{tc}((\mathbb{P} \otimes \mathbb{Q}) \cup (\mathbb{P} \otimes \mathbb{R})) = \\
&(\mathbb{P} \otimes \mathbb{Q}) \vee (\mathbb{P} \otimes \mathbb{R})
\end{aligned}$$

4. Commutativity of $\otimes$ is inherited directly from Lemma D.13 and the commutativity of $\sqcup$ in DL.

$$\begin{aligned}
&\mathbb{P} \otimes \mathbb{Q} = \\
&\mathrm{tc}(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}} (P \sqcup Q)) = \\
&\mathrm{tc}(\bigcup_{P \in \mathbb{P}, Q \in \mathbb{Q}} (Q \sqcup P)) = \\
&\mathbb{Q} \otimes \mathbb{P}
\end{aligned}$$

□

Appendix D Relation Between DQ and QoI

We first provide some auxiliary lemmas, and then proceed to prove Lemma D.2.

Lemma D.14

Given sets of sets of queries $\mathbb{Q}, \mathbb{P} \subseteq DL(\mathcal{Q})$, $\mathrm{tc}(\mathbb{Q}) \subseteq \mathrm{tc}(\mathbb{P})$ *on the DQ implies* $[\![\mathrm{tc}(\mathbb{Q})]\!] \subseteq [\![\mathrm{tc}(\mathbb{P})]\!]$ *on the QoI defined on* $\{[\![\mathbb{Q}]\!] \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$.

Proof. Trivial from the Def. D.6. □

Lemma D.15

$\bigvee_i \mathbb{P}_i$ *on the DQ implies* $\bigvee_i [\![\mathbb{P}_i]\!]$ *on the QoI defined on* $\{[\![\mathbb{Q}]\!] \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$.

Proof. Trivial from the Def. D.6. □

Lemma D.16

Given sets of sets of queries $\mathbb{Q}, \mathbb{P} \subseteq DL(\mathcal{Q})$, $\mathrm{tc}(\mathbb{Q}) \otimes \mathrm{tc}(\mathbb{P})$ *on the DQ implies* $[\![\mathrm{tc}(\mathbb{Q})]\!] \otimes [\![\mathrm{tc}(\mathbb{P})]\!]$ *on the QoI defined on* $\{[\![\mathbb{Q}]\!] \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$.

Proof. Follows trivially from Def. D.6 and Lemma D.7. □

Lemma D.2

For all $\mathcal{Q}$*, there is a quantale homomorphism from the Determinacy Quantale* $DQ(\mathcal{Q})$ *to the Quantale of Information defined on* $\{[\![\mathbb{Q}]\!] \mid \mathbb{Q} \subseteq DL(\mathcal{Q})\}$.

Proof. To prove this homomorphism, we need to show that the Determinacy Quantale's ordering, join and tensor, as well as the top and bottom elements imply their QoI counterparts. Lemmas D.14, D.15, and D.16 provide the proofs of ordering, join and tensor, respectively. The proof of the top element:

- $DL(\mathcal{Q}) \to LoI([\![\mathcal{Q}]\!])$

follows from Def. D.6 and Lemma D.1, and the proof of the bottom element:

- $\varnothing \to \varnothing$

is trivial. □

Appendix E Correctness of Dependency Analysis

To show that the diagram in Figure D.11 commutes, we aim to show commutativity for each cell in it. In this section, we establish this for the bottommost cell of it. To that end, we need to establish that the QoI point $[\![\mathrm{QL}_u]\!]$ that corresponds to the query list $\mathrm{QL}_u = \{Q_1, \ldots, Q_n\}$ extracted from a program prg by the dependency analysis is an upper bound on the knowledge relation $[\![\mathrm{prg}_u]\!]$ induced by prg.

The basic outline of the argument rests on identifying a particular *single* equivalence relation $k(\mathrm{QL}, \mathrm{prg}) \in \mathrm{mix}([Q_{1\sim}], \ldots, [Q_{n\sim}])$, which satisfies $[\![\mathrm{prg}]\!] \sqsubseteq k(\mathrm{QL}, \mathrm{prg})$. Intuitively, this relation captures how much information the program could leak at most if it output the full result of every query that its output depends on. As long as the analysis is sound, this is an instantiation of the disjunction represented by QL, with each disjunct selected precisely for those starting configurations where the program's output turns out to depend on the queries enumerated in that disjunct.

For a fixed program prg and user u, we assume the existence of a function $Q = Q_{\text{prg},u}$ from databases $db \in \Omega_D$ to sets of queries, which returns the set of those queries performed when executing prg on database db whose result taints some output to the user u. We formally define the function Q by relying on a taint analysis.

Taint analysis. The semantics of the taint analysis enriches the normal operational semantics of the language in the sense that it has transitions whenever the operational one does, and acts the same on those components of a configuration that exist in the operational one; so runs in it can be put in one-to-one correspondence to operational ones.

The rules of the taint analysis presented in Figure D.12 are fairly straightforward. We use mapping Δ to map each variable to a set of dependencies of variables and queries.

The rules for `if` rely on auxiliary command `set` pc `to` δ to restore the dependency set of pc to its previous state ($\Delta(pc)$) upon exiting the `if` branch. We sequentially composite this command with the body of `if` to ensure its execution after leaving the `if` branch's body. The rules for `while` use `set` pc `to` δ in a similar manner.

The rule TA-OUTPUT uses $fv(e)$ to extract all the variables of expression e, and relies on the union of the Δs of those variables to calculate β, which is the set of dependencies the execution up to this output, depended on.

We extend the definition of trace τ to a sequence of observations of the form $\langle vl, u, \beta \rangle$, and use the notation $\tau \Downarrow_u$ to denote the sequence of all βs in τ that u can observe. We use this notation to define function Q as follows:

Definition D.13

Given a database state db and user u, such that $\langle c, m_0, db \rangle \overset{\tau}{\Longrightarrow}_u$, $Q(db)$ is defined as $\{\beta \mid \beta \in \tau \Downarrow_u\}$

A proof of Lemma D.3 can then proceed by a straightforward induction on the semantics.

In Def. D.13 we formally define the function Q. This function satisfies a closure property that informally states that if on two given databases the output depended on different sets of queries, then the choice of the set of dependencies itself must have been due to the outcome of a query which is among the dependencies in both databases and evaluates to a different result.

Lemma D.17

For all $db, db' \in \Omega_D$, if $Q(db) \neq Q(db')$, then there exists a particular query $q \in Q(db) \cap Q(db')$ such that $[\![q]\!]^{db} \neq [\![q]\!]^{db'}$.

TA-Skip

$$\frac{}{\langle \Delta, \texttt{skip}, m, db\rangle \xrightarrow{\epsilon} \langle \Delta, \epsilon, m, db\rangle}$$

TA-Assign

$$\frac{\langle e, m, db\rangle \downarrow vl \qquad m' = m[x \mapsto vl] \qquad \Delta' = \Delta[x \mapsto \Delta(pc) \cup \bigcup_{x \in fv(e)} \Delta(x)]}{\langle \Delta, x := e, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', \epsilon, m', db\rangle}$$

TA-QueryEval

$$\frac{vl = [\![q]\!]^{db} \qquad m' = m[x \mapsto vl] \qquad \Delta' = \Delta[x \mapsto \Delta(pc) \cup q]}{\langle \Delta, x \leftarrow q, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', \epsilon, m', db\rangle}$$

TA-IfTrue

$$\frac{\langle e, m, db\rangle \downarrow n \qquad n \neq 0 \qquad c_1' = c_1; \texttt{set}\ pc\ \texttt{to}\ \Delta(pc) \qquad \Delta' = \Delta[pc \mapsto \Delta(pc) \cup \bigcup_{x \in fv(e)} \Delta(x)]}{\langle \Delta, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', c_1', m, db\rangle}$$

TA-IfFalse

$$\frac{\langle e, m, db\rangle \downarrow n \qquad n = 0 \qquad c_2' = c_2; \texttt{set}\ pc\ \texttt{to}\ \Delta(pc) \qquad \Delta' = \Delta[pc \mapsto \Delta(pc) \cup \bigcup_{x \in fv(e)} \Delta(x)]}{\langle \Delta, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', c_2', m, db\rangle}$$

TA-WhileTrue

$$\frac{\begin{array}{c}\langle e, m, db\rangle \downarrow n \qquad n \neq 0 \\ c' = c; \texttt{while}\ e\ \texttt{do}\ c; \texttt{set}\ pc\ \texttt{to}\ \Delta(pc) \\ \Delta' = \Delta[pc \mapsto \Delta(pc) \cup \bigcup_{x \in fv(e)} \Delta(x)]\end{array}}{\langle \Delta, \texttt{while}\ e\ \texttt{do}\ c, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', c', m, db\rangle}$$

TA-WhileFalse

$$\frac{\begin{array}{c}\langle e, m, db\rangle \downarrow n \qquad n = 0 \\ c' = \texttt{set}\ pc\ \texttt{to}\ \Delta(pc) \\ \Delta' = \Delta[pc \mapsto \Delta(pc) \cup \bigcup_{x \in fv(e)} \Delta(x)]\end{array}}{\langle \Delta, \texttt{while}\ e\ \texttt{do}\ c, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', \epsilon, m, db\rangle}$$

TA-Seq

$$\frac{\langle \Delta, c_1, m, db\rangle \xrightarrow{\alpha} \langle \Delta', c_1', m', db'\rangle}{\langle \Delta, c_1; c_2, m, db\rangle \xrightarrow{\alpha} \langle \Delta', c_1'; c_2, m', db'\rangle}$$

TA-SeqEmpty

$$\frac{}{\langle \Delta, \epsilon; c, m, db\rangle \xrightarrow{\epsilon} \langle \Delta, c, m, db\rangle}$$

TA-Output

$$\frac{\langle e, m, db\rangle \downarrow vl \qquad \beta = \Delta(pc) \cup \bigcup_{x \in fv(e)} \Delta(x)}{\langle \Delta, \texttt{out}(e, u), m, db\rangle \xrightarrow{\langle vl, u, \beta\rangle} \langle \Delta, \epsilon, m, db\rangle}$$

TA-SetPC

$$\frac{\Delta' = \Delta[pc \mapsto \delta]}{\langle \Delta, \texttt{set}\ pc\ \texttt{to}\ \delta, m, db\rangle \xrightarrow{\epsilon} \langle \Delta', \epsilon, m, db\rangle}$$

Figure D.12: Taint analysis rules

We say that database states db and db' are equivalent with respect to a dependency set S (written as $db \approx_S db'$) iff $[\![y]\!]^{db} = [\![y]\!]^{db'}$ for all $y \in S$ where $y \in \mathcal{Q}$.

Lemma D.18

For all states db_1 and db_2 and users u, if $\langle c, m_0, db_1\rangle \overset{t_1}{\Longrightarrow}_u$, $\langle c, m_0, db_2\rangle \overset{t_2}{\Longrightarrow}_u$, $Q \triangleq Q(db_1) = Q(db_2)$ and $db_1 \approx_Q db_2$, then $t_1\!\downharpoonright_u = t_2\!\downharpoonright_u$.

We then define $k(\mathrm{QL}_u, \mathrm{prg})$ as the equivalence relation

$$\{(db, db') \in \Omega_D^2 \mid Q(db) = Q(db') \wedge (db, db') \in Q(db)_\sim\},$$

that is, we partition each respective subset of databases db that shares one set of queries $Q(db)$ into equivalence classes according to the knowledge relation induced by $Q(db)$.

Lemma D.19

$[\![\mathrm{prg}]\!]_u \sqsubseteq k(\mathrm{QL}_u, \mathrm{prg}) \sqsubseteq [\![\mathrm{QL}_u]\!]$.

Proof.

- $k(\mathrm{QL}_u, \mathrm{prg}) \sqsubseteq [\![\mathrm{QL}_u]\!]$: Will in fact show that $k(\mathrm{QL}_u, \mathrm{prg}) \in \mathrm{mix}([Q_{1\sim}], \ldots, [Q_{n\sim}])$, where $\mathrm{QL}_u = \{Q_1, \ldots, Q_n\}$. For that, it suffices to show that every equivalence class $x \in [k(\mathrm{QL}_u, \mathrm{prg})]$ is also an equivalence class of one of the Q_i. Let $db \in x$ be arbitrary. Then claim that $x \in [Q(db)]$, which suffices since by Lemma D.3, $Q(db)$ is one of the Q_i. To establish this, just need to show that $Q(db) = Q(db')$ for all db' such that $(db, db') \in Q(db)_\sim$, so that $(db, db') \in k(\mathrm{QL}_u, \mathrm{prg})$ as well. But this follows from Lemma D.17: if some db' has $(db, db') \in Q(db)_\sim$, then $[\![q]\!]^{db} = [\![q]\!]^{db'}$ for all $q \in Q(db)$, but then we must not have $Q(db) \neq Q(db')$.

- $[\![\mathrm{prg}]\!]_u \sqsubseteq k(\mathrm{QL}_u, \mathrm{prg})$: Straightforward application of Lemma D.18.

□

Appendix F Query Analysis

Symbolic Tuple Ordering

To show that the symbolic tuples ordering of Def. D.9 induces a determinacy order and prove Lemma D.20 we first need to define the evaluation of a symbolic tuple in a database state.

Symbolic tuple evaluation. The evaluation of a symbolic tuple $\langle T, \phi, \pi\rangle$ in the database state db written as $[\![\langle T, \phi, \pi\rangle]\!]^{db}$ is a π-projection on the set of db's tuples defined on the join of tables in T that satisfy the constraint ϕ. Formally:

Definition D.14

Given database state db and symbolic tuple $\langle T, \phi, \pi\rangle$, $[\![\langle T, \phi, \pi\rangle]\!]^{db}$ is defined as:

$$\{tp\lfloor_{\pi} \mid tp \in \prod_{t \in T} [\![t]\!]^{db}, tp \models \phi\}$$

where $tp\lfloor_{\pi}$ is a tuple with its columns limited to those in π, and $tp \models \phi$ means that tuple tp satisfies formula ϕ.

We proceed to prove Lemma D.20.

Lemma D.20

Given two sets of queries Q_1 and Q_2, if $\mathrm{sts}(Q_1) \sqsubseteq_{\mathrm{st}} \mathrm{sts}(Q_2)$ *then* $Q_1 \preceq Q_2$.

Proof. Assume $\ell_{Q_1} = \mathrm{sts}(Q_1)$ and $\ell_{Q_2} = \mathrm{sts}(Q_2)$. By Def. D.9 we want to show that if for all symbolic tuples $\langle T, \phi, \pi\rangle \in \ell_{Q_1}$, there is a set of well-formed symbolic tuples $S = \langle T_1, \phi_1, \pi_1\rangle, ..., \langle T_n, \phi_n, \pi_n\rangle$ such that $S \subseteq \ell_{Q_2}$, $T_1, ..., T_n$ are disjoint, $T \subseteq (T_1 \cup ... \cup T_n)$, $\phi \models (\phi_1 \wedge ... \wedge \phi_n)$, and $\mathrm{dep}(\phi) \cup \pi \subseteq (\pi_1 \cup ... \cup \pi_n)$, then $Q_1 \preceq Q_2$.

We assume an intermediate symbolic tuple $\mathrm{st}_{\mathrm{itr}}$ and define it as $\langle T_1 \cup ... \cup T_n, \phi_1 \wedge ... \wedge \phi_n, \pi_1 \cup ... \cup \pi_n\rangle$. $\mathrm{st}_{\mathrm{itr}}$ models the symbolic tuples created from the join of $\langle T_1, \phi_1, \pi_1\rangle, ..., \langle T_n, \phi_n, \pi_n\rangle$. Additionally, $T_1, ..., T_n$ are disjoint, which by the definition of symbolic tuples means that $\pi_1, ..., \pi_n$ and the dependencies of $\phi_1, ..., \phi_n$ are also disjoint, effectively making $\mathrm{st}_{\mathrm{itr}}$ the symbolic tuple of the Cartesian product of tuples $\langle T_1, \phi_1, \pi_1\rangle, ..., \langle T_n, \phi_n, \pi_n\rangle$.

We want to show that the symbolic tuples in S can determine $\mathrm{st}_{\mathrm{itr}}$:

$$\forall db_1, db_2 \in \Omega_D.\ [\![\mathrm{st}]\!]^{db_1} = [\![\mathrm{st}]\!]^{db_2}\ \forall \mathrm{st} \in S \rightarrow [\![\mathrm{st}_{\mathrm{itr}}]\!]^{db_1} = [\![\mathrm{st}_{\mathrm{itr}}]\!]^{db_2} \tag{1}$$

For a specific database state db, $[\![\mathrm{st}_{\mathrm{itr}}]\!]^{db}$ would give us all the tuples defined on $T_1, ..., T_n$ satisfying $\phi_1 \wedge ... \wedge \phi_n$ and projected on the columns in $\pi_1 \cup ... \cup \pi_n$.

Assume there is a pair of databases $db_1, db_2 \in \Omega_D$ such that $[\![\mathrm{st}]\!]^{db_1} = [\![\mathrm{st}]\!]^{db_2}\ \forall \mathrm{st} \in S$ holds but $[\![\mathrm{st}_{\mathrm{itr}}]\!]^{db_1} \neq [\![\mathrm{st}_{\mathrm{itr}}]\!]^{db_2}$. By the assumption $[\![\mathrm{st}]\!]^{db_1} = [\![\mathrm{st}]\!]^{db_2}\ \forall \mathrm{st} \in S$ we know that for all $\mathrm{st} \in S$, if tuple tp is in $[\![\mathrm{st}]\!]^{db_1}$ it is also in $[\![\mathrm{st}]\!]^{db_2}$, and vice versa.

For $[\![\mathrm{st}_{\mathrm{itr}}]\!]^{db_1} \neq [\![\mathrm{st}_{\mathrm{itr}}]\!]^{db_2}$ to hold, we have to consider two cases:

1. There is a tuple $tp \in [\![\text{st}_{\text{itr}}]\!]^{db_1}$ such that tp cannot be constructed from the tuples in set $\{tp' \in [\![\text{st}]\!]^{db_1} \mid \text{st} \in S\}$

   - All of the symbolic tuples st $\in S$ are well-formed and $T_1, ..., T_n$ are disjoint, which makes st_{itr} the symbolic tuple of the Cartesian product of S. This means that tuple $tp \in [\![\text{st}_{\text{itr}}]\!]^{db_1}$ is defined on the product of tables $T_1, ..., T_n$, satisfies $\phi_1 \wedge ... \wedge \phi_n$, and projected on $\pi_1 \cup ... \cup \pi_n$. Which means that each tuple $tp \in [\![\text{st}_{\text{itr}}]\!]^{db_1}$ is constructed from the merge of tuples $tp_1, ..., tp_n$ where $tp_i \in [\![\langle T_i, \phi_i, \pi_i \rangle]\!]^{db_1}$ for $i = 1, ..., n$. Thus, this case is not possible.

2. There is a tuple $[\![\text{st}_{\text{itr}}]\!]^{db_2}$ such that tp cannot be constructed from the tuple set $\{tp' \in [\![\text{st}]\!]^{db_2} \mid \text{st} \in S\}$

   - Similar to the first case.

Next, we need to show that st_{itr} determines $\langle T, \phi, \pi \rangle$:

$$\forall db_1, db_2 \in \Omega_D.\ [\![\text{st}_{\text{itr}}]\!]^{db_1} = [\![\text{st}_{\text{itr}}]\!]^{db_2} \to [\![\langle T, \phi, \pi \rangle]\!]^{db_1} = [\![\langle T, \phi, \pi \rangle]\!]^{db_2} \qquad (2)$$

By $[\![\text{st}_{\text{itr}}]\!]^{db_1} = [\![\text{st}_{\text{itr}}]\!]^{db_2}$ we know $\forall tp_1 \in [\![\text{st}_{\text{itr}}]\!]^{db_1}, \exists tp_2 \in [\![\text{st}_{\text{itr}}]\!]^{db_2}$ and $tp_1 = tp_2$, and $\forall tp_2 \in [\![\text{st}_{\text{itr}}]\!]^{db_2}, \exists tp_1 \in [\![\text{st}_{\text{itr}}]\!]^{db_1}$ and $tp_2 = tp_1$.

Intuitively, for a given database db, $[\![\text{st}_{\text{itr}}]\!]^{db}$ has has more columns and tuples than $[\![\langle T, \phi, \pi \rangle]\!]^{db}$. Symbolic tuple $\langle T, \phi, \pi \rangle$ throws away some columns by limiting the resulting tuples to tables in T which is a subset of $T_1 \cup ... \cup T_n$ and projecting on π which is a subset of $\pi_1 \cup ... \cup \pi_n$. It also eliminate some rows by applying ϕ to the result set, which is stronger than $\phi_1 \wedge ... \wedge \phi_n$.

We need to show that applying these limitations maintains query determinacy. We consider these cases separately:

Columns: Projecting away some columns from the evaluation of st_{itr} is going to maintain query determinacy. We denote by $tp\downharpoonright_{\pi}$, projecting tuple tp to only columns specified in π, additionally we use the notation $\text{col}(T)$ to indicate the columns of T. We use the same notation for tuples and write $\text{col}(tp)$ to denote the set of columns of tuple tp. For a tuple tp such that $tp \in [\![\text{st}_{\text{itr}}]\!]^{db_1}$ and $tp \in [\![\text{st}_{\text{itr}}]\!]^{db_2}$, by projecting away some columns from tp we end up with a new tuple $tp' = tp\downharpoonright_{\pi}$ such that $\text{col}(tp') \subseteq \text{col}(tp)$. Since tp is in both $[\![\text{st}_{\text{itr}}]\!]^{db_1}$ and $[\![\text{st}_{\text{itr}}]\!]^{db_2}$, and by the definition of ordering $\pi \subseteq \pi_1 \cup ... \cup \pi_n$, we can conclude tp' will also be in both $[\![\langle T_1 \cup ... \cup T_n, \phi_1 \wedge ... \wedge \phi_n, \pi \rangle]\!]^{db_1}$ and $[\![\langle T_1 \cup ... \cup T_n, \phi_1 \wedge ... \wedge \phi_n, \pi \rangle]\!]^{db_2}$, this follows easily from Def. D.14.

Rows: Removing some rows from the last step is going to maintain query determinacy. By the definition of ordering we know that $\text{dep}(\phi) \cup \pi \subseteq \pi_1 \cup ... \cup \pi_n$ and that $\langle T_1, \phi_1, \pi_1 \rangle, ..., \langle T_n, \phi_n, \pi_n \rangle$ are well-formed, which means that ϕ

only applies to the columns that were retrieved by the intermediate tuple (projected to π). Since ϕ is a stronger condition than $\phi_1 \wedge \ldots \wedge \phi_n$, for a tuple tp such that $tp \in [\![\langle T_1 \cup \ldots \cup T_n, \phi_1 \wedge \ldots \wedge \phi_n, \pi\rangle]\!]^{db_1}$ and $tp \in [\![\langle T_1 \cup \ldots \cup T_n, \phi_1 \wedge \ldots \wedge \phi_n, \pi\rangle]\!]^{db_2}$, if tp satisfies ϕ then tp would also be in both $[\![\langle T_1 \cup \ldots \cup T_n, \phi, \pi\rangle]\!]^{db_1}$ and $[\![\langle T_1 \cup \ldots \cup T_n, \phi, \pi\rangle]\!]^{db_2}$. Otherwise, if tp is not in one of then, it is not going to be in the other one either.

Tables: Similar to the first case, for a tuple tp such that $tp \in [\![\langle T_1 \cup \ldots \cup T_n, \phi, \pi\rangle]\!]^{db_1}$ and $tp \in [\![\langle T_1 \cup \ldots \cup T_n, \phi, \pi\rangle]\!]^{db_2}$, by projecting away the columns of some of the tables from tp we end up with a new tuple $tp' = tp|_{\text{col}(T)}$. Since tp is in both $[\![\langle T_1 \cup \ldots \cup T_n, \phi_1 \wedge \ldots \wedge \phi_n, \pi\rangle]\!]^{db_1}$ and $[\![\langle T_1 \cup \ldots \cup T_n, \phi_1 \wedge \ldots \wedge \phi_n, \pi\rangle]\!]^{db_2}$, and by Def. D.9 $T \subseteq T_1 \cup \ldots \cup T_n$, we can conclude t' will also be in both $[\![\langle T, \phi, \pi\rangle]\!]^{db_1}$ and $[\![\langle T, \phi, \pi\rangle]\!]^{db_2}$.

(1) and (2) would give us:

$$\forall db_1, db_2 \in \Omega_D.\ [\![\text{st}]\!]^{db_1} = [\![\text{st}]\!]^{db_2}\ \ \forall \text{st} \in S \rightarrow [\![\langle T, \phi, \pi\rangle]\!]^{db_1} = [\![\langle T, \phi, \pi\rangle]\!]^{db_2}$$

which allows us to conclude $\langle T_1, \phi_1, \pi_1\rangle \ldots \langle T_n, \phi_n, \pi_n\rangle$ determines $\langle T, \phi, \pi\rangle$.

Repeating this process for all of the symbolic tuples in ℓ_{Q_1} would give us $Q_2 \twoheadrightarrow Q_1$ which means $Q_1 \preceq Q_2$. □

Symbolic Tuple and DQ Ordering

We present the proof of Lemma D.4.

Lemma D.4

If $\sigma_{\text{st}}(\{Q_1, \ldots, Q_n\}) \sqsubseteq_* \sigma_{\text{st}}(\{P_1, \ldots, P_m\})$*, then in the DQ,* $(Q_1 \vee \ldots \vee Q_n) \sqsubseteq (P_1 \vee \ldots \vee P_m)$.

Proof. Assume $\sigma_{\text{st}}(\{Q_1, \ldots, Q_n\}) = \{\ell_{Q_1}, \ldots, \ell_{Q_n}\}$ and $\sigma_{\text{st}}(\{P_1, \ldots, P_m\}) = \{\ell_{P_1}, \ldots, \ell_{P_m}\}$. We have $\{\ell_{Q_1}, \ldots, \ell_{Q_n}\} \sqsubseteq_* \{\ell_{P_1}, \ldots, \ell_{P_m}\}$.

By the definition of $\sqsubseteq_*$ and Lemma D.20, we know that for each Q_i in $\{Q_1, \ldots, Q_n\}$ there is at least one P_j in $\{P_1, \ldots, P_m\}$ such that $Q_i \preceq P_j$.

We apply tc to Q_i and P_j which would give us $\text{tc}(Q_i) \subseteq \text{tc}(P_j)$. By applying tc to every element of $\{P_1, \ldots, P_m\}$, using the basic properties of $\cup$ we will have $\text{tc}(Q_i) \subseteq \text{tc}(P_1) \cup \ldots \cup \text{tc}(P_m)$ for all $i \in \{1, \ldots, n\}$.

Since the tiling closure of each Q_i is individually less than $\text{tc}(P_1) \cup \ldots \cup \text{tc}(P_m)$, their union would still be less that $\text{tc}(P_1) \cup \ldots \cup \text{tc}(P_m)$ which gives us:

$$\text{tc}(Q_1) \cup \ldots \cup \text{tc}(Q_n) \subseteq \text{tc}(P_1) \cup \ldots \cup \text{tc}(P_m) \tag{1}$$

We apply the tiling closure to both sides of (1) and rely on Lemma D.10 to remove the nested uses of tc, which would give us:

$$\mathrm{tc}(Q_1 \cup ... \cup Q_n) \subseteq \mathrm{tc}(P_1 \cup ... \cup P_m)$$

which by the definition of $\vee$ in the DQ would mean $(Q_1 \vee ... \vee Q_n) \sqsubseteq (P_1 \vee ... \vee P_m)$ $\square$

Paper E

Securing P4 Programs by Information Flow Control

Anoud Alshnakat, Amir M. Ahmadian, Musard Balliu,
Roberto Guanciale, and Mads Dam

Abstract

Software-Defined Networking (SDN) has transformed network architectures by decoupling the control and data planes, enabling fine-grained control over packet processing and forwarding. P4, a language designed for programming data plane devices, allows developers to define custom packet processing behaviors directly on programmable network devices. This provides greater control over packet forwarding, inspection, and modification. However, the increased flexibility provided by P4 also brings significant security challenges, particularly in managing sensitive data and preventing information leakage within the data plane.

This paper presents a novel security type system for analyzing information flow in P4 programs by combining security types with interval analysis. The proposed type system allows the specification of security policies in terms of input packet bits rather than program variables. We formalize this type system and prove it sound, guaranteeing that well-typed programs satisfy noninterference. Our prototype implementation Tap4s, evaluated on several use cases, demonstrates the effectiveness of this approach in detecting security violations and information leakages.

E.1 Introduction

Software-Defined Networking (SDN) [127] is a software-driven approach to networking that enables programmatic control of network configuration and packet processing rules. SDN achieves this by decoupling the routing process, performed in the control-plane, from the forwarding process performed in the data-plane. The control-plane is often implemented by a logically-centralized SDN controller that is responsible for network configuration and setting forwarding rules. The data-plane consists of network devices, such as programmable switches, that process and forward packets based on instructions received from the control-plane. Before SDN, hardware providers had complete control over the supported functionalities of the devices, leading to lengthy development cycles and delays in deploying new features. SDN has shifted this paradigm, allowing application developers and network engineers to implement specific network behaviors, such as deep packet inspection, load balancing, and VPNs, and execute them directly on networking devices.

Network Functions Virtualization (NFV) further expands upon this concept, enabling the deployment of multiple virtual data-planes over a single physical infrastructure [118]. SDN and NFV together offer increased agility and optimization, making them cornerstones of future network architectures. Complementing this evolution, the Programming Protocol-independent Packet Processors (P4) [105] domain-specific language has emerged as a leading standard for programming the data-plane's programmable devices, such as FPGAs and switches. Additionally, P4 serves as a specification language to define the behavior of the switches as it provides a suitable level of abstraction, yet detailed enough to accurately capture the behavior of the switch. It maintains a level of simplicity and formalism that allows for effective automated analysis [189].

NFVs and SDNs introduce new security challenges that extend beyond the famous and costly outages caused by network misconfigurations[169]. Many data-plane applications process sensitive data, such as cryptographic keys and internal network topologies. The complexity of these applications, the separation of ownership of platform and data-plane in virtualized environments, and the integration of third-party code, facilitate undetected information leakages. Misconfiguration may deliver unencrypted packets to public network, bugs may leak sensitive packet metadata or routing configurations that expose internal network topology, and malicious code may build covert channels to exfiltrate data via legitimate packet fields such as TCP sequence number and TTL [50].

In this domain, the core challenge lies in the data dependency of what is observable, what is secret, and the packet forwarding behavior. An attacker may be able to access only packets belonging to a specific subnetwork, only packets for a specific network protocol may be secret, and switches may drop packets based on the matching of their fields with routing configurations. These data dependencies make information leakage a complex problem to address in SDN-driven networks.

Existing work in the area of SDN has focused on security of routing configurations by analyzing network flows that are characterized by port numbers and endpoints. However, these works ignore indirect flows that may leak information via other packet fields. Existing work in the PL area (including P4BID [192]) substantially ignore data dependencies and lead to overapproximations unsuitable for SDN applications. For example, the sensitivity of a field in a packet might depend on the packet's destination.

We develop a new approach to analyze information flow in P4 programs. A key idea is to augment a security type system (which is a language-based approach to check how information can flow in a program) with interval analysis, which in the domain of SDNs can be used to abstract over the network's parameters such as subnetworks segments, range of ports, and non-expired TTLs. Therefore, in our approach, in addition to a security label, the security type also keeps track of an interval.

Our analysis begins with an input policy, expressed as an assignment of types to fields of the input packet. For instance, a packet might be considered sensitive only if its source IP belongs to the internal network. The analysis conservatively propagates labels and intervals throughout the P4 program in manner reminiscent of dynamic information flow control [47] and symbolic execution, cf. [90]. This process is not dependent on a prior assignment of security labels to internal program variables, thus eliminating the need for the network engineer to deal with P4 program internals. The proposed analysis produces multiple final output packet typings, corresponding to different execution paths. These types are statically compared with the output security policy, which allows to relate observability of the output to intervals of fields of the resulting packets and their metadata.

The integration of security types and intervals is challenging. On one hand, the analysis should be path-sensitive and be driven by values in the packet fields to avoid rejecting secure programs due to overapproximation. On the other hand the analysis must be sound and not miss indirect information flows. Another challenge is that the behaviors of P4 programs depend on tables and external functions, but these components are not defined in P4. We address this by using user-defined contracts that overapproximate their behavior.

Summary of contributions.

- We propose a security type system which combines security labels and abstract domains to provide noninterference guarantees on P4 programs.
- Our approach allows defining data-dependent policies without the burden of annotating P4 programs.
- We implement the proposed type system in prototype tool TAP4S[1] and evaluate it on a test suite and 5 use cases.

[1] https://github.com/amir-ahmadian/TAP4S

E.2 P4 Language and Security Challenges

This section provides a brief introduction to the P4 language and its key features, while motivating the need for novel security analysis that strikes a balance between expressiveness of security policies and automation of the verification process.

P4 manipulates and forwards packets via a pipeline consisting of three stages: parser, match-action, and deparser. The parser stage dissects incoming packets, converting the byte stream into structured header formats. In the match-action stage, these headers are matched against rules to determine the appropriate actions, such as modifying, dropping, or forwarding the packet to specific ports. Finally, the deparser stage reconstructs the processed packet back into a byte stream, ready for transmission over the network.

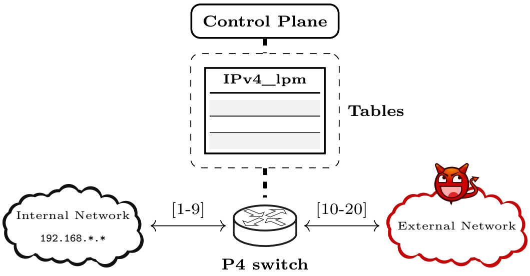


Figure E.1: *Congestion notifier* network layout

Program E.1 implements a switch that manages congestion in the network of Figure E.1. We assume this switch is the *only* ingress and egress point for traffic entering and exiting the internal network, connecting it to external networks. We use this program as a running example throughout the paper. In an IPv4 packet, the explicit congestion notification (ECN) field provides the status of congestion experienced during the packet's transmission. ECN value 0 indicates that neither the sender nor the receiver supports the ECN capability, 1 and 2 indicate that the sender supports ECN and the packet can be marked if congestion occurs, and 3 indicates that the packet has experienced congestion.

```
const bit<19> THRESHOLD = 10;

struct headers {
    ethernet_t  eth;
    ipv4_t      ipv4;
}

void decrease (inout bit<8> x) {
    x = x - 1;
}

```

```
parser MyParser(packet_in packet, out headers hdr,
                inout metadata meta,
                inout standard_metadata_t
                      standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.eth);
        transition select(hdr.eth.etherType) {
            0x0800: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
control MyCtrl(inout headers hdr,
               inout metadata meta,
               inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop(standard_metadata);
    }
    action ipv4_forward(bit<48> dstAddr, bit<9> port) {
        standard_metadata.egress_spec = port;
        hdr.eth.srcAddr = hdr.eth.dstAddr;
        hdr.eth.dstAddr = dstAddr;
        decrease(hdr.ipv4.ttl);
    }
    table ipv4_lpm {
        key = { hdr.ipv4.dstAddr: lpm; }
        actions = { ipv4_forward; drop; }
        default_action = drop();
    }
    apply {
        if (hdr.ipv4.isValid()) {
            if (hdr.ipv4.dstAddr[31:24] == 192 &&
              // BUG: hdr.ipv4.dstAddr[7:0] == 192 &&
                hdr.ipv4.dstAddr[23:16] == 168){
                if (standard_metadata.enq_qdepth >= THRESHOLD)
                    hdr.ipv4.ecn = 3;
            } else {
                hdr.ipv4.ecn = 0;
            }
            ipv4_lpm.apply(); //forward all valid packets
        } else {
            drop();
}   }   }
```

Program E.1: Congestion notifier

P4 structs and headers. Structs are records used to define the format of P4 packets. Headers are special structs with an additional implicit boolean indicating the header's validity, which is set when the header is extracted. Special function `isValid` (line 52) is used to check the validity of a header.

For example, the struct `headers` on line 3 has two headers of type `ethernet_t` and `ipv4_t`, as depicted in Figure E.2. The fields of the `ethernet_t` specify the source and destination MAC addresses and the Ethernet type. The header `ipv4_t` represents a standard IPv4 header with fields such as ECN, time-to-live (TTL), source and destination IP addresses.

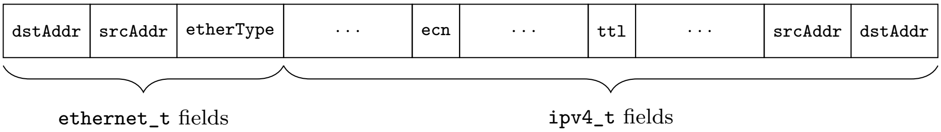


Figure E.2: Packet header

Parser. The parser dissects incoming raw packets (`packet` on line 12), extracts the raw bits, and groups them into headers. The parser's execution begins with the `start` state and terminates either in `reject` state or `accept` state accepting the packet and moving to the next stage of the pipeline.

For example, `MyParser` consists of three states. The parsing begins at the `start` state (line 17) and transitions to `parse_ethernet` extracting the Ethernet header from the input packet (line 22), which automatically sets the header's validity boolean to true. Next, depending on the value of `hdr.eth.etherType`, which indicates the packet's protocol, the parser transitions to either state `parse_ipv4` or state `accept`. If the value is 0x0800, indicating an IPv4 packet, the parser transitions to state `parse_ipv4` and extracts the IPv4 header (line 30). Finally, it transitions to the state `accept` (line 31), accepts the packet, and moves to the match-action stage.

Match-Action. This stage processes packets as instructed by control-plane-configured tables. A table consists of key-action rows and each row determines the action to be performed based on the key value. Key-action rows are updated by the control-plane, externally to P4. By applying a table, the P4 program matches the key value against table entries and executes the corresponding action. An action is a programmable function performing operations on a packet, such as forwarding, modifying headers, or dropping the packet.

The match-action block `MyCtrl` of Program E.1 starts at line 34. If the IPv4 header is not valid (line 52) the packet is dropped. Otherwise, if the packet's destination (line 53-55) is the internal network, the program checks for congestion. The standard metadata's `enq_qdepth` field indicates the length of the queue that stores packets waiting to be processed. A predefined `THRESHOLD` is used to determine the congestion status and store it in the `ecn` field (line 57 and 59). Finally, the packet is forwarded by applying the `ipv4_lpm` table (line 61). This table, defined at line 46, matches based on longest prefix (*lpm*) of the IPv4 destination address (`hdr.ipv4.dstAddr`), and has two actions (shown on line 48): `ipv4_forward` which forwards the packet and `drop` which drops the packet. If no match exists, the default action on line 49 is invoked.

Calling conventions. The P4 is a heapless language, implementing a unique copy-in/copy-out calling convention that allows static allocation of resources. P4 function parameters are optionally annotated with a direction (`in`, `inout` or `out`). The direction indicates how arguments are handled during function invocation and termination, offering fine-grained control over data visibility and potential side effects.

For example, `inout` indicates that the invoked function can both read from and write to a local copy of the caller's argument. Once the function terminates, the caller receives the updated value of that argument. For instance, assume `hdr.ipv4.ttl` value is 10 in line 43. The invocation of `decrease` copies-in the value 10 to parameter `x`, and the assignment on line 9 modifies `x` to value 9. Upon termination, the function copies-out the value 9 back to the caller's parameter, changing the value of `hdr.ipv4.ttl` to 9 in line 44.

Externs. Externs are functionalities that are implemented outside the P4 program and their behavior is defined by the underlying hardware or software platform. Externs are typically used for operations that are either too complex or not directly expressible in P4's standard constructs. This includes operations like hashing, checksum computations, and cryptographic functions. Externs can directly affect the global architectural state that is external to the P4 state, but their effects to the P4 state are controlled by the copy-in/copy-out calling convention.

For example, the extern function `mark_to_drop` (line 38) signals to the forwarding pipeline that a packet should be discarded. Generally, the packet is sent to the port identified by the standard metadata's `egress_spec` field, and dropping a packet is achieved by setting this field to the drop port of the switch. The drop port's value depends on the target switch; we assume the value is 0.

Problem statement

The power and flexibility of P4 to programmatically process and forward packets across different networks provides opportunities for security vulnerabilities such as information leakage and covert channels. For instance, in Program E.1, the standard metadata's `enq_qdepth` field, which indicates the length of the queue that stores packets waiting to be processed, indirectly reveals the congestion status of the current switch. Similarly, the `ecn` field of a packet, carrying explicit congestion information, can also reveal the congestion status of the network.

Given that the congestion information of an internal network and the switch are considered sensitive, packets traveling to the external networks should not carry this information. Therefore, Program E.1 sets the `ecn` field to 0 if the packet's destination IP address belongs to the external network (line 59), otherwise the congestion status is changed according to the switch's congestion. The application of the `ipv4_lpm` table (line 61) forwards the packet to a table-specified port. If the forwarding ports are not configured properly by the table or a bug (line 54) sets the `ecn` field on the packets leaving the internal network, the packets forwarded to an external network will leak information about the internal network's congestion state. Covert channels can also result from buggy or malicious programs. For example, by encoding the `ecn` field into the `ttl` field, an adversary can simply inspect `ttl` to deduce the congestion status.

To detect these vulnerabilities, we set out to study the security of P4 programs by means of information flow control (IFC). IFC tracks the flow of information within a program, preventing leakage from sensitive sources to public sinks. Information flow security policies are typically expressed by assigning security labels to the sources and sinks and the flow relations between security labels describe the allowed (and disallowed) information flows. In our setting, the sensitivity of sources (sinks) depends on predicates on the input (output) packets and standard metadata. Therefore, we specify the security labels of sources (i.e. input packet and switch state) by an *input policy*, while the security labels of the sinks (i.e. output packet and switch state) are specified by an *output policy*.

The input policy of Program E.1, describing the security label of its sources is defined as:

> *If the switch's input packet has the protocol IPv4 (i.e.* `hdr.eth.etherType` *is* 0x0800*) and its IPv4 source address* `hdr.ipv4.srcAddr` *belongs to the internal network subnet* `192.168.*.*`*, then the* `ecn` *field is secret, otherwise it is public. All the other fields of the input packet are always public, while the switch's* `enq_qdepth` *is always secret.* (1)

Program E.1 should not leak sensitive information to external networks. An output policy defines public sinks by the ports associated with the external network and labels the fields of the corresponding packets as public.

> *Packets leaving the switch through ports* 10 *to* 20 *are forwarded to the external network and are observable by attackers. Therefore, all fields of such packets should be public. All the other packets are not observable by attackers.* (2)

Our goal is to identify a static security analysis that strikes a balance between expressiveness and automation of the verification process. We identify three main challenges that a security analysis of P4 programs should address:

1. Security policies are data-dependent. For instance, the `ecn` field is sensitive only if the packet is IPv4 and its IP source address is in the range `192.168.*.*`.

2. The analysis should be value- and path-sensitive, reflecting the different values of header fields. For example, the value of the field `etherType` determines the packet's protocol and its shape. This information influences the reachability of program paths; for instance if the packet is IPv4 the program will not go through the parser states dedicated to processing IPv6 packets.

3. Externs and tables behavior are not defined in P4. Tables are statically-unknown components and configured at runtime. For example, a misconfiguration of the `ipv4_lpm` table may insecurely forward packets with sensitive fields to an external network.

Note that P4 lacks many features that could negatively affect analysis precision, including heap, memory aliasing, recursion, and loops.

Threat model. Our threat model considers a network attacker that knows the code of the P4 program and observes data on public sinks, as specified by a policy. We also assume that the keys and the actions of the tables are public and observable, but tables can pass secret data as the arguments of the actions. Because of the batch-job execution model, security policies can be specified as data-dependent security types over the initial and final program states. We aim at protecting against storage channels pertaining to explicit and implicit flows, while deferring other side channels, e.g. timing, to future work.

E.3 Solution Overview

We develop a novel combination of security type systems and interval abstractions to check information flow policies. We argue that our lightweight analysis of P4 programs provides a sweet spot balancing expressiveness, precision and automation.

Data-dependent policies are expressed by security types augmented with intervals, and the typing rules ensure that the program has no information flows from secret (H) sources to public (L) sinks. Specifically, a security type is a pair (I, ℓ) of an interval I indicating a range of possible values and a security label $\ell \in \{L, H\}$. For simplicity, we use the standard two-element security lattice $\{L, H\}$ ordered by $\sqsubseteq$ with lub $\sqcup$. For example, the type $(\langle 1, 5 \rangle, L)$ of the `ttl` field of the `ipv4` header specifies that the `ttl` field contains public data ranging between 1 and 5.

The security types allow to precisely express data-dependent policies such as (1). The input and output policies in our approach specify the shape of the input and output packets. Since packets can have many different shapes (e.g. IPv4 or IPv6), these policies may result in multiple distinct policy cases. For example, input policy (1) results in two cases:

In the *first input policy case*, the packet's `hdr.eth.etherType` is 0x0800, its IPv4 source address is in the internal network of interval $\langle\texttt{192.168.0.0}, \texttt{192.168.255.255}\rangle$, `hdr.ipv4.ecn` and standard metadata's `enq_qdepth` can contain any value (represented as $\langle * \rangle$) but are classified as H, while all other header fields are $(\langle * \rangle, L)$ (omitted here). We express this policy using our security types as follows:

$$\begin{aligned}
\texttt{hdr.eth.etherType} &: (\langle \text{0x0800}, \text{0x0800}\rangle, L)\\
\texttt{hdr.ipv4.srcAddr} &: (\langle \texttt{192.168.0.0},\ \texttt{192.168.255.255}\rangle, L)\\
\texttt{hdr.ipv4.ecn} &: (\langle * \rangle, H)\\
\texttt{standard\_metadata.enq\_qdepth} &: (\langle * \rangle, H)
\end{aligned}$$

The intervals and labels in these security types describe the values and labels of the initial state of the program under this specific input policy case.

The *second input policy case* describes all the packets where `hdr.eth.etherType` is not 0x0800 or IPv4 source address is outside of the range $\langle\texttt{192.168.0.0}, \texttt{192.168.255.255}\rangle$, all of the packet header fields are $(\langle * \rangle, L)$, while the standard metadata's `enq_qdepth` is still $(\langle * \rangle, H)$.

Similarly, the output policy (2) can be expressed with the output policy case: "if the standard metadata's `egress_spec` is $(\langle 10, 20\rangle, L)$, then all of the packet's header fields are $(\langle * \rangle, L)$."

It turns out that this specific output policy case is the only interesting one, even though output policy (2) can result in two distinct policy cases. In the alternative case, the fact that the attacker is unable to observe the output packet can be represented by assigning $(\langle * \rangle, H)$ to all of the fields of the packet. The flow relation among security labels, as determined by the ordering of the security labels, only characterizes flows from H sources to L sinks as insecure. This implies that any policy cases where the source is L or the sink is H cannot result in insecure flows. Thus, the alternative case is irrelevant and can be safely ignored.

Driven by the data-dependent types, we develop a new security type system that uses the intervals to provide a finer-grained assignment of security labels. Our interval analysis allows the type system to statically eliminate execution paths that are irrelevant to the security policy under consideration, thus addressing the second challenge of precise analysis. For example, our interval analysis can distinguish between states where `hdr.eth.etherType` is 0x0800 and the states where it is not, essentially providing a path-sensitive analysis. This enables the analysis to avoid paths where `hdr.eth.etherType` is *not* 0x0800 when checking the policy of IPv4 packets. As a result, we exclude paths visited by non-IPv4 packets when applying the `ipv4_lpm` table in line 61. This reduces the complexity of the analysis as we avoid exploring irrelevant program paths, and helps reduce false positives in the results.

Finally, to address the challenge of tables and externs, we rely on user-defined contracts which capture a bounded model of the component's behavior. Upon analyzing these components, the type system uses the contracts to drive the analysis. For Program E.1, the contract for a correctly-configured table `ipv4_lpm` ensures that if the packet's `hdr.ipv4.dstAddr` belongs to the internal network, then the action `ipv4_forward` (line 40) forwards the packet to ports and MAC addresses connected to the internal network.

Even if the `ipv4_lpm` table is correctly configured and its contract reflects that, bugs in the program can still cause unintended information leakage. For example, on line 54, the if condition might have been incorrect and instead of checking the 8 most significant bits (i.e. [31,24]) of the `hdr.ipv4.dstAddr`, it checks the least significant bits (i.e. [7,0]). This bug causes the `hdr.ipv4.ecn` field in some packets destined for the external network to include congestion information, leading to unintended information leaks. Such errors are often overlooked but can be detected by our type system.

In the end, to ensure that the program does not leak sensitive information, the final types produced by the type system are checked against an output policy. If this check succeeds, the program is deemed secure. The details of this process and the role of the interval information in the verification process are explained in Section E.6.

E.4 Semantics

In this section, we briefly summarize a big-step semantics of P4. The language's program statements, denoted by s, include standard constructs such as assignments, conditionals, and sequential composition. Additionally, P4 supports transition statements, function calls, table invocations, and extern invocations as shown in Figure E.3.

Values, represented by v, are either big-endian bitvectors $\overline{b}$ (raw packets) or structs $\{f_1 = v_1, ..., f_n = v_n\}$ (representing headers).

P4 states m are mappings from variables x to values v. In this slightly simplified semantics, variables are either global or local. States can thus be represented as disjoint unions (m_g, m_l), where m_g (m_l) maps global (local) variables only.

While externs in P4 can modify the architectural state, they cannot change the P4 state itself. To simplify our model, we integrate the architectural state into

$$
\begin{aligned}
v &::= \overline{b} \mid \{f_1 = v_1, ..., f_n = v_n\} \\
e &::= v \mid x \mid \ominus e \mid e \oplus e' \mid e.f \mid e[\overline{b} : \mathit{bitv}'] \mid \{f_1 = e_1, ..., f_n = e_n\} \\
\mathit{lval} &::= x \mid \mathit{lval}.f \mid \mathit{lval}[\overline{b} : \overline{b}'] \\
s &::= \texttt{skip} \mid \mathit{lval} := e \mid s_1; s_2 \mid \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{apply}\ \mathit{tbl} \mid \\
&\qquad f(e_1, ..., e_n) \mid \texttt{transition select}\, e\, \{v_1 : st_1, ..., v_n : st_n\}\, st
\end{aligned}
$$

Figure E.3: Syntax

P4's global state, treating it as a part of the global state. Therefore, in our model the externs are allowed to modify the global state of P4. To maintain isolation between the program's global variables and the architectural state, we assume that the variable names used to represent the global state are distinct from those used for the architectural state.

Expressions e use a standard selection of operators including binary $\oplus$, unary $\ominus$, comparison $\otimes$, and struct field access, as well as bitvector slicing $e[b : a]$ extracting the slice from index a to index b of e, and $m(e)$ is the evaluation of e in state m. An lvalue *lval* is an assignable expression, either a variable, a field of a struct, or a bitvector slice. The semantics of expressions is standard and consists of operations over bitvectors and record access.

The semantics of statements uses a mapping E from function names f to pairs $(s, \overline{(x, d)})$, where $\overline{(x, d)}$ is the signature of f, a sequence of pairs (x_i, d_i) of function parameters with their directions $d_i \in \{\texttt{in}, \texttt{out}, \texttt{inout}\}$. Additionally, E maps parser state names *st* to their bodies. Furthermore, since P4 programs may depend on external components, E also maps externs f and tables t to their respective implementations.

The semantic rules presented in Figure E.4 rely on judgments of the form $E : m_1 \xrightarrow{s} m_2$ to represent the execution of statement s under mapping E which starts from state m_1 and terminates in m_2.

Many of the rules in Figure E.4 are standard and are therefore not explained here. Rule S-Call fetches the invoked function's body s and signature, and copies in the arguments into m_l', which serves as the local state for the called function and is used to execute the function's body. Note that the function's body can modify the global state, but cannot change the caller's local state due to P4's calling conventions. After executing the function's body, the variables in final local state m_l'' must be copied out according to the directions specified in the function's signature. Given a direction d_i, the auxiliary function isOut returns true if the direction is `out` or `inout`. We rely on this function to copy-out the values from m_l'' back to the callee only for parameters with `out` and `inout` direction.

S-Skip

$$\frac{}{E : m \xrightarrow{\texttt{skip}} m}$$

S-Assign

$$\frac{m' = m[lval \mapsto m(e)]}{E : m \xrightarrow{lval := e} m'}$$

S-Seq

$$\frac{\begin{array}{c} E : m \xrightarrow{s_1} m' \\ E : m' \xrightarrow{s_2} m'' \end{array}}{E : m \xrightarrow{s_1 ; s_2} m''}$$

S-Cond-T

$$\frac{\begin{array}{c} m(e) = \text{true} \\ E : m \xrightarrow{s_1} m' \end{array}}{E : m \xrightarrow{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} m'}$$

S-Cond-F

$$\frac{\begin{array}{c} m(e) = \text{false} \\ E : m \xrightarrow{s_2} m' \end{array}}{E : m \xrightarrow{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} m'}$$

S-Call

$$\frac{\begin{array}{c} (s, \overline{(x, d)}) = E(f) \qquad m_l' = \{x_i \mapsto (m_g, m_l)(e_i)\} \\ E : (m_g, m_l') \xrightarrow{s} (m_g', m_l'') \end{array}}{E : (m_g, m_l) \xrightarrow{f(e_1, \ldots, e_n)} (m_g', m_l)[e_i \mapsto m_l''(x_i) \mid \text{isOut}(d_i)]}$$

S-Extern

$$\frac{\begin{array}{c} (sem_f, \overline{(x, d)}) = E(f) \qquad m_l' = \{x_i \mapsto (m_g, m_l)(e_i)\} \\ (m_g', m_l'') = sem_f(m_g, m_l') \end{array}}{E : m \xrightarrow{f(e_1, \ldots, e_n)} (m_g', m_l)[e_i \mapsto m_l''(x_i) \mid \text{isOut}(d_i)]}$$

S-Trans

$$\frac{st' = \begin{cases} st_i & \text{if } m(e) = v_i \\ st & \text{otherwise} \end{cases} \qquad E : m \xrightarrow{E(st')} m'}{E : m \xrightarrow{\texttt{transition select}\, e\, \{v_1 : st_1, \ldots, v_n : st_n\}\ st} m'}$$

S-Table

$$\frac{\begin{array}{c} (\overline{e}, sem_{tbl}) = E(tbl) \qquad sem_{tbl}((m_g, m_l)(e_1), ..., (m_g, m_l)(e_n)) = (a, \overline{v}) \\ (s, (x_1, \text{none}), ..., (x_n, \text{none})) = E(a) \qquad m_l' = \{x_i \mapsto v_i\} \qquad E : (m_g, m_l') \xrightarrow{s} (m_g', m_l'') \end{array}}{E : (m_g, m_l) \xrightarrow{\texttt{apply } tbl} (m_g', m_l)}$$

Figure E.4: Semantic rules

For example, in Program E.1 let $m_l = \{\texttt{hdr.ipv4.ttl} \mapsto 2\}$ when invoking `decrease` at line 44. The local state of `decrease` (i.e. the copied-in state) becomes $m_l' = \{\texttt{x} \mapsto 2\}$. After executing the function's body (line 8), the final local state will be $m_l'' = \{\texttt{x} \mapsto 1\}$ while the global state m_g remains unchanged. Finally, the copying out operation updates the caller's state to $m'' = (m_g, \{\texttt{ttl} \mapsto 1\})$ by updating its local state.

The S-Extern rule is similar to S-Call. The key difference is that instead of keeping a body in E, we keep the extern's behavior defined through sem_f. This function takes a state containing the global m_g and copied-in state m_l' and returns (possibly) modified global and local states, represented as $sem_f(m_g, m'_l) = (m'_g, m''_l)$. Finally, the extern rule preforms a copy-out procedure similar to the function call.

The S-TRANS rule defines how the program transitions between parser states based on the evaluation of expression e. It includes a default state name st for unmatched cases. If in program state m, expression e evaluates to value v_i, the program transitions to state name st_i according to the defined value-state pattern. However, if the evaluation result does not match any of the v_i values, the program instead transitions to the default state st. For example, assume that $m($`hdr.eth.etherType`$) = 0\text{x}0800$ on line 23 of Program E.1. The select expression within the transition statement will transition to the state `parse_ipv4`, and executes its body.

Rule S-TABLE fetches from E the table's implementation sem_{tbl} and a list of expressions $\overline{e}$ representing table's keys. It then proceeds to evaluate each of these expressions in the current state (m_g, m_l), passing the evaluated values as key values to sem_{tbl}. The table's implementation sem_{tbl} then returns an action a and its arguments $\overline{v}$. We rely on E again to fetch the body and signature of action a, however, since in P4 action parameters are directionless we use `none` in the signature to indicate there is no direction. Finally, similar to S-CALL we copy-in the arguments into m'_l, which serves as the local state for the invoked action and is used to execute the action's body. For example, let $m($`hdr.ipv4.dstAddr`$) = 192.168.2.2$ at line 61, and the semantics of table `ipv4_lpm` contains:

$$192.168.2.2 \mapsto \texttt{ipv4\_forward (4A:5B:6C:7D:8E:9F, 5)}$$

then the table invokes action `ipv4_forward` with arguments `4A:5B:6C:7D:8E:9F` and `5`.

E.5 Types and Security Condition

In our approach types are used to represent and track both bitvector abstractions (i.e. intervals) and security labels, and we use the same types to represent input and output policies.

In P4, bitvector values represent packet fragments, where parsing a bitvectors involves slicing it into sub-bitvectors (i.e. slices), each with different semantics such as payload data or header fields like IP addresses and ports. These header fields are typically evaluated against various subnetwork segments or port ranges. Since header fields or their slices are still bitvectors, they can be conveniently represented as integers, enabling us to express the range of their possible values as $I = \langle a, b \rangle$, the interval of integers between a and b.

We say a bitvector v is typed by type τ, denoted as $v : \tau$, if τ induces a slicing of v that associates each slice with a suitable interval I and security level $\ell \in \{L, H\}$. We use the shorthand I^{ℓ}_{i} to represent a slice of length i, with interval $I \subseteq \langle 0, 2^i - 1 \rangle$ and security label ℓ. The bitvector type can therefore be presented as $\tau = {I_n}_{i_n}^{\ell_n} \cdots {I_1}_{i_1}^{\ell_1}$, representing a bitvector of length $\Sigma_{j=1}^{n} i_j$ with n slices, where each slice i has interval I_i and security label ℓ_i. Singleton intervals are abbreviated $\langle a \rangle$, $\langle \rangle$ is the empty interval, and $\langle * \rangle$ is the complete interval, that is, the range $\langle 0, 2^i - 1 \rangle$ for a slice of length i. Function $\mathrm{lbl}(\tau)$ indicates the least upper bound of the labels of slices in τ.

To illustrate this, let τ_1 be $\langle * \rangle_2^{H} \cdot \langle 0, 1 \rangle_3^{L}$ which types a bitvector of length 5 consisting of two slices. The first slice has a length of 3, with values drawn from the interval $\langle 0, 1 \rangle$ and security label L. The second slice, with a length of 2, has a security label H, and its values drawn from the complete interval $\langle 0, 3 \rangle$ (indicated by $*$). Accordingly, $\mathrm{lbl}(\tau_1)$ evaluates to $H \sqcup L = H$.

Type τ is also used to denote a record type, where record $\{f_1 = v_1, ..., f_n = v_n\}$ is typed as $\langle f_1 : \tau_1; ...; f_n : \tau_\mathrm{n} \rangle$ if each value v_i is typed with type τ_i.

In this setting, the types are not unique, as it is evident from the fact that a bitvector can be sliced in many ways and a single value can be represented by various intervals. For example, bitvector $\boxed{1}\boxed{0}\boxed{0}$ can be typed as $\langle 4 \rangle_3^{L}$, or $\langle * \rangle_1^{L} \cdot \langle 0, 1 \rangle_2^{L}$, or $\langle 2 \rangle_2^{L} \cdot \langle * \rangle_1^{L}$.

State types. A type environment, or *state type*, $\gamma = (\gamma_g, \gamma_l)$ is a pair of partial functions from variable names x to types τ. Here, γ_g and γ_l represent global and local state types, respectively, analogous to the global (m_g) and local (m_l) states in the semantics. We say that γ can type state m, $\gamma \vdash m$, if $\forall x \in m,\ m(x) : \gamma(x)$. A state type might include a type with an empty interval; we call this state type *empty* and denote it as $\bullet$.

Let $\mathrm{lblOf}(\mathit{lval}, \gamma)$ be the security label of *lval* in state type γ. The states m_1 and m_2 are considered *low equivalent with respect to* γ, denoted as $m_1 \underset{\gamma}{\sim} m_2$, if for all *lval* such that $\mathrm{lblOf}(\mathit{lval}, \gamma) = L$, then $m_1(\mathit{lval}) = m_2(\mathit{lval})$ holds.

Example E.1

Assume a state type $\gamma = \{x \mapsto \langle * \rangle_1^{H} \cdot \langle 0, 1 \rangle_2^{L}\}$ The following states $m_1 = \{x \mapsto \boxed{0}\boxed{0}\boxed{0}\}$ and $m_2 = \{x \mapsto \boxed{1}\boxed{0}\boxed{0}\}$ are low equivalent wrt. γ. However, states $m_1 = \{x \mapsto \boxed{0}\boxed{0}\boxed{0}\}$ and $m_3 = \{x \mapsto \boxed{1}\boxed{0}\boxed{1}\}$ are not low equivalent even though both can be typed by γ.

Contracts. A table consists of key-action rows, and in our threat model, we assume the keys and actions of the tables are always public (i.e. L), but the arguments of the actions *can* be secret (i.e. H). Given that tables are populated by the control-plane, the behavior of a table is unknown at the time of typing. We rely on user-specified contracts to capture a bounded model of the behavior of the tables. In our model, a table's contract has the form $(\overline{e}, \mathrm{Cont}_{tbl})$, where $\overline{e}$ is a list of expressions indicating the keys of the table, and Cont_{tbl} is a set of tuples $(\phi, (a, \overline{\tau}))$, where ϕ is a boolean expression defined on $\overline{e}$, and a denotes an action to be invoked with argument types $\overline{\tau}$ when ϕ is satisfied.

For instance, the `ipv4_lpm` table of Program E.1 uses `hdr.ipv4.dstAddr` as its key, and can invoke two possible actions: `drop` and `ipv4_forward`. An example of a contract for this table is depicted in Figure E.5. This contract models a table that forwards the packets with $\texttt{hdr.ipv4.dstAddr} = \texttt{192.*.*.*}$ to ports 1-9, the ones with $\texttt{hdr.ipv4.dstAddr} = \texttt{10.*.*.*}$ to ports 10-20, and `drops` all the other packets. Notice that in the first case, the first argument resulting from the table look up is secret.

$$\begin{aligned}
&\Big(\ [\texttt{hdr.ipv4.dstAddr}],\\
&\Big\{\Big(\texttt{dstAddr}[31:24] = 192, (\texttt{ipv4\_forward}, [\langle * \rangle_{48}^{H}, \langle 1, 9\rangle_{9}^{L}])\Big)\\
&\quad\Big(\texttt{dstAddr}[31:24] = 10, (\texttt{ipv4\_forward}, [\langle * \rangle_{48}^{L}, \langle 10, 20\rangle_{9}^{L}])\Big)\\
&\quad\Big(\texttt{dstAddr}[31:24] \neq 192 \wedge \texttt{dstAddr}[31:24] \neq 10, (\texttt{drop},\ [])\Big)\Big\}\Big)
\end{aligned}$$

Figure E.5: The contract of `ipv4_lpm` table

The table contracts are essentially the security policies of the tables, where ϕ determines a subset of table rows that invoke the same action (a) with the same argument types ($\overline{\tau}$). Using the labels in $\overline{\tau}$, and given action arguments $\overline{v}_1$ and $\overline{v}_2$, we define $\overline{v}_1 \underset{\overline{\tau}}{\sim} \overline{v}_2$ as $|\overline{v}_1| = |\overline{v}_2| = |\overline{\tau}|$ and for all i, $v_{1_i} : \tau_i$ and $v_{2_i} : \tau_i$, and if $\text{lbl}(\tau_i) = L$ then $v_{1_i} = v_{2_i}$. Note that $\text{lbl}(\tau_i)$ returns the least upper bound of the labels of all τ_i's slices, hence if there is even one H slice in τ_i, $\text{lbl}(\tau_i)$ would be H. We use mapping T to associate table names *tbl* with their contracts.

We say that two mappings E_1 and E_2 are considered *indistinguishable* wrt. T, denoted as $E_1 \underset{T}{\sim} E_2$, if for all tables *tbl* such that $(\overline{e}_1, sem_{1_{tbl}}) = E_1(tbl)$, $(\overline{e}_2, sem_{2_{tbl}}) = E_2(tbl)$, $(\overline{e}, \text{Cont}_{tbl}) = T(tbl)$ then $\overline{e}_1 = \overline{e}_2 = \overline{e}$, and for all $(\phi, (a, \overline{\tau})) \in \text{Cont}_{tbl}$, and for all arbitrary states m_1 and m_2, such that $m_1(\overline{e}) = m_2(\overline{e}) = v$ and $m_1(\phi) \Leftrightarrow m_2(\phi)$, if $m_1(\phi)$ then exists $\overline{v}_1, \overline{v}_2$ such that $sem_{1_{tbl}}(\overline{v}) = (a, \overline{v}_1)$, $sem_{2_{tbl}}(\overline{v}) = (a, \overline{v}_2)$, and $\overline{v}_1 \underset{\overline{\tau}}{\sim} \overline{v}_2$. In other words, T-indistinguishability of E_1 and E_2 guarantees that given equal key values, E_1 and E_2 return the same actions with $\overline{\tau}$-indistinguishable arguments $\overline{v_1}$ and $\overline{v_2}$ such that these arguments are in bound wrt. their type $\overline{\tau}$.

Security condition. As overview in Section E.3 the input and output policy cases are expressed by assigning types to program variables. As such, state types, specifying security types of program variables are used to formally express input and output policy cases. Hereafter, we use γ_i and γ_o to denote input and output policy cases, respectively. Using this notation, the input policy, denoted by Γ_i, is represented as a set of input policy cases γ_i. Similarly, the output policy is expressed as a set of output policy cases γ_o and denoted by Γ_o.

Given this intuition, we say two states m_1 and m_2 are *indistinguishable* wrt. a policy case γ if $\gamma \vdash m_1$, $\gamma \vdash m_2$, and $m_1 \underset{\gamma}{\sim} m_2$. Relying on this, we present our definition of noninterference as follows:

Definition E.1 (Noninterference)

A program s is *noninterfering* wrt. the input and output policies cases γ_i and γ_o, table contracts in T, mappings E_1 and E_2, and initial states m_1 and m_2, if the following hold:

- $E_1 \underset{T}{\sim} E_2$,
- $\gamma_i \vdash m_1$, $\gamma_i \vdash m_2$, and $m_1 \underset{\gamma_i}{\sim} m_2$,
- $E_1 : m_1 \xrightarrow{s} m_1'$

then

- there exists a state m_2' such that $E_2 : m_2 \xrightarrow{s} m_2'$,
- if $\gamma_o \vdash m_1'$, then $\gamma_o \vdash m_2'$ and $m_1' \underset{\gamma_o}{\sim} m_2'$.

Note that the noninterference definition holds trivially if $\gamma_o \nvdash m_1'$. This is because at least one of the intervals in γ_o does not capture the values in m_1', indicating that policy case γ_o does not apply to this specific state.

The existential quantifier over the state m_2' does not mean that the language is non-deterministic, in fact if such state exists it is going to be unique. This existential quantifier guarantees that our security condition is termination sensitive, meaning that it only applies to cases where the program terminates for both initial states m_1 and m_2.

Example E.2

Assume program `if y==1 then x=1 else x=x+1`, input policy case $\gamma_i = \{x \mapsto \langle * \rangle_2^{H}, y \mapsto \langle 1 \rangle_3^{L}\}$, and initial states $m_1 = \{x \mapsto \boxed{1}\boxed{0}, y \mapsto \boxed{0}\boxed{0}\boxed{1}\ \}$ and $m_2 = \{x \mapsto \boxed{0}\boxed{1}, y \mapsto \boxed{0}\boxed{0}\boxed{1}\ \}$. We can see that $\gamma_i \vdash m_1$, $\gamma_i \vdash m_2$, and $m_1 \underset{\gamma_i}{\sim} m_2$. In a scenario where the only initial states are m_1 and m_2, executing this program would result in final states $m_1' = \{x \mapsto \boxed{0}\boxed{1}, y \mapsto \boxed{0}\boxed{0}\boxed{1}\ \}$ and $m_2' = \{x \mapsto \boxed{0}\boxed{1}, y \mapsto \boxed{0}\boxed{0}\boxed{1}\ \}$, respectively. Given output policy case $\gamma_o = [x \mapsto \langle * \rangle_2^{L}, y \mapsto \langle 1 \rangle_3^{L}]$, we say that this program is noninterfering wrt. γ_o because $\gamma_o \vdash m_1'$, $\gamma_o \vdash m_2'$, and $m_1' \underset{\gamma_o}{\sim} m_2'$.

We extend the definition of noninterference to input policies Γ_i and output policies Γ_o, requiring the program to be noninterfering for *every pair* of input and output policy cases. In our setting, the output policy, which indicates the shape of the output packets, describes what the attacker observes. As such, it is typically independent of the shape of the input packet and the associated input policy. Thus, our approach does not directly pair input and output policy cases. Instead, it ensures that the program is noninterfering for all combinations of input and output policy cases.

E.6 Security Type System

We introduce a security type system that combines security types and interval abstractions. Our approach begins with an input policy case and conservatively propagates labels and intervals of P4 variables. In the following, we assume that the P4 program is well-typed.

Typing of Expressions

The typing judgment for expressions is $\gamma \vdash e : \tau$. Rules for values, variables, and records are standard and omitted here.

P4 programs use bitvectors to represent either raw packets (e.g. `packet_in packet` of line 12) or finite integers (e.g. `x` of line 8). While there is no distinction between these two cases at the language level, it is not meaningful to add or multiply two packets, as it is not extracting a specific byte from an integer representing a time-to-live value. For this reason, we expect that variables used to marshal records have multiple slices but are not used in arithmetic operations, while variables used for integers have one single slice and are not used for sub-bitvector operations. This allows us to provide a relatively simple semantics of the slice domain, which is sufficient for many P4 applications.

$$\textsc{T-SingleSliceBs}\quad \frac{\gamma \vdash e_1 : \langle I_1 \rangle_i^{\ell_1} \qquad \gamma \vdash e_2 : \langle I_2 \rangle_i^{\ell_2}}{\begin{array}{c}\gamma \vdash e_1 \oplus e_2 : \langle I_1 \oplus I_2 \rangle_i^{\ell_1 \sqcup \ell_2} \\ \gamma \vdash \ominus e_1 : \langle \ominus I_1 \rangle_i^{\ell_1} \\ \gamma \vdash e_1 \otimes e_2 : \langle I_1 \otimes I_2 \rangle_i^{\ell_1 \sqcup \ell_2}\end{array}}$$

T-SingleSliceBs rule allows the reuse of standard interval analysis for binary, unary, and comparison operations over bitvectors that have only *one* single slice. The resulting label is the least upper bound of labels associated with the input types.

In the slicing rules, sub-bitvector (i.e. $e[b : a]$) preserves precision only if the slices of the input are aligned with sub-bitvector's indexes, otherwise sub-bitvector results in $\langle * \rangle$, representing all possible values. The following lemmas show that interval and labeling analysis of expressions is sound:

T-AlignedSlice

$$\frac{\gamma \vdash e : \langle I_n \rangle_{i_n}^{\ell_n} \cdots \langle I_1 \rangle_{i_1}^{\ell_1}}{\gamma \vdash e[\sum_{j=1}^{b} i_j : \sum_{j=1}^{a} i_j] : \langle I_b \rangle_{i_b}^{\ell_b} \cdots \langle I_a \rangle_{i_a}^{\ell_a}}$$

T-NonAlignedSlice

$$\frac{\gamma \vdash e : \langle I_n \rangle_{i_n}^{\ell_n} \cdots \langle I_1 \rangle_{i_1}^{\ell_1}}{\gamma \vdash e[b : a] : \langle * \rangle_{b-a}^{\bigsqcup \ell_{i_j}}}$$

Lemma E.1

Given expression e, state m, and state type γ such that $\gamma \vdash m$, if the expression is well-typed $\gamma \vdash e : \tau$, and evaluates to a value $m(e) = v$, then:

- *v is well-typed wrt. to the interval of type τ (i.e. $v : \tau$).*
- *for every state m' such that $m \underset{\gamma}{\sim} m'$, if $\mathrm{lbl}(\tau) = L$, then $m'(e) = v$.*

Typing of statements

To present the typing rules for statements, we rely on some auxiliary notations and operations to manipulate state types, which are introduced informally here due to space constraints. The properties guaranteed by these operations are reported in Appendix E.4. $\gamma[\mathit{lval} \mapsto \tau]$ indicates updating the type of *lval*, which can be a part of a variable, in state type γ. $\gamma {+\!\!+} \gamma'$ updates γ such that for every variable in the domain of γ', the type of that variable in γ is updated to match γ'. $\mathrm{refine}(\gamma, e)$ returns an overapproximation of γ that satisfy the abstraction of γ and the predicate e. $\mathrm{join}(\gamma_1, \gamma_2)$ returns an overapproximation of γ_1, whose labels are at least as restrictive as γ_1 *and* γ_2. These operations tend to overapproximate, potentially causing a loss of precision in either the interval or the security label, as illustrated in the following example:

Example E.3

Let x be mapped to an interval between 2 and 8, or in binary, bitvectors between $\begin{array}{|c|c|c|c|}\hline 0&0&1&0\\\hline\end{array}$ and $\begin{array}{|c|c|c|c|}\hline 1&0&0&0\\\hline\end{array}$, in γ. That is, $\gamma = \{x \mapsto \langle 2, 8 \rangle_4^L\}$. The following update $\gamma[x[3:3] \mapsto \langle 0 \rangle_1^H]$ modifies the slice $x[3:3]$ and results in the state type $\{x \mapsto \langle 0 \rangle_1^H \cdot \langle * \rangle_3^L\}$. Here, lvalue $x[2:0]$ loses precision because after updating $x[3:3]$, the binary representation of the interval of lvalue $x[2:0]$ would be between $\begin{array}{|c|c|c|}\hline 0&1&0\\\hline\end{array}$ and $\begin{array}{|c|c|c|}\hline 0&0&0\\\hline\end{array}$, that is every 3-bit value except $\begin{array}{|c|c|c|}\hline 0&0&1\\\hline\end{array}$. Such value set cannot be represented by a single continuous interval, hence we overapproximate to the complete interval $\langle * \rangle$. Similarly, the operation $\mathrm{refine}(\gamma, x[3:3] < 1)$ updates the interval of lvalue $x[3:3]$ which results in $\{x \mapsto \langle 0 \rangle_1^L \cdot \langle * \rangle_3^L\}$ where lvalue $x[2:0]$ again loses precision. On the other hand, an operation such as $\mathrm{join}(\gamma, \{x \mapsto \langle * \rangle_1^H \cdot \langle * \rangle_3^L\})$ does

> not modify the intervals of γ, but since the $\langle * \rangle_1^{H}$ slice overlaps with a slice of x in γ its label should be raised, which results in $\gamma' = \{x \mapsto \langle 2, 8 \rangle_4^{H}\}$.

The security typing of statement s uses judgments of the form $T, pc, \gamma \vdash s : \Gamma$, where pc is the security label of the current program context, T is a static mapping, and γ is a state type. We use T to map a parser state name (st) or function name (f) to their bodies. For functions, T also returns their signatures. Moreover, as described in Section E.5, we also use T to map externs and tables to their contracts. The typing judgment concludes with Γ, which is a set of state types. In our type system, the security typing is not an on-the-fly check that immediately rejects a program when encountering an untypeable statement. Instead, we proceed with typing the program and produce a state type for each path and accumulate all of those in a final set Γ. This is done in order to increase precision, by minimizing the need to unify, and hence overapproximate, intermediate typings during type derivation. This is indeed one of the key technical innovations of our type system, as explained in more detail below. Once the final set Γ is obtained, the state types within Γ are then verified against the output security policy Γ_o, ensuring that they respect the requirements of all the output policy cases γ_o within it.

In the following rules, we use $\text{raise}(\tau, \ell)$ to return a type where each label ℓ' within τ has been updated to $\ell' \sqcup \ell$. We omit the typing rules for assignment and sequential composition due space constraints. The full list of our typing rules can be found in Figure E.8 in the Appendix.

$$\textsc{T-Cond}\quad \frac{\begin{array}{c}\gamma \vdash e : \tau \qquad \ell = \text{lbl}(\tau) \qquad pc' = pc \sqcup \ell \\ T, pc', (\text{refine}(\gamma, e)) \vdash s_1 : \Gamma_1 \qquad T, pc', (\text{refine}(\gamma, \neg e)) \vdash s_2 : \Gamma_2\end{array}}{T, pc, \gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \text{joinOnHigh}(\Gamma_1 \cup \Gamma_2, \ell)}$$

T-Cond This rule types the two branches using state types refined with the branch condition and its negation, which results in the state type sets Γ_1 and Γ_2, respectively. The final state type set is a simple union of Γ_1 and Γ_2.

However, in order to prevent implicit information leaks, if the branch condition is H, the security labels of Γ_1 and Γ_2 should be joined. We do this by the auxiliary function joinOnHigh, defined as follows:

$$\text{joinOnHigh}(\Gamma, \ell) = \begin{cases} \text{join}(\Gamma) & \text{if } \ell = H \\ \Gamma & \text{otherwise} \end{cases}$$

where the join operator has been lifted to Γ and defined as $\text{join}(\Gamma) = \{\text{join}(\gamma, \Gamma) \mid \gamma \in \Gamma\}$, $\text{join}(\gamma, \{\gamma'\} \cup \Gamma) = \text{join}(\text{join}(\gamma, \gamma'), \Gamma)$ and $\text{join}(\gamma, \varnothing) = \gamma$.

Example E.4

Consider the conditional statement on line 56 of Program E.1, where initially $\gamma = \{$`enq_qdepth` $\mapsto \langle * \rangle^{H}_{19},$ `hdr.ipv4.ecn` $\mapsto \langle * \rangle^{L}_{2}, \ldots\}$. Since the label of `enq_qdepth` is H, after the assignment on line 57, `hdr.ipv4.ecn` becomes H in Γ_1. However, since there is no `else` branch, s_2 is trivially `skip`, meaning that `hdr.ipv4.ecn` remains L in Γ_2. Typically, in IFC, the absence of an update for `hdr.ipv4.ecn` in the `else` branch leaks that the if statement's condition does not hold. To prevent this, we join the security labels of all state types if the branch condition is H. Therefore, in the final state set Γ', `hdr.ipv4.ecn` is labeled H.

Even on joining the security labels, T-Cond does not merge the final state types in order to maintain abstraction precision. To illustrate this consider program `if b then x[0:0]=0 else x[0:0]=1`, where the *pc* and the label of b are both L, and an initial state type $\gamma = \{x \mapsto \langle * \rangle^{H}_{3}; b \mapsto \langle * \rangle^{L}_{1}\}$. After typing both branches, the two typing state sets are $\Gamma_1 = \{\{x \mapsto \langle * \rangle^{H}_{2} \cdot \langle 0 \rangle^{L}_{1}\}\}$ and $\Gamma_2 = \{\{x \mapsto \langle * \rangle^{H}_{2} \cdot \langle 1 \rangle^{L}_{1}\}\}$. Performing a union after the conditional preserves the labeling and abstraction precision of `x[0:0]`, whereas merging them would result in a loss of precision.

The rule for typing parser transitions is similar to T-Cond, it individually types each state's body and then joins or unions the final state types based on the label of *pc*. This rule can be found in Figure E.8 in the Appendix.

T-EmptyType

$$\frac{}{T, L, \bullet \vdash s : \Gamma}$$

T-EmptyType Refining a state type might lead to an empty abstraction for some variables. We call these states empty and denote them by $\bullet$. An empty state indicates that there is no state m such that $\bullet \vdash m$. The rule states that from an empty state type, any statement can result in any final state type, since there is no concrete state that matches the initial state type. Notice that Γ can simply be *empty* and allow the analysis to prune unsatisfiable paths. This rule applies *only* when *pc* is L. For cases where *pc* is H, simply pruning the empty states is *unsound*, as illustrate by the following example:

Example E.5

Assume the state type $\gamma = \{$`enq_qdepth` $\mapsto \langle 5 \rangle^{H}_{19},$ `hdr.ipv4.ecn` $\mapsto \langle * \rangle^{L}_{2}, \ldots\}$, upon reaching the conditional statement on line 56 of Program E.1. The refinement of the `then` branch under the condition `enq_qdepth` $\geq$ `THRESHOLD` (where `THRESHOLD` is a constant value 10) results in the empty state $\bullet = \{$`enq_qdepth` $\mapsto \langle\rangle^{H}_{19}, \ldots\}$, where $\langle\rangle^{H}_{19}$ denotes an empty interval. If we prune this empty state type, the final state type set Γ' contains only the state types obtained from the `else` branch (which is `skip`). This is unsound because a L-observer would be able see that the value of `hdr.ipv4.ecn` has remained unchanged and infer that the H field `enq_qdepth` was less than 10.

There is a similar problem of implicit flows in dynamic information flow control, where simply upgrading a L variable to H in only one of the branches when pc is H might result in partial information leakage. This is because the variable contains H data in one execution while it might remain L on an alternative execution. To overcome this problem, many dynamic IFC methods employ the so-called no-sensitive-upgrade (NSU) check [62], which terminates the program's execution whenever a L variable is updated in a H context. Here, to overcome this problem, we type all the statements in all branches whenever the pc is H, even when the state type is empty [36, 132]. For instance, in Example E.5, we type-check the then branch under an empty state type, and by rule T-Cond the security labels of the final state types of both branches are joined, resulting in `hdr.ipv4.ecn`'s label being H in all of the final state types.

$$\textsc{T-Call}\quad \frac{\gamma \vdash \vec{e} : \vec{\tau} \qquad \text{tCall}(T, f, pc, \vec{\tau}, \gamma, \Gamma)}{T, pc, \gamma \vdash f(\vec{e}) : \Gamma}$$

T-Call rule types function calls. It individually types the function arguments e_i to obtain their types τ_i, and passes them to auxiliary function tCall, defined as:

$$\begin{array}{l}(s, \overline{(x, d)}) = T(f) \quad \gamma_f = \{x_i \mapsto \tau_i\} \quad T, pc, (\gamma_g, \gamma_f) \vdash s : \Gamma' \\ \Gamma = \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\}\end{array}$$

which retrieves the function's body s and its signature $\overline{(x, d)}$ from the mapping T. Creates a new local state type γ_f by assigning each argument it corresponding type (i.e. copy-in), and then types the function's body to obtain the resulting state type set Γ'. Finally, tCall produces Γ by copying out the `out` and `inout` parameters (identified by the isOut function), which means updating the passed lvalues (i.e. e_i) with the final types of their corresponding parameters (i.e. $\gamma'_f(x_i)$).

Example E.6

Assume that at line 44 of Program E.1, the `ttl` in the state type is mapped to $\langle 1, 10\rangle_8^L$. Calling `decrease` entails creating a new local state type and copying in the arguments, which yields $\gamma_{\texttt{decrease}} = \{x \mapsto \langle 1, 10\rangle_8^L\}$. Typing the function's body (`x = x - 1`) results in the state type $\gamma'_{\texttt{decrease}} = \{x \mapsto \langle 0, 9\rangle_8^L\}$. The final Γ'' is produced by copying out arguments back to the

> initial state type which would map `ttl` to $\langle 0, 9\rangle^{L}_{8}$.

In contrast to standard type systems, we directly type the body of the function, instead of typing functions separately and in isolation. The main reason is that the intervals and labels of the types of actual arguments can be different for each invocation of the function. Notice that the nested analysis of the invoked function does not hinder termination of our analysis since P4 does not support recursion, eliminating the need to find a fix point for the types [46].

T-Table

$$\frac{\begin{array}{c}(\overline{e}, \text{Cont}_{\text{tbl}}) = T(\mathit{tbl}) \qquad \gamma \vdash e_i : \tau_i \qquad \ell = \bigsqcup_{i} \text{lbl}(\tau_i) \qquad pc' = pc \sqcup \ell \\ \forall(\phi_j, (a_j, \overline{\tau}_j)) \in \text{Cont}_{\text{tbl}}. \qquad \gamma_j = \text{refine}(\gamma, \phi_j) \qquad \text{tCall}(T, a_j, pc', \overline{\tau}_j, (\gamma_g, \gamma_l), \Gamma_j)\end{array}}{T, pc, \gamma \vdash \texttt{apply}\ \mathit{tbl} : \text{joinOnHigh}(\cup_j \Gamma_j, \ell)}$$

T-Table rule is similar to T-Cond and T-Call. It relies on user-specified contracts to type the tables. A contract, as introduced in Section E.5, has the form $(\overline{e}, \text{Cont}_{tbl})$, where Cont_{tbl} consists of a set of triples $(\phi, (a, \overline{\tau}))$. Each triple specifies a condition ϕ, under which an action a is executed with arguments of specific types $\overline{\tau}$. A new context pc' is produced by the initial pc with the least upper bound of the labels of the keys.

For each triple $(\phi_j, (a_j, \overline{\tau}_j))$, T-Table relies on tCall to type the action a_j's body under pc', similar to T-Call, and accumulates the resulting state types into a set (i.e. $\cup_j \Gamma_j$). Finally, T-Table uses $\text{joinOnHigh}(\cup_j \Gamma_j, \ell)$ to join their labels if ℓ was H.

> **Example E.7**
>
> Given the table contract depicted in Figure E.5, assume a state type γ where pc is L and `hdr.ipv4.dstAddr` is typed as $\langle 192\rangle^{L}_{8} \cdot \langle 168\rangle^{L}_{8} \cdot \langle *\rangle^{L}_{16}$. According to T-Table, refining γ produces three state types, out of which only one is not empty: refine(γ, `dstAddr`$[31:24] = 192$). This refined state is used to type the action `ipv4_forward` with arguments $[\langle *\rangle^{H}_{48}, \langle 1, 9\rangle^{L}_{9}]$. The two empty states should be used to type the actions `ipv4_forward` (with arguments $[\langle *\rangle^{L}_{48}, \langle 10, 20\rangle^{L}_{9}]$) and `drop`. However, these states can be pruned by T-EmptyType, since pc' is L.

The rule for typing extern calls is similar to T-Call. Just like T-Table, it relies on user-defined contracts to capture the side effects of calling an extern. This rule is detailed in Appendix E.3.

Soundness

Given initial state types γ_1 and γ_2, and initial states m_1 and m_2, we write $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\approx}} m_2$ to indicate that $\gamma_1 \vdash m_1$, $\gamma_2 \vdash m_2$, and $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$.

The type system guarantees that a well-typed program terminates, and the final result is well-typed wrt. at least one of the resulting state types.

Lemma E.2 (Soundness of abstraction and labeling)

Given initial state types γ_1 *and* γ_2*, and initial states* m_1 *and* m_2*, such that* $T, pc, \gamma_1 \vdash s : \Gamma_1$ *and* $T, pc, \gamma_2 \vdash s : \Gamma_2$*, and* $E_1 \underset{T}{\sim} E_2$*, and* $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\approx}} m_2$ *then there exists* m_1' *and* m_2' *such that* $E_1 : m_1 \xrightarrow{s} m_1'$*,* $E_2 : m_2 \xrightarrow{s} m_2'$*,* $\gamma_1' \in \Gamma_1$*,* $\gamma_2' \in \Gamma_2$*, and* $m_1' \overset{\gamma_2'}{\underset{\gamma_1'}{\approx}} m_2'$*.*

Lemma E.2 states that starting from two indistinguishable states wrt. $\gamma_1 \sqcup \gamma_2$, a well-typed program results in two indistinguishable states wrt. *some* final state types in Γ_1 and Γ_2 that can also type the resulting states m_1' and m_2'.

We rely on Theorem E.1 to establish noninterference, that is, if every two states m_1 and m_2 that are indistinguishable wrt. *any* two final state types are also indistinguishable by the output policy, then the program is noninterfering:

Theorem E.1 (Noninterference)

Given input policy case γ_i *and output policy* Γ_o*, if* $T, pc, \gamma_i \vdash s : \Gamma$ *and for every* $\gamma_a, \gamma_b \in \Gamma$ *such that* $m_1 \overset{\gamma_b}{\underset{\gamma_a}{\approx}} m_2$ *it also hold that* $m_1 \overset{\gamma_o}{\underset{\gamma_o}{\approx}} m_2$ *for all* $\gamma_o \in \Gamma_o$*, then* s *is noninterfering wrt. the input policy case* γ_i *and the output policy* Γ_o*.*

Theorem E.1 is required to be proved for *every* possible pair of states. To make the verification process feasible, we rely on the following lemma to show that this condition can be verified by simply verifying a relation between the final state types (Γ) and the output policy (Γ_o):

Lemma E.3 (Sufficient condition)

If for every $\gamma_1, \gamma_2 \in \Gamma$ *and every* $\gamma_o \in \Gamma_o$ *such that* $\gamma_1 \sqcap \gamma_o \neq \bullet$*, it holds that:*

(1) if $\gamma_2 \sqcap \gamma_o \neq \bullet$*, then* $\gamma_1 \sqcup \gamma_2 \sqsubseteq \gamma_o$*,*

(2) for every lval either $\gamma_2(lval) \subseteq \gamma_o(lval)$ *or* $\gamma_1 \sqcup \gamma_2(lval) = L$ *hold,*

> *then for every* $\gamma_1, \gamma_2 \in \Gamma$ *such that* $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\sim}} m_2$*, and every* $\gamma_o \in \Gamma_o$*, it holds that if* $\gamma_o \vdash m_1$ *then* $\gamma_o \vdash m_2$ *and* $m_1 \underset{\gamma_o}{\sim} m_2$*.*

where $\gamma_2(\mathit{lval}) \subseteq \gamma_o(\mathit{lval})$ indicates that the interval of *lval* in γ_2 is included in the interval specified in γ_o.

Intuitively, Lemma E.3 formalizes that the least upper bound of any pair in the set of final state types (Γ) should not be more restrictive than the output policy (e.g. if H information has flown to a variable, that variable should also be H in the output policy cases) *and* the abstractions specified in the output policy cases (i.e. the intervals) are either always satisfied or do not depend on H variables.

Revisiting the basic congestion program

We revisit Program E.1 to illustrate some key aspects of our typing rules. Here, we only consider the first policy case of the input policy (1) of Section E.2, where the input packet is IPv4 and it is coming from the internal network.

For the initial state derived from this input policy case, since (1) the parser's transitions depend on L variables, (2) the type system does not merge state types, and (3) the type system prunes the unreachable transition to `accept` from `parse_ethernet`, then the parser terminates in a single state type where both `hdr.eth` and `hdr.ipv4` are valid, and their respective headers include the slices, intervals, and labels defined by the initial state type.

After the parsing stage is finished, the program's control flow reaches the `MyCtrl` control block. Since `hdr.ipv4` is valid and *pc* is L, pruning empty states allows us to ignore the `else` branch on line 62. Afterwards, the nested if statement at line 53 entails two possible scenarios. First scenario, when the destination address is in range `192.168.*.*`, as described in Example E.4, the two state types resulting from the `if` at line 56 have `hdr.ipv4.ecn` set to H. As in Example E.7, these state types satisfy only the first condition of the table's contract, which results in assigning the type $\langle 1, 9\rangle_9^L$ to `egress_spec` and producing the state types γ_{int}^1 and γ_{int}^2.

Second scenario, when the destination address on line 53 does not match `192.168.*.*`, the state type is refined for the `else` branch, producing one state type under condition `ipv4.dstAddr` $\geq$ `192.169.0.0` and one under `ipv4.dstAddr` $<$ `192.168.0.0`. For both of these state types, `hdr.ipv4.ecn` is set to $\langle 0\rangle_2^L$ by assignment on line 59. Since in this case all branch conditions were L, there is no H field left in the headers. The first of these two refined state types only satisfies the first condition of the table contract, resulting in one single (after pruning empty states) final state type, γ_{int}^3, where the packet has been forwarded to the internal network and `egress_spec` is set to $\langle 1, 9\rangle_9^L$. The second refined state type however satisfies all the conditions of the table contract, resulting in three final state types γ^{4}_{int}, γ^{1}_{ext}, γ^{1}_{drop} with `egress_spec` being set to $\langle 1, 9\rangle^{L}_{9}$, $\langle 10, 20\rangle^{L}_{9}$, and $\langle 0\rangle^{L}_{9}$, respectively. Notice that among these states, only γ^{3}_{int} and γ^{4}_{int} contains a H fields (i.e. `ipv4.dstAddr`) due to the first argument returned by the table being $\langle * \rangle^{H}_{48}$.

We finally check the sufficient condition for the output policy (2) and its only output policy case γ_o, which states that when `egress_spec` is $\langle 10, 20\rangle^{L}_{9}$ (i.e. the packet leaves the internal network) all header fields are L. Only state type γ^{1}_{ext} matches the output policy case (i.e. $\sqcap\gamma_o \neq \bullet$), and this state type satisfies $\gamma^{1}_{ext} \sqcup \gamma^{1}_{ext} \sqsubseteq \gamma_o$ since all header fields and `egress_spec` are L in γ^{1}_{ext}. All other state types (i.e. γ^{1}_{int}, γ^{2}_{int}, γ^{3}_{int}, γ^{4}_{int}, and γ^{1}_{drop}) do not match the output policy condition (i.e. $\sqcap\gamma_o = \bullet$), since they do not correspond to packets sent to the external network (i.e. their `egress_spec` is not in range $\langle 10, 20\rangle$). Therefore, we conclude that for this this specific input policy case, Program E.1 is noninterfering wrt. the output policy case γ_o.

Our analysis can also detect bugs, such as the one on line 54 of Program E.1. To illustrate this, assume that the program is buggy and instead of checking the 8 most significant bits (i.e. [31:24]) of the `hdr.ipv4.dstAddr`, it checks the least significant bits (i.e. [7:0]). This means that IPv4 packets with destination address is in range `*.168.*.192` would satisfy the condition of the `if` statement on line 53. Similar to the non-buggy program, the `if` at line 56 would produce two state types with `hdr.ipv4.ecn` set to H. These state types satisfy all the conditions of the table contract. For presentation purposes, let us focus on only one of these state types. Applying the table on line 61 would produce three final state types γ^{1}_{int}, γ^{1}_{ext}, γ^{1}_{drop} with `egress_spec` being set to $\langle 1, 9\rangle^{L}_{9}$, $\langle 10, 20\rangle^{L}_{9}$, and $\langle 0\rangle^{L}_{9}$, respectively. Note that in all these final state types, `hdr.ipv4.ecn` is H. When checking the sufficient condition, state type γ^{1}_{ext} matches the output policy case (i.e. $\sqcap\gamma_o \neq \bullet$) but it does not satisfy $\gamma^{1}_{ext} \sqcup \gamma^{1}_{ext} \sqsubseteq \gamma_o$, because the `hdr.ipv4.ecn` field H in γ^{1}_{ext} and L γ_o. Hence this buggy program will be marked as interfering, highlighting the fact that some of the packets destined for the external network contain congestion information and unintentionally leak sensitive information.

The benefit of value- and path-sensitivity of our approach can also be demonstrated here. For all other input policy cases that describe non-IPv4 packets, the `parse_ipv4` state is not going to be visited. A path-insensitive analysis, which merges the results of the parser transitions, would lose the information about the validity of the `hdr.ip4` header. This would then lead to the rejection of the program as insecure because an execution where the `parse_ipv4` state has not been visited, yet the `if` branch on line 52 has been taken, will be considered feasible. Our analysis, on the other hand, identifies that any execution that has not visited `parse_ipv4` results in an invalid `hdr.ip4` header. Consequently, for all such executions, it produces a final state type where the packet is dropped, and `egress_spec` is set to $\langle 0\rangle^{L}_{9}$. This state type satisfies the sufficient condition, since `egress_spec` does not intersect γ_o(`egress_spec`) and is L.

Table E.1: Evaluation results

	Time (ms)			
	Total	**Typing**	**Security Check**	**Number of Final γs**
Basic Congestion	5930	966	4794	97
Basic Tunneling	610	157	290	15
Multicast	199	16	23	6
Firewall	4560	1015	3378	44
MRI	7646	523	6957	23
Dataplane Routing	274	109	8	12
In-Network Caching	261	91	14	6
Resource Allocation	256	87	10	9
Network Isolation - Alice	243	27	62	3
Network Isolation - Top	242	23	63	3
Topology	202	40	4	3

E.7 Implementation and Evaluation

To evaluate our approach we developed TAP4S[2], a prototype tool which implements the security type system of Section E.6. TAP4S is developed in Python and uses the lark parser library [201] to parse P4 programs.

TAP4S takes as input a P4 program, an input policy, and an output policy. Initially, it parses the P4 program, generates an AST, and relies on this AST and the input policy γ_i to determine the initial type of input packet fields and the standard metadata. Because the input policy is data-dependent, the result of this step can generate multiple state types $(\gamma_1, \ldots, \gamma_m)$, one state type for each input interval. TAP4S uses each of these state types as input for implementing the type inference on the program. During this process TAP4S occasionally interacts with a user-defined contract file to retrieve the contracts of the tables and externs. Finally, TAP4S yields a set of final state types $(\gamma'_1, \ldots, \gamma'_n)$ which are checked against an output policy, following the condition in Lemma E.3. If this check is successful the program is deemed secure wrt. the output policy, otherwise the program is rejected as insecure.

Test suite. To validate our implementation we rely on a functional test suite of 25 programs. These programs are P4 code snippets designed to validate the support for specific functionalities of our implementation, such as extern calls, refinement, and table application.

[2] https://github.com/amir-ahmadian/TAP4S

Use cases. We evaluate Tap4s on 5 use cases, representing different real-world scenarios. The results of this evaluation are summarized in Table E.1. Due to space constraints, detailed descriptions of these use cases are provided in Appendix E.1. We also implement and evaluate the use cases from P4BID [192]. These use cases are described in Appendix E.2, and their corresponding evaluation results are included in Table E.1. They serve as a baseline for comparing the feasibility of Tap4s with P4BID. On average, P4BID takes 30 ms to analyze these programs, whereas Tap4s takes 246 ms. Despite the increased time, this demonstrates that Tap4s performs the analysis with an acceptable overhead. On the other hand, due to the data-dependent nature of our use cases and their reliance on P4-specific features such as slicing and externs, P4BID cannot reliably check these scenarios, leading to their outright rejection in all cases.

E.8 Related Work

IFC for P4. Our work draws inspiration from P4BID [192], which adapts and implements a security type system [18] for P4, ensuring that well-typed programs satisfy noninterference. By contrast, we show that security policies are inherently data-dependent, thus motivating the need for combining security types with interval-based abstractions. This is essential enforcing IFC in real-world P4 programs without code modifications, as demonstrated by our 5 use cases. Moreover, our analysis handles P4 features such as slicing and externs, while supporting the different stages of the P4 pipeline, beyond a single control block of the match-action stage.

IFC policy enforcement. Initial attempts at enforcing data-dependent policies [94, 96, 145, 163, 165] used dynamic information flow control. The programmer declaratively specifies data-dependent policies and delegates the enforcement to a security-enhanced runtime, thus separating the policy specification from the code implementation.

Our approach shares similarities with static enforcement of data-dependent IFC policies such as *dependent information flow types* and *refinement information flow types*. Dependent information flow types [117] rely on dependent type theory and propose a dependent security type system, in which the security level of a type may depend on its runtime value. Eichholz et al. [190] introduced a dependent type system for the P4 language, called Π4, which ensures properties such as preventing the forwarding of expired packets and invalid header accesses. Value-dependent security labels [100] partition the security levels by indexing their labels with values, resulting in partitions that classify data at a specific level, depending on the value. Dependent information flow types provide a natural way to express data-centric policies where the security level of a data structure's field may depend on values stored in other fields.

Later approaches have focussed on trade-offs between automation and decidability of the analysis. Liquid types [103, 112] are an expressive yet decidable refinement type system [182] to statically express and enforce data-dependent information flow polices. Lifty [176] provides tool support for specifying data-dependent policies and uses Haskell's liquid type-checker [112] to verify and repair the program against these policies. Storm [183] is a web framework that relies on liquid types to build MVC web applications that can statically enforce data-dependent policies on databases using liquid types.

Our interval-based security types can be seen as instantiations of refinement types and dependent types. Our simple interval analysis appears to precisely capture the key ingredients of P4 programs, while avoiding challenges with more expressive analysis. By contrast, the compositionality of analysis based on refinement and dependent types can result in precision loss and is too restrictive for our intended purposes (as shown in Section E.6), due to merging types of different execution paths. We solve this challenges by proposing a global path-sensitive analysis that avoids merging abstract state in conditionals. We show that our simple yet tractable abstraction is sufficient to enforce the data-dependent policies while precisely modeling P4-specific constructs such as slicing, extract, and emit.

Other works use abstract interpretation in combination with IFC. De Francesco and Martini [54] implement information-flow analysis for stack-based languages like Java. They analyze the instructions an intermediate language by using abstract interpretation to abstractly execute a program on a domain of security levels. Their method is flow-sensitive but not path-sensitive. Cortesi and Halder [106] study information leakage in databases interacting with Hibernate Query Language (HQL). Their method uses a symbolic domain of positive propositional formulae that encodes the variable dependencies of database attributes to check information leaks. Amtoft and Banerjee formulate termination-insensitive information-flow analysis by combining abstract interpretation and Hoare logic[32]. They also show how this logic can be extended to form a security type system that is used to encode noninterference. This work was later extended to handle object-oriented languages in [42].

Analysis and verification of network properties. Existing works on network analysis and verification do not focus on information flow properties. Symbolic execution is widely used for P4 program debugging, enabling tools to explore execution paths, find bugs, and generate test cases. Vera [155] uses symbolic execution to explore all possible execution paths in a P4 program, using symbolic input packets and table entries. Vera catches bugs such as accesses fields of invalid headers and checking that the `egress_spec` is zero for dropped packets. Additionally, it allows users to specify policies, such as; ensuring that the NAT table translates packets before reaching the output ports, and the NAT drops all packets if its entries are empty. Recently, Scaver [196] uses symbolic execution to verify forwarding properties of P4 programs. To address the path explosion problem, they propose multiple pruning strategies to reduce the number of explored paths. ASSERT-P4 [153] combines symbolic execution with assertion checking to find bugs in P4 programs, for example, that the packets with TTL value of zero are not dropped and catching invalid fields accesses. Tools like P4Testgen [195] and p4pktgen [154] use symbolic execution to automatically generate test packets. This approach supports test-driven development and guarantees the correct handling of packets by synthesizing table entries for thorough testing of P4 programs.

Abstract interpretation has also been used to verify functional properties such as packet reachability and isolation. While these properties ensure that packets reach their intended destinations, they do not address the flow of information within the network. Alpernas et al. [151] introduce an abstract interpretation algorithm for networks with stateful middleboxes (such as firewalls and load balancers). Their method abstracts the order and cardinality of packet on channels, and the correlation between middleboxes states, allowing for efficient and sound analysis. Beckett et al. [158] develop ShapeShifter, which uses abstract interpretation to abstract routing algebras to verify reachability in distributed network control-planes, including objects such as path vectors and IP addresses and methods such as path lengths, regular expressions, intervals, and ternary abstractions.

E.9 Conclusion

This paper introduced a novel type system that combines security types with interval analysis to ensure noninterference in P4 programs. Our approach effectively prevents information leakages and security violations by statically analyzing data-dependent flows in the data-plane. The type system is both expressive and precise, minimizing overapproximation while simplifying policy specification for developers. Additionally, our type system successfully abstracts complex elements like match-action blocks, tables, and external functions, providing a robust framework for practical security verification in programmable networks. Our implementation, TAP4S, demonstrated the applicability of the security type system on real-world P4 use cases without losing precision due to overapproximations. Future research includes adding support for declassification, advanced functionalities such as cryptographic constructs, and extending the type system to account for side channels.

Appendices

Appendix A Use Cases

Basic Tunneling

Our first use case, shown in Program E.2, outlines procedures for handling standard IPv4 packets and encapsulated tunneling packets. The parser `MyParser` starts by extracting the Ethernet header on line 6, and for `etherType` 0x1212 (tunneled packet), it transitions to `parse_myTunnel` state (line 14), extracts the tunnel header, checks the `proto_id` field, and transitions to `parse_ipv4` state (line 21) if an IPv4 packet is indicated. For `etherType` 0x0800 (IPv4 packet), it directly transitions to `parse_ipv4` and extracts the IPv4 header. Once the headers are parsed, the pipeline proceeds to the `MyCtrl` control block, starting from the apply block on line 42 which contains two if statements: If only the IPv4 header is valid (line 43), the `ipv4_lpm` table is applied which forward or drop the packet based on a longest prefix match (lpm) on the destination IPv4 address. If the tunnel header is valid (line 46), the `myTunnel_exact` table forwards the packet based on an exact match of the `myTunnel` header's `dst_id`, using the `myTunnel_forward` action.

Since the header fields of the tunneled packets are not modified while they are forwarded (Lines 34-36), to keep the source MAC address of the packets within the internal networks private, the program should not forward tunneled packets to an external network. The input policy in this use case indicates that if the input packet the packet is tunneled (i.e. its `etherType` is 0x1212) then the packet's `srcAddr` is H. The output policy relies on the output port the packet is sent to, and ensures that "if the `egress_spec` is between 10-511 then the packet has left the internal network, therefore all field of the packet's headers should be L."

The general behavior of table `ipv4_lpm` is reflected in its contract as it makes sure the packets with `ipv4` destination address `198.*.*.*` are forwarded to the ports connected to the internal network, while all the other destination addresses are forwarded to ports connected to the external network.

The contract of table `myTunnel_exact` plays a crucial role in the security of Program E.2. A `correct` behavior for this table only forwards the tunneled packets to ports connected to the internal network.

```
struct headers {
    ethernet_t  eth; myTunnel_t  myTunnel; ipv4_t  ipv4;
}
parser MyParser(/* omitted */) {
    state start { transition parse_ethernet; }
    state parse_ethernet {
        packet.extract(hdr.eth);
        transition select(hdr.eth.etherType) {
            0x1212: parse_myTunnel;
            0x0800: parse_ipv4;
            default: accept;
        }
    }
    state parse_myTunnel {
        packet.extract(hdr.myTunnel);
        transition select(hdr.myTunnel.proto_id) {
            0x0800: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}

control MyCtrl(/* omitted */) {
    action drop()
        { /* drops the packet */ }
    action ipv4_forward(bit<48> dstAddr, bit<9> port)
        { /* basic forward */ }
    table ipv4_lpm
        { /* omitted */}

    action myTunnel_forward(bit<19> port) {
        standard_metadata.egress_spec = port;
    }
    table myTunnel_exact {
        key = {hdr.myTunnel.dst_id: exact;}
        actions = { myTunnel_forward; drop;}
        default_action = drop();
    }
    apply {
        if (hdr.ipv4.isValid() && !hdr.myTunnel.isValid()) {
            ipv4_lpm.apply(); // Process non-tunneled packets
        }
        if (hdr.myTunnel.isValid()) {
            myTunnel_exact.apply(); // Process tunneled packets
        }
    }
}
```

Program E.2: Basic tunneling

The evaluation reported in Table E.1 is performed under a contract that reflected this behavior, which results in DiVerT accepting the program as secure. If this table is somehow misconfigured and forwards the tunneled packet to any port connected to the external network, DiVerT can capture this and flag the program as insecure.

Multicast

Our next use case is Program E.3 which is capable of multicasting packets to a group of ports. Upon receiving a packet, the switch looks up its destination MAC address `dstAddr`, if it is destined to any of the hosts connected to the switch, the packet is forwarded to its destination (line 11), otherwise the switch broadcasts the packet on ports belonging to a multicast group by setting the `standard_metadata.mcast_grp` to 1 (line 9). Figure E.6 illustrates the network schema of this scenario.

```
1   struct headers {
2       ethernet_t  ethernet;
3   }
4   control MyCtrl(/* omitted */) {
5       action drop() {
6           mark_to_drop(standard_metadata);
7       }
8       action multicast() {
9           standard_metadata.mcast_grp = 1;
10      }
11      action mac_forward(bit<9> port) {
12          standard_metadata.egress_spec = port;
13      }
14      table mac_lookup {
15          key = { hdr.ethernet.dstAddr : exact; }
16          actions = { multicast; mac_forward; drop; }
17      }
18      apply {
19          if (hdr.ethernet.isValid())
20              mac_lookup.apply();
21      }
22  }
```

Program E.3: Multicast

To implement this functionality, the program utilizes the table `mac_lookup` which is populated by the control plane, and contains the mac addresses and the port information needed to forward non-multicast packets.

While the broadcast packets are sent to all of the multicast ports, it is desirable to ensure the packets that are not supposed to be broadcast are indeed not broadcasted. Our input security policy in this scenario sets the packets destined to any of the hosts connected to the switch as H, while labeling the broadcast packets L. The

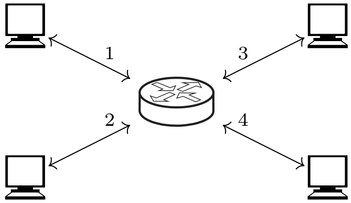


Figure E.6: Multicast schema

contract of the table `mac_lookup`'s needs to capture the essence of this use case, that is, packets with the `dstAddr` of any of the hosts need to be forwarded by invoking the `mac_forward` action (line 11), and all the other packets need to be broadcast by invoking the `multicast` action (line 8). To ensure the program behaves desirably, the output policy checks that all the packets send to the multicast ports (which have their `mcast_grp` set to 1 according to line 9) are L.

As illustrated in Table E.1, under these policies and contracts, Program E.3 is secure. It results in 6 final state types (γ), and takes approximately 220 milliseconds to verify the program is policy compliance.

Firewall

This use case models a scenario where the switch is running the firewall Program E.4 which allows it to monitor the connections between an internal and an external network. The network schema of this scenario is presented in Figure E.7.

After parsing an input packet, the switch applies the `ipv4_lpm` table (line 26), which based on the packet's IPv4 destination address forwards or drops the packet. Next, it applies the `check_ports` table, which based on the input port number (`ingress_port`) identifies whether the packet is coming from the external or the internal network. As depicted in Figure E.7 port 4 is connected to the external network and ports $1 - 3$ are connected to the hosts of the internal network, therefor if the standard metadata's `ingress_port` was 4, the `check_ports` table sets the direction to 1 which indicates the packet is coming from the external network.

The policy of the firewall is that the hosts in the internal network are allowed to communicate with the outside networks, but the hosts in the external network are only allowed to `ssh` to the internal hosts. To this end, for all the packets with direction 1, the program will drop all the packets whose tcp port (`hdr.tcp.srcPort`) is not 22 (line 31).

To enforce such policy, we rely on integrity labels (instead of confidentiality) to designate which packets are allowed (trusted), and which packets are not. The input security policy in this scenario sets the packets coming the internal network,

```
header tcp_t{
    bit<16> srcPort;
    // omitted
}
struct headers {
    ethernet_t  ethernet; ipv4_t  ipv4; tcp_t  tcp;
}
control MyCtrl(/* omitted */) {
    action drop()
        { /* drops the packet */ }
    action ipv4_forward(bit<48> dstAddr, bit<9> port)
        { /* basic forward */ }
    table ipv4_lpm
        { /* omitted */}

    action set_direction(bit<1> dir) {
        direction = dir;
    }
    table check_ports {
        key = { standard_metadata.ingress_port: exact; }
        actions = { set_direction; NoAction; }
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
            if (hdr.tcp.isValid()) {
                check_ports.apply();
                if (direction == 1) {// Packet is from outside
                    // only allow ssh connections from outside
                    if (hdr.tcp.srcPort != 22) { drop(); }
                }
            }
        }
    }
}
```

Program E.4: Firewall

identified by their `ingress_port` as trusted (L), while any packet coming from the external network (with `ingress_port` 4) is untrusted (H) except when its TCP source port `srcPort` is 22.

The contract of the `ipv4_lpm` captures the behavior of this table by making sure the packets with `ipv4` destination address `198.*.*.*` are forwarded to the ports connected to the internal network (ports 1 to 3), and all the other destination addresses are forwarded to port 4. The contract of table `check_ports` updates the direction by checking the `ingress_port` of the incoming packet, setting the direction to 1 if the `ingress_port` was 4.

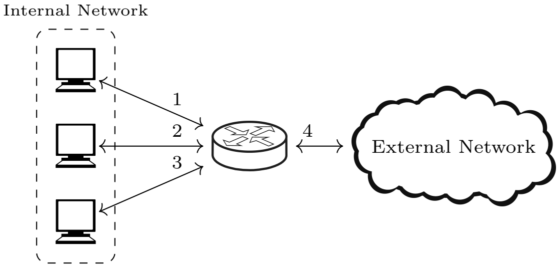


Figure E.7: Firewall schema

The output policy checks that all the packets leaving the switch are trusted and L. Table E.1 depicts the results of DiVerT for this use case. Under these policies and contracts Program E.3 is deemed secure. DiVerT produces 44 final state types, and takes approximately 6 seconds to verify the security of the program.

Multi-Hop Route Inspection

Program E.5 implements a simplified version of In-Band Network Telemetry, called Multi-Hop Route Inspection (MRI). The purpose of MRI is to let the users to track the path and the length of queues that every packet travels through. To do this, the P4 program adds an ID and queue length to the header stack of every packet (line 10). Upon reaching the destination, the sequence of switch IDs shows the path the packet took, and each ID is followed by the queue length at that switch.

After parsing the packet, the program applies the `ipv4_lpm` table to forward the packet based on its IPv4 destination address. Afterwards, in the *MyEgress* control block, the `swtrace` table (line 27), based on the port information specified in the `egress_spec`, decides whether to add the queue length data to `swtraces` header or not.

While it makes sense to add this information for packets that are traveling within a local network, similar to the basic congestion example (Program E.1) the id of the switches and their queue length can give an external adversary information about the state of the local network. Therefore it is desirable to protect the local network by making sure that Program E.5 only adds this data to the packets being forwarded within the local network.

The input policy of this scenario labels the input packet as L and only marks the `deq_qdepth` of the standard metadata as H. The contract of the `ipv4_lpm` table forwards the packets with `ipv4` destination address `198.*.*.*` to the ports connected to the internal network, while all the other destination addresses are forwarded to ports connected to the external network. If the `egress_spec` indicates ports connected to the internal network, the contract of the `swtrace` table invokes the

```
header switch_t {
    switchID_t  swid;
    qdepth_t  qdepth;
}
struct headers {
    ethernet_t  ethernet;
    ipv4_t  ipv4;
    ipv4_option_t  ipv4_option;
    mri_t  mri;
    switch_t[9]  swtraces;
}
control MyIngress(/* omitted */) {
    action drop()
        { /* drops the packet */ }
    action ipv4_forward(bit<48> dstAddr, bit<9> port)
        { /* basic forward */ }
    table ipv4_lpm
        { /* omitted */}
    apply {
        if (hdr.ipv4.isValid()) { ipv4_lpm.apply(); }
    }
}
control MyEgress(/* omitted */) {
    action add_swtrace(switchID_t swid) {
        // updates the swtraces header
    }
    table swtrace {
        key = { standard_metadata.egress_spec: exact; }
        actions = { add_swtrace; NoAction;}
    }
    apply {
        if (hdr.mri.isValid()) {
            swtrace.apply();
        }
    }
}
```

Program E.5: Multi-Hop Route Inspection

add_swtrace action, adding the queue length data to swtraces header, otherwise NoAction takes place.

The output policy ensures that in all of the packets going to the external network, identified by their hdr.ipv4.dstAddr being anything other than 198.*.*.*, have L switch_t header.

Table E.1 depicts the results of type checking this program with DiVerT. It generates 23 final state types and takes approximately 4 seconds to verify the security of the program. swtrace table is crucial for the security of this Program and if it is misconfigured and calls the add_swtrace action on outgoing packets, the program will be rejected by DiVerT.

Appendix B P4BID Use Cases

We implemented the use cases of P4BID [192] in DiVerT to ensure that it can correctly evaluate all of their use cases and that its verdict is inline with the results reported in [192]. These use cases and their corresponding policies are simpler than our own use cases because P4BID does not support data-dependent policies and hence only labels program variables without taking the value of the packet header fields into account. The results of this evaluation is depicted in Table E.1.

Dataplane Routing *Routing* is the process of determining how to send a packet from its source to its destination. In traditional networks the control plane is responsible for routing, but recently, Subramanian et al. [187] proposed an approach to implement the routing in the data plane. Their approach uses pre-loaded information about the network topology and link failures to perform a breadth-first search (BFS) and find a path to the destination.

In this scenario we do not care about the details of this BFS search algorithm, but we want to make sure that the sensitive information about the private network (such as the number of hops in the network) do not leak to an external network. Similar to P4BID [192] labeling the number of hops as H will result in the program being rejected by DiVerT because the forwarding action uses this information to update the packet's priority field, which results in an indirect leakage of sensitive information.

In-Network Caching In order to enable the fast retrieval of popular items, switches keep track of the frequently requested items in a cache and only query the controller when an item cannot be found in the cache. Similar to any cache system, the result of a query is the same regardless of where the item is stored. However, from a security perspective, an observer can potentially detect variations in item retrieval time. This timing side-channel can potentially allow an adversary to learn about the state of the system.

To model the cache in this scenario, we mark the request query as sensitive, because whether this query is a *hit* or a *miss* leaks information about the internal state of the switch. The variable marking the state of the result in the cache (`response.hit`) is not sensitive because it is considered observable by the adversary. Similar to P4BID [192] this labeling will result in the program being rejected by DiVerT because a sensitive query can indirectly affect the value of `response.hit`, resulting in the leakage of sensitive information.

Resource Allocation This use case models a simple resource allocation program, where the switch increases the priority of the packets belonging to latency-sensitive applications. The application ID in the packet's header will indicate which application the packet belongs to. A table will matches on this application ID and sets the packet's priority by modifying the `priority` field of the `ipv4` header.

The problem is that a malicious client can manipulate the application ID to increase the priority of their packets. We rely on integrity labels (instead of confidentiality) to address this issue, that is, the application ID will be labeled as untrusted (H) and the `ipv4 priority` field will be labeled as trusted (L). Since the program sets the `priority` field based on the value of the application ID, the `priority` field will also be labeled untrusted by the type system, which results in the rejection of the program.

Network Isolation This use case models a private network used by two clients, Alice and Bob. Each client runs its own P4 program, but the packets sent between these two clients have a shared header with separate fields for Alice and Bob. In this scenario we want to make sure that Alice does not touch Bob's fields, and vice versa.

The isolation property in this example can be modeled by a four-point lattice with labels $\{A, B, \top, \bot\}$, where A is the label of Alice's data, B is for Bob's data, $\top$ is the top element confidential to both Alice and Bob, and $\bot$ is public. By IFC, data from level ℓ can flow to ℓ' if and only if $\ell \sqsubseteq \ell'$.

In this use case, we consider Alice's program in which she updates the fields belonging to herself. Additionally, we use label $\top$ to label the telemetry data which can be updated by Alice's program, but she cannot leak information from $\top$-labeled data into her own fields.

Since DiVerT only support simple lattice with two levels, we type check this program twice, with two policies. First where Alice is H and everything else is L, and a second time where $\top$ is H and everything else is L. The same process can be repeated for Bob's program as well. This program is accepted by DiVerT, and the results for both cases are reported in Table E.1.

Topology This use case is a P4 program which processes packets as they enters a local network. The incoming packets refers to a virtual address which needs to be translated to a physical address as the packet is routed in the local network.

Our security policy dictates that the routing details of this local network should not leak into fields that are visible when the packet leaves the network. As such, the program relies on a separate header to store the local information, and as long as the packet is inside the local network, the switches do not modify the `ipv4` and Ethernet headers, instead, they parse, use, and update this local header with the routing information.

As explained in P4BID [192], this program has a bug where it incorrectly stores the local `ttl` in the `ipv4` header instead of the local header. Marking the local fields as H, DiVerT flags this program as insecure and facilitates the process of catching and fixing these types of errors.

Appendix C Typing Rules

In this section we present the typing rule for the externs:

$$\textsc{T-Extern}\quad \dfrac{\begin{array}{l} (\gamma_g, \gamma_l) \vdash e_i : \tau_i \qquad (\text{Cont}_\text{E}, (x_1, d_1), ..., (x_n, d_n)) = T(f) \\ \quad \gamma_f = \{x_i \mapsto \tau_i\} \qquad \forall(\gamma_i, \phi, \gamma_t) \in \text{Cont}_\text{E}.\ (\gamma_g, \gamma_f) \sqsubseteq \gamma_i \\ \Gamma' = \{\gamma' \mathbin{+\!\!+} \text{raise}(\gamma_t, pc) \mid (\gamma_i, \phi, \gamma_t) \in \text{Cont}_\text{E} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \text{refine}((\gamma_g, \gamma_f), \phi) = \gamma' \neq \bullet\} \\ \quad \Gamma'' = \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\} \end{array}}{T, pc, (\gamma_g, \gamma_l) \vdash f(e_1, ..., e_n) : \Gamma''}$$

T-Extern types the invocation of external functions. It is similar to T-Call with the main difference that the semantics of external functions are not defined in P4, therefore, we rely on user-specified contracts to approximate their behavior. An extern contract is a set of tuples $(\gamma_i, \phi, \gamma_t)$, where γ_i is the input state type, ϕ is a boolean expression defined on the parameters of the extern, and γ_t indicates the state type components updated by the extern function (i.e. its side effects).

γ_i denotes a contract-defined state type that must be satisfied prior to the invocation of the extern, and the rule T-Extern ensure that the initial state (γ_g, γ_f) is at most as restrictive as γ_i. This approach is standard in type systems where functions are type-checked in isolation using predefined pre- and post-typing environments. For each $(\gamma_i, \phi, \gamma_t)$ tuple in the contract, T-Extern refines the initial state type (γ_g, γ_l) by ϕ yielding γ', and filters out all γ's that do not satisfy ϕ (i.e. the refinement $\text{refine}((\gamma_g, \gamma_f), \phi)$ is $\bullet$). This is sound because we assume for all the variables appeared in ϕ, the least upper bound of their labels within γ_i is less restrictive than the lower bound of γ_t. We raise the label of all elements in the γ_t to pc to capture indirect flows arising from updating the state type γ' in a H context, and then use use $\mathbin{+\!\!+}$ operation to update γ' with the types in γ_t. The final state type set Γ' is produced by copying out the `out` and `inout` parameters from γ'.

Example E.1. In Program E.1, let the contract for `mark_to_drop` at line 38 be defined as:

$$(\{\texttt{egress\_spec} \mapsto \langle * \rangle_9^L\}, \text{true}, \{\texttt{egress\_spec} \mapsto \langle 0 \rangle_9^L\})$$

which indicates that given an input state type $\{\texttt{egress\_spec} \mapsto \langle * \rangle_9^L\}$ the extern always sets the value of `egress_spec` to zero. Assuming an initial state type $\gamma = \{\texttt{egress\_spec} \mapsto \langle 7 \rangle_9^L\}$. Since the condition of the contract is true the refinement in this state type does not modify γ. This state type will be updated with the contract's γ_t to become $\{\texttt{egress\_spec} \mapsto \langle 0 \rangle_9^L\}$ if pc is L, otherwise $\{\texttt{egress\_spec} \mapsto \langle 0 \rangle_9^H\}$.

To guarantee the abstraction soundness of externs, for any input state m to the externs semantics $m' = sem_f(m)$ and the contracts set $(\gamma_i, \phi, \gamma_t)$ must satisfy the following properties:

1. Every input state m must satisfy some condition in the contract set ϕ, i.e. $\exists \phi\ .\phi(m)$
2. All modified variables in output state m' must be in the domain of γ_t, and their abstraction types in γ_t must hold, i.e. $\{x.\ m(x) \neq m'(x)\} \subseteq \text{domain}(\gamma_t)$, and for all $x \in \text{domain}(\gamma_t)$ holds $m'(x) : \gamma_t(x)$.

Additionally, to guarantee the labeling soundness of externs, the contracts must satisfy the following properties:

1. Conditions must preserve secrecy with respect to the output state type. For all variable names $\{x_1, ..., x_n\}$ appearing in the contract's condition ϕ, holds $\text{lbl}(\gamma_i(x_1)) \sqcup ... \sqcup \text{lbl}(\gamma_i(x_n)) \sqsubseteq lb(\gamma_t)$.
2. Extern semantics must preserve low-equivalence. Given any states m_1 and m_2, If $\phi(m_1)$ and $m_1 \underset{\gamma_i}{\sim} m_2$, then $m'_1 = sem_f(m_1)$, $m'_2 = sem_f(m_2)$, then the difference between the two output states must be also low equivalent $(m'_1 \setminus m_1) \underset{\gamma_t}{\sim} (m'_2 \setminus m_2)$.

Appendix D State Type Operations

A type τ' is considered an overapproximation of type τ (written as $\tau \leq \tau'$) iff for every value $v : \tau$ it holds that $v : \tau'$ and $\text{lbl}(\tau) \sqsubseteq \text{lbl}(\tau')$. We denote two non-overlapping lvalues by $lval \divideontimes lval'$, which means if *lval* is a record *lval'* is not one of its fields, and if *lval* is a bitvector *lval'* is not one of its sub-slices.

We present the properties that operators over the state types must guarantee:

- $\gamma[lval \mapsto \tau] = \gamma'$ indicates updating the type of *lval*, which can be a part of a variable, in state type γ. This operator guarantees that $\gamma' \vdash lval : \tau$ and for every lvalue $lval' \divideontimes lval$ such that $\gamma \vdash lval' : \tau'$ and $\gamma' \vdash lval' : \tau''$ then $\tau' \leq \tau''$.
- $\gamma +\!\!+ \gamma'$ updates γ such that for every variable in the domain of γ', the type of that variable in γ is updated to match γ'.
- $\text{refine}(\gamma, e) = \gamma'$ returns an overapproximation of states that satisfy the abstraction of γ and the predicate e. It guarantees that if $\gamma \vdash m$ and e evaluates to *true* in m then $\gamma' \vdash m$ and for every *lval*, if $\gamma \vdash lval : \tau$ and $\gamma' \vdash lval : \tau'$ then $\text{lbl}(\tau) \sqsubseteq \text{lbl}(\tau')$

- $\text{join}(\gamma_1, \gamma_2) = \gamma_3$ returns an overapproximation of γ_1, whose labels are at least as restrictive as γ_1 *and* γ_2. This operator guarantees that if $\gamma_1 \vdash m$ then $\gamma_3 \vdash m$ and for every *lval*, then $\text{lbl}(\gamma_1(\mathit{lval})) \sqcup \text{lbl}(\gamma_2(\mathit{lval})) \sqsubseteq \text{lbl}(\gamma_3(\mathit{lval}))$.

Appendix E Proofs and Guarantees

We use $T \vdash E$ to represent the abstraction and labeling soundness guarantees for externs and tables, and in the following we assume that this condition holds. Note that the proofs of this section use the *full* typing rules presented in Figure E.8.

Sufficient Condition Proof

Lemma E.3 (Sufficient condition)

If for every $\gamma_1, \gamma_2 \in \Gamma$ and every $\gamma_o \in \Gamma_o$ such that $\gamma_1 \cap \gamma_o \neq \bullet$, it holds that:

(1) if $\gamma_2 \cap \gamma_o \neq \bullet$, then $\gamma_1 \sqcup \gamma_2 \sqsubseteq \gamma_o$,

(2) for every lval either $\gamma_2(\mathit{lval}) \subseteq \gamma_o(\mathit{lval})$ or $\gamma_1 \sqcup \gamma_2(\mathit{lval}) = L$ hold,

then for every $\gamma_1, \gamma_2 \in \Gamma$ such that $m_1 \underset{\gamma_1}{\overset{\gamma_2}{\sim}} m_2$, and every $\gamma_o \in \Gamma_o$, it holds that if $\gamma_o \vdash m_1$ then $\gamma_o \vdash m_2$ and $m_1 \underset{\gamma_o}{\sim} m_2$.

Proof. First, we prove $\gamma_o \vdash m_2$. By definition, it is sufficient to prove that for every *lval*, $m_2(\mathit{lval}) : \gamma_o(\mathit{lval})$.
From the definition of $m_1 \underset{\gamma_1}{\overset{\gamma_2}{\sim}} m_2$ we know that $\gamma_1 \vdash m_1$ and $\gamma_2 \vdash m_2$ hold. Since $\gamma_o \vdash m_1$ and $\gamma_1 \vdash m_1$, then trivially $\gamma_1 \cap \gamma_o \neq \bullet$ holds. From the second hypothesis of the sufficient condition, two cases are possible.

1. $\gamma_2(\mathit{lval}) \subseteq \gamma_o(\mathit{lval})$. Since $\gamma_2 \vdash m_2$ hold, then by definition $m_2(\mathit{lval}) : \gamma_2(\mathit{lval})$ holds, therefore trivially $m_2(\mathit{lval}) : \gamma_o(\mathit{lval})$ holds.

2. $\gamma_1 \sqcup \gamma_2(\mathit{lval}) = L$. Since $m_1 \underset{\gamma_1}{\overset{\gamma_2}{\sim}} m_2$ then, indeed $m_1(\mathit{lval}) = m_2(\mathit{lval})$ holds. By definition of $\gamma_o \vdash m_1$, we know that $m_1(\mathit{lval}) : \gamma_o(\mathit{lval})$ also holds. Therefore, we can trivially show that $m_2(\mathit{lval}) : \gamma_o(\mathit{lval})$.

Second, we prove $m_1 \underset{\gamma_o}{\sim} m_2$. Previously, we showed that $\gamma_o \vdash m_2$ holds, and since $\gamma_2 \vdash m_2$, then trivially $\gamma_2 \cap \gamma_o \neq \bullet$ holds. From the first hypothesis of the sufficient condition, we can show that $\gamma_1 \sqcup \gamma_2 \sqsubseteq \gamma_o$. By definition of $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\sim}} m_2$, we know that $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$ holds. Therefore, trivially $m_1 \underset{\gamma_o}{\sim} m_2$. □

Hypothesis for Refinement

Hypothesis E.1 (Interval typedness - boolean expressions' refinement)

$$
\begin{aligned}
&\gamma \vdash e : \tau \;\wedge\; \gamma \vdash m \implies \\
&\qquad \Big(m(e) = \mathit{true} \implies \\
&\qquad\qquad (\mathrm{refine}(\gamma, e)) \vdash m \;\wedge\; m(e) = \mathit{false} \implies \\
&\qquad\qquad (\mathrm{refine}(\gamma, \neg e)) \vdash m\Big)
\end{aligned}
$$

Hypothesis E.2 (Interval typedness - select expressions' refinement)

$$
\begin{aligned}
&\gamma \vdash e : \tau \;\wedge\; m(e) = v \;\wedge\; \gamma \vdash m \\
&\quad \wedge\, i = \min\{i.\; v = v_i \vee i = n + 1\} \implies \\
&\qquad \gamma_1, ..., \gamma_{\mathrm{n}}, \gamma_{\mathrm{n}+1} = (\mathrm{refine}(\gamma, e = v_i)) \implies \gamma_i \vdash m
\end{aligned}
$$

Hypothesis E.3 (Interval typedness - externs and tables refinement)

$$
\gamma \vdash m \;\wedge\; \phi(m) \implies \mathrm{refine}(\gamma, \phi) \vdash m
$$

Hypothesis E.4 (Label typedness - boolean expressions' refinement)

$$
\begin{aligned}
&\qquad\qquad \gamma \vdash e : \tau_1 \;\wedge\; \gamma \vdash e : \tau_2 \\
&\qquad \wedge\, \mathrm{lbl}(\tau_1) = L \;\wedge\; \mathrm{lbl}(\tau_2) = L \\
&\qquad\qquad \wedge\; m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2 \implies \\
&\Big((\mathrm{refine}(\gamma_1, e) = \gamma_1' \;\wedge\; \mathrm{refine}(\gamma_2, e) = \gamma_2' \\
&\quad \wedge\, m_1(e) = \mathit{true} \;\wedge\; m_2(e) = \mathit{true} \implies m_1 \underset{\gamma 1' \sqcup \gamma 2'}{\sim} m_2) \\
&\wedge\; \big(\mathrm{refine}(\gamma_1, \neg e) = \gamma_1' \;\wedge\; \mathrm{refine}(\gamma_2, \neg e) = \gamma_2' \\
&\quad \wedge\, m_1(\neg e) = \mathit{true} \;\wedge\; m_2(\neg e) = \mathit{true} \implies \\
&\qquad\qquad m_1 \underset{\gamma 1' \sqcup \gamma 2'}{\sim} m_2)\Big)
\end{aligned}
$$

Hypothesis E.5 (Label typedness - select expressions' refinement)

$$\gamma \vdash e : \tau \;\wedge\; \text{lbl}(\tau) = L \;\wedge\; m_1 \underset{\gamma}{\sim} m_2 \implies$$
$$\Big(\gamma_1, ..., \gamma_{\text{n}}, \gamma_{\text{n+1}} = \text{refine}(\gamma, e = v_i) \;\wedge\; m_1(e) = v \implies$$
$$\forall j \leq n+1.\; m_1 \underset{\gamma_j}{\sim} m_2\Big)$$

Lemmas

Lemma E.4 (Expression reduction preserves the type)

$$\gamma \vdash m \;\wedge\; \gamma \vdash e : \tau \;\wedge\; m(e) = v \implies v : \tau$$

Lemma E.5 (lvalue updates preserves the type)

$$\gamma \vdash m \;\wedge\; v : \tau \implies \gamma[\mathit{lval} \mapsto \tau] \vdash m[\mathit{lval} \mapsto v]$$

Lemma E.6 (Expressions have types same as their values)

$$\gamma \vdash m \;\wedge\; \gamma \vdash e : \tau \implies \exists v.\; m(e) = v \;\wedge\; v : \tau$$

Lemma E.7 (Join does not modify the intervals)

$$\gamma \vdash m \;\wedge\; \gamma \in \Gamma_1 \implies$$
$$\forall \Gamma_2 ... \Gamma_n.\; \exists \gamma' \in \mathit{join}(\Gamma_1 \cup \Gamma_2 \cup ... \cup \Gamma_n).$$
$$\gamma' \vdash m \;\wedge\; \gamma \sqsubseteq \gamma'$$

Lemma E.8 (Expression evaluation of consistent states)

$$m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2 \wedge \gamma_1 \vdash m_1 \;\wedge\; \gamma_2 \vdash m_2 \;\wedge$$
$$\gamma_1 \vdash e : \tau_1 \wedge \gamma_2 \vdash e : \tau_2 \wedge$$
$$\text{lbl}(\tau_1) = L \;\wedge\; \text{lbl}(\tau_2) = L \implies$$
$$(m_1(e) = v \;\wedge\; m_2(e) = v)$$

Lemma E.9 (State equivalence preservation)

$$\gamma' \sqsubseteq \gamma \;\wedge\; m_1 \underset{\gamma'}{\sim} m_2 \quad\implies\quad m_1 \underset{\gamma}{\sim} m_2$$

Lemma E.10 (Branch on high - state preservation)

$$\begin{aligned}
&E : m \xrightarrow{s} m' \;\wedge\; T, H, \gamma \vdash s : \Gamma \;\wedge\; \gamma' \in \Gamma \implies \\
&\qquad (\gamma'(\mathit{lval}) = \tau \;\wedge\; \mathrm{lbl}(\tau) = L) \implies \\
&\qquad\qquad m(\mathit{lval}) = m'(\mathit{lval})
\end{aligned}$$

Lemma E.11 (Join's low label implication)

$$\begin{aligned}
&\Gamma = \mathit{join}(\Gamma_1 \cup ... \cup \Gamma_n) \;\wedge\; \gamma \in \Gamma \;\wedge \\
&\quad \gamma(\mathit{lval}) = \tau \;\wedge\; \mathrm{lbl}(\tau) = L \implies \\
&\qquad \forall \gamma' \in \Gamma_1, ..., \Gamma_n.\ \gamma'(\mathit{lval}) = \tau' \;\wedge\; \mathrm{lbl}(\tau') = L
\end{aligned}$$

Lemma E.12 (High program's final types)

$$T, H, \gamma \vdash s : \Gamma \quad\implies\quad \forall \gamma' \in \Gamma.\ \gamma \sqsubseteq \gamma'$$

Lemma E.13 (Branch on high - state lemma)

$$\begin{aligned}
&T, H, \gamma \vdash s : \Gamma \;\wedge\; \gamma' \in \Gamma \;\wedge \\
&\quad \gamma'(\mathit{lval}) = \tau' \;\wedge\; \mathrm{lbl}(\tau') = L \implies \\
&\qquad \forall \gamma'' \gamma'''.\ T, pc, \gamma'' \vdash s : \Gamma' \;\wedge\; \gamma''' \in \Gamma' \implies \\
&\qquad\qquad \Big(\gamma''(\mathit{lval}) = \tau'' \;\wedge\; \gamma'''(\mathit{lval}) = \tau''' \\
&\qquad\qquad\quad \implies \mathrm{lbl}(\tau'') \sqsubseteq \mathrm{lbl}(\tau''')\Big)
\end{aligned}$$

Lemma E.14 (Branch on high is never empty)

$$T, H, \gamma \vdash s : \Gamma \implies \Gamma \neq \varnothing$$

Lemma E.15 (Low equivalence distribution)

$$(m_1, m_1') \underset{(\gamma_a,\gamma_a')\sqcup(\gamma_b,\gamma_b')}{\sim} (m_2, m_2') \Leftrightarrow$$
$$(m_1 \underset{\gamma_a\sqcup\gamma_b}{\sim} m_2 \ \wedge\ m_1' \underset{\gamma_a'\sqcup\gamma_b'}{\sim} m_2')$$

Lemma E.16 (Low equivalence update)

$$m_1 \underset{\gamma_a\sqcup\gamma_b}{\sim} m_2 \ \wedge\ m_3 \underset{\gamma_c\sqcup\gamma_d}{\sim} m_4 \implies$$
$$m_1[\mathit{lval} \mapsto m_3(e)] \underset{\gamma_a[\mathit{lval}\mapsto\gamma_c(e)]\sqcup\gamma_b[\mathit{lval}\mapsto\gamma_d(e)]}{\sim} m_2[\mathit{lval} \mapsto m_4(e)]$$

Soundness of Abstraction

Theorem E.2

$$\forall\, s\, T\, m\, \gamma\, pc\, \Gamma.\ T, pc, \gamma \vdash s : \Gamma \implies$$
$$T \vdash E \ \wedge\ \gamma \vdash m \implies$$
$$\exists m'.E : m \xrightarrow{s} m' \ \wedge\ \exists\gamma' \in \Gamma.\ \gamma' \vdash m'$$

Proof. In this proof we assume that the program is well typed and does not get stuck according to HOL4P4 type system. By induction on the typing tree of the program *stmt*, i.e. s. Note that, initially $\gamma = (\gamma_g, \gamma_m)$ and $m = (m_g, m_l)$. In the following proof, we know that $\gamma \vdash m$ holds in all subcases.

◇ **Case assignment**: Here *stmt* is *lval* := e. From assignment typing rule we know:

1. $\gamma \vdash e : \tau$
2. $\tau' = \text{raise}(\tau, pc)$
3. $\gamma' = \gamma[\mathit{lval} \mapsto \tau']$
4. $\Gamma = \{\gamma'\}$

We need to prove $\exists m'.E : m \xrightarrow{\mathit{lval}:=e} m'$ and $\exists\gamma'' \in \Gamma.\gamma'' \vdash m'$. From the assignment reduction definition, we can rewrite the goal's conjunctions to:

1. $m(e) = v$ (from assignment reduction)

2. $\exists m'.\ m' = m[lval \mapsto v]$ (from assignment reduction)
3. $\exists \gamma'' \in \{\gamma'\}.\gamma'' \vdash m'$

Goal 1: we know that the initial variable map is typed using the state type, i.e. $\gamma \vdash m$ from assumptions. Using $\gamma \vdash m$ and 1 from typing rule, then we can use Lemma E.6 to directly infer that the expression's reduction indeed keeps the type, i.e. $\exists v.\ m(e) = v \ \wedge \ v : \tau$ and this resolves goal 1.

Goal 2: trivial, as we can instantiate m' to be $m[lval \mapsto v]$.

Goal 3: we know that assignment typing rule produces a singleton set from 4 of typing rule, thus we can instantiate γ'' to be $\gamma[lval \mapsto \tau']$, thus allows us to rewrite the goal to $\gamma[lval \mapsto \tau'] \vdash m[lval \mapsto v]$.

Given $\gamma \vdash m$, this entails (by definition) that γ and m they contain the same variable name in the domain, and also the values are well-typed, i.e. $domain(\gamma) = domain(m) \ \wedge \ \forall x \in domain(m).\ m(x) : \gamma(x)$.

Using Lemma E.4, we know that the assigned value v can be typed with τ, i.e. $v : \tau$. The assignment typing rule raises the labels using the function raise, however by definition we know that the abstraction is unaffected, so the interval of τ' and τ are the same, thus we can infer that $v : \tau'$.

Using Lemma E.5, we can see that the update preserves the type, thus goal proved.

◇ **Case condition**: Here *stmt* is `if` e `then` s_1 `else` s_2. From conditional typing rule we know:

1. $\gamma \vdash e : \tau$
2. $\ell = \mathrm{lbl}(\tau)$
3. $pc' = pc \sqcup \ell$
4. $T, pc', (\mathrm{refine}(\gamma, e)) \vdash s_1 : \Gamma_1$
5. $T, pc', (\mathrm{refine}(\gamma, \neg e)) \vdash s_2 : \Gamma_2$
6. $\Gamma' = \Gamma_1 \cup \Gamma_2$
7. $\Gamma'' = \begin{cases} \mathrm{join}(\Gamma') & \text{if } \ell = H \\ \Gamma' & \text{otherwise} \end{cases}$

We need to prove $\exists m'.E : m \xrightarrow{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} m'$ and $\exists \gamma'' \in \Gamma''.\gamma'' \vdash m'$.

In the goal, the boolean guard e evaluates to *true* or *false*. So we will get two goal cases with similar proofs. This only solves for e, while case $\neg e$ follows the same proof strategy. We can rewrite the goal to:

1. $\exists m'.\ E : m \xrightarrow{s_1} m'$

2. $\exists \gamma'' \in \Gamma''.\gamma'' \vdash m'$

In this proof, we will get two induction hypotheses for statement: for s_1 call it IH1 and for s_2 call it IH2.

IH1 is the following (note that IH2 is the same, but instantiated for s_2):

$$\begin{aligned} \forall T\, m\, \gamma\, pc\, \Gamma.\ T, pc, \gamma \vdash s_1 : \Gamma &\implies \\ T \vdash E\ \wedge\ \gamma \vdash m &\implies \\ \exists m'.\ E : m \xrightarrow{s_1} m'\ \wedge\ \exists \gamma' \in \Gamma.\ \gamma' \vdash m' \end{aligned}$$

We first prove that the set of refined state types using e can still type the staring concrete memory m. This we can show, because we know initially $\gamma \vdash m$, and we know that e is typed as a boolean (assumed to be well-typed), and we know that e reduces to *true* from the reduction rule, these allow us to infer $(\text{refine}(\gamma, e)) \vdash m$ using Hyp E.1.

Now we instantiate the induction hypothesis IH1 using $(T, m, \text{refine}(\gamma, e), pc', \Gamma_1)$, to show that exists m' such that $E : m \xrightarrow{s_1} m'$, i.e. there indeed exists a transition to a final configuration in the semantics to m' (which resolves goal 1). Additionally, we can show from IH1 that exists $\gamma' \in \Gamma_1$ such that $\gamma' \vdash m'$.

Now we implement cases on the expression's label ℓ being H or L:

case $\text{lbl}(\tau) = H$: we need to prove $\exists \gamma'' \in \text{join}\ \{\Gamma_1 \cup \Gamma_2\}.\gamma'' \vdash m'$. Then it is easy to deduct that the goal holds; because we showed that there is a state type $\gamma' \in \Gamma_1$ such that it is a sound abstraction of the final state $\gamma' \vdash m'$, and we know that join operation does not change the abstraction, it just modifies the security label. Hence, indeed there exists a state type γ'' in join $\{\Gamma_1 \cup \Gamma_2\}$ such that it is also a sound abstraction of final state $\gamma'' \vdash m'$.

case $\text{lbl}(\tau) = L$: we need to prove $\exists \gamma'' \in \{\Gamma_1 \cup \Gamma_2\}.\gamma'' \vdash m'$, which is trivially true.

For the negation case, use IH2 and follow the same steps.

◇ **Case sequence**: Here *stmt* is $s_1; s_2$. From sequence typing rule we know:

1. $T, pc, \gamma \vdash s_1 : \Gamma_1$

2. $\forall \gamma_1 \in \Gamma_1.\ T, pc, \gamma_1 \vdash s_2 : \Gamma_2^{\gamma_1}$

3. $\Gamma' = \bigcup_{\gamma_1 \in \Gamma_1} \Gamma_2^{\gamma_1}$

In this proof, we will get two induction hypotheses for statement: for s_1 call it IH1 and for s_2 call it IH2.
IH1 is (note that IH2 is the same, but instantiated for s_2):

$$\begin{aligned} \forall T m \gamma\, pc\, \Gamma.\ T, pc, \gamma \vdash s_1 : \Gamma \implies \\ T \vdash E \ \wedge\ \gamma \vdash m \implies \\ \exists m'.E : m \xrightarrow{s_1} m' \ \wedge\ \exists \gamma' \in \Gamma.\gamma' \vdash m' \end{aligned}$$

We need to prove $\exists m''.\ E : m \xrightarrow{s_1;s_2} m''$ and $\exists \gamma'' \in \Gamma'.\ \gamma'' \vdash m''$.

We can rewrite the goal using the definition of sequence case to:

1. $E : m \xrightarrow{s_1} m'$
2. $\exists m''.E : m' \xrightarrow{s_2} m''$
3. $\exists \gamma'' \in \Gamma'.\gamma'' \vdash m''$

Goal 1: We can instantiate the IH1 with $(T, m, \gamma, pc, \Gamma_1)$ to infer that exists m_1' such that $E : m \xrightarrow{s_1} m_1'$, and also exists $\gamma' \in \Gamma_1$ such that $\gamma' \vdash m_1'$. Since the semantics are deterministic, m_1' and m' are equivalent, thus it holds $E : m \xrightarrow{s_1} m'$ and $\gamma' \vdash m'$.

Goal 2 and 3 : We showed from IH1 that $\gamma' \vdash m'$, now we can instantiate 2 from sequence typing rule with γ'. Now we can to instantiate IH2 with $(T, m', \gamma', pc, \Gamma_2^{\gamma'})$, and infer that exists m_2' such that $E : m' \xrightarrow{s_2} m_2'$, and also exists $\gamma''' \in \Gamma_2^{\gamma'}$ such that $\gamma''' \vdash m_2'$. Since the semantics are deterministic, m_2' and m'' are equivalent, thus it holds $E : m \xrightarrow{s_2} m''$ and $\gamma''' \vdash m'$.

From 3 in sequence typing rule, we know that Γ' is the union of all resulted state type sets, such that it can type s_2, since we know that $\gamma''' \in \Gamma_2^{\gamma'}$ will be in Γ', then we prove the goal $\exists \gamma'' \in \Gamma'.\gamma'' \vdash m''$.

$\diamond$ **Case function call**: Here *stmt* is $f(e_1, ..., e_n)$. From call typing rule we know (note that here we explicitly write the global and local state type):

1. $(\gamma_g, \gamma_l) \vdash e_i : \tau_i$
2. $(s, (x_1, d_1), ..., (x_n, d_n)) = (C, F)(f)$
3. $\gamma_f = \{x_i \mapsto \tau_i\}$
4. $(C, F), pc, (\gamma_g, \gamma_f) \vdash s : \Gamma'$
5. $\Gamma'' = \{(\gamma_g', \gamma_l)[e_i \mapsto \gamma_f'(x_i) \mid \text{isOut}(d_i)] \mid (\gamma_g', \gamma_f') \in \Gamma'\}$

In the following proof, we know that $(\gamma_g, \gamma_l) \vdash (m_g, m_l)$ and $(C, F) \vdash (X, F)$ hold. Additionally, we get an induction hypothesis IH for the body of the function s.

$$\begin{aligned}&\forall (C,F)\ m \gamma\, pc\, \Gamma.\ (C,F), pc, \gamma \vdash s : \Gamma \implies \\ &\quad (C,F) \vdash (X,F)\ \wedge\ \gamma \vdash m \implies \\ &\qquad \exists m'.(X,F) : m \xrightarrow{s} m'\ \wedge\ \exists \gamma' \in \Gamma. \gamma' \vdash m'\end{aligned}$$

We need to prove $\exists m'.\ (X, F) : (m_g, m_l) \xrightarrow{f(e_1,\ldots,e_n)} m'\ \wedge\ \exists \gamma' \in \Gamma''.\ \gamma' \vdash m'$

From call reduction rule we can rewrite the goal to:

1. $\exists s\ (x_1, d_1), ..., (x_n, d_n).\ \ (s, (x_1, d_1), ..., (x_n, d_n)) = (X, F)(f)$
2. $\exists m_f.\ m_f = \{x_i \mapsto (m_g, m_l)(e_i)\}$
3. $\exists (m'_g, m'_f).\ (X, F) : (m_g, m_f) \xrightarrow{s} (m'_g, m'_f)$
4. $\exists m''.m'' = (m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$
5. $\exists \gamma' \in \Gamma''.\gamma' \vdash m''$

Goal 1: From $(C, F) \vdash (X, F)$ we know indeed the function's body and signature found in the semantics is the same found in the typing rule.

Goal 2: Trivial, the existence can be instantiated with $\{x_i \mapsto (m_g, m_l)(e_i)\}$.

Goal 3: To prove that there exists a state where the body of the function reduces to, we need to use the IH. Thus, we first need to show that the resulted copy-in map is also well-typed i.e. $\gamma_f \vdash m_f$, more specifically $\forall i.\ \{x_i \mapsto \tau_i\} \vdash \{x_i \mapsto (m_g, m_l)(e_i)\}$. Given that initially the state type can type the state $(\gamma_g, \gamma_l) \vdash (m_g, m_l)$ from assumptions and given that the expressions e_i have a type τ_i from typing rule 1, now we can use Lemma E.4 to infer that $\forall v_i : \tau_i$ such that v_i is the evaluation of $(m_g, m_l)(e_i)$. This leads us to trivially infer that $\forall i.\ \{x_i \mapsto \tau_i\} \vdash \{x_i \mapsto v_i\}$ holds.

Now we can use the IH by instantiating it to $((C, F), (m_g, m_f), (\gamma_g, \gamma_f), pc, \Gamma')$ in order to infer that exists (m'_g, m'_f) such that $(X, F) : (m_g, m_f) \xrightarrow{s} (m'_g, m'_f)$ and exists $(\gamma''_g, \gamma''_f) \in \Gamma'$ such that $(\gamma''_g, \gamma''_f) \vdash (m'_g, m'_f)$.

Goal 4: Trivial, we can instantiate the existence by $(m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$.

Goal 5: The goal is to prove copy-out operation to be well-typed, thus we can rewrite the goal to $\exists \gamma' \in \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\}$ such that $\gamma' \vdash (m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$.

We can choose γ' to be $(\gamma''_g, \gamma_l)[e_i \mapsto \gamma''_f(x_i) \mid \text{isOut}(d_i)]$.

Since we are able to choose a state type that types the final state, we can rewrite the goal to prove again to be $(\gamma''_g, \gamma_l)[e_i \mapsto \gamma''_f(x_i) \mid \text{isOut}(d_i)] \vdash (m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$.

First, we know that $\gamma_l \vdash m_l$ holds from the assumptions. Also, we showed in (Goal 3) that $(\gamma''_g, \gamma''_f) \vdash (m'_g, m'_f)$, thus trivially $\gamma''_g \vdash m'_g$ and $\gamma''_f \vdash m'_f$ hold. Thus, we can deduct that $(\gamma''_g, \gamma_l) \vdash (m''_g, m_l)$, and $m'_f(x_i) : \gamma''_f(x_i)$, and then we can use Lemma E.5 to deduct that the update preserves the well-typedness, i.e. $(\gamma''_g, \gamma_l)[e_i \mapsto \gamma''_f(x_i)] \vdash (m'_g, m_l)[e_i \mapsto m'_f(x_i)]$, which proves the goal.

$\diamond$ **Case extern**: Here *stmt* is $f(e_1, ..., e_n)$. From extern typing rule we know (note that here we explicitly write the global and local state type):

1. $(\gamma_g, \gamma_l) \vdash e_i : \tau_i$
2. $(\text{Cont}_\text{E}, (x_1, d_1), ..., (x_n, d_n)) = (C, F)(f)$
3. $\gamma_f = \{x_i \mapsto \tau_i\}$
4. $\forall (\gamma_i, \phi, \gamma_t) \in \text{Cont}_\text{E}.\ (\gamma_g, \gamma_l) \sqsubseteq \gamma_i$
5. $\Gamma' = \{\gamma' \mathbin{+\!\!+} \text{raise}(\gamma_t, pc) \mid (\gamma_i, \phi, \gamma_t) \in \text{Cont}_\text{E} \ \wedge\ \text{refine}((\gamma_g, \gamma_f), \phi) = \gamma' \neq \bullet\}$
6. $\Gamma'' = \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\}$

In the following proof, we know that $(\gamma_g, \gamma_l) \vdash (m_g, m_l)$ and $(C, F) \vdash (X, F)$ hold.

We need to prove $\exists m'.\ (X, F) : (m_g, m_l) \xrightarrow{f(e_1,\ldots,e_n)} m' \ \wedge\ \exists \gamma'' \in \Gamma''.\ \gamma'' \vdash m'$

From extern reduction rule we can rewrite the goal to:

1. $\exists (sem_f, (x_1, d_1), ..., (x_n, d_n)).\ (sem_f, (x_1, d_1), ..., (x_n, d_n)) = (X, F)(f)$
2. $\exists m_f.\ m_f = \{x_i \mapsto (m_g, m_l)(e_i)\}$
3. $\exists (m'_g, m'_f).\ (m'_g, m'_f) = sem_f(m_g, m_f)$
4. $\exists m''.m'' = (m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$
5. $\exists \gamma'' \in \Gamma''.\gamma'' \vdash m''$

Goal 1: From the environment's well-typedness $(C, F) \vdash (X, F)$, we know domain(C) $\cap$ domain$(F) = \emptyset$ and extWT C X holds. This mean that indeed the extern is defined only in C. Additionally, from well-typedness $(C, F) \vdash (X, F)$, that if $C(f) = (sem_f, \overline{(x, d)})$ then $X(f) = (\text{Cont}_E, \overline{(x, d)})$, thus indeed exist Cont_E and signature $(x_1, d_1), ..., (x_n, d_n)$.

Goal 2: Trivial, by instantiating m_f to be $\{x_i \mapsto (m_g, m_l)(e_i)\}$.

We can here also prove that the resulted copy-in map is also well-typed i.e. $\gamma_f \vdash m_f$, more specifically $\forall i.\ \{x_i \mapsto \tau_i\} \vdash \{x_i \mapsto (m_g, m_l)(e_i)\}$. Given that initially the typing state types the state $(\gamma_g, \gamma_l) \vdash (m_g, m_l)$ from assumptions and given that the expressions e_i have a type τ_i from typing rule 1, now we can use Lemma E.4 to infer that $\forall v_i : \tau_i$ such that v_i is the evaluation of $(m_g, m_l)(e_i)$. This leads us to trivially infer that $\forall i.\ \{x_i \mapsto \tau_i\} \vdash \{x_i \mapsto v_i\}$ holds.

Goal 3: From $(C, F) \vdash (X, F)$, we know that extWT $C\ X$ holds, and from its definition, we know that indeed exists $(\gamma_i, \phi, \gamma_t)$ such that $\phi(m_g, m_f)$, this means that indeed there exists a contract's predicate satisfied by the values in the initial concrete input state, i.e. $\phi(m_g, m_f)$. This implies, from the definition of extWT $C\ X$, that indeed exists (m'_g, m'_f) such that $(m'_g, m'_f) = sem_f(m_g, m_f)$.

Goal 4: Trivial, by instantiating m'' to $(m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$.

Goal 5: We can rewrite the goal to exists $\gamma'' \in \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\}$ such that $\gamma'' \vdash (m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$.

We can prove this goal by first find a $\gamma_A \in \Gamma'$ such that it types the final states of the extern's semantic (m'_g, m'_f). Second, we find the $\gamma'' \in \Gamma''$ such that it can type the final state m'' after copying out the extern.

We previously established $\gamma_f \vdash m_f$ (from goal 2), also given that $\gamma_g \vdash m_g$ from assumptions we can trivially $(\gamma_g, \gamma_f) \vdash (m_g, m_f)$. Since we previously showed that $\phi(m_g, m_f)$ (from goal 3), now we can use Hyp E.3 in order to infer that the refined state $\text{refine}((\gamma_g, \gamma_f), \phi)$ can also type (m_g, m_f) also we infer that it is not empty, i.e. $\text{refine}((\gamma_g, \gamma_f), \phi) \vdash (m_g, m_f)$ and $\text{refine}((\gamma_g, \gamma_f), \phi) \neq \bullet$.

The definition of extWT $C\ X$ states that $\gamma_t \vdash (m'_g, m'_f)$ holds for the set of variables that the extern's semantics has changed i.e. $\{x.\ (m_g, m_f)(x) \neq (m'_g, m'_f)(x)\} \subseteq domain(\gamma_t)$.

Consequently, we can further prove that indeed exists a γ_A in Γ' (from 4 in typing rule of extern) that can type the output of the extern's semantics (m'_g, m'_f) including the unchanged variables. Thus, we can make cases on extern's semantics input and output as following:

case $(m_g, m_f)(x) = (m'_g, m'_f)(x)$: This means that variable x not in the domain of γ_t, thus it is unchanged, therefore it is typed by the refined state $(m'_g, m'_f)(x) : (\text{refine}((\gamma_g, \gamma_f), \phi))(x)$. Trivially, we can also infer that

$$(m'_g, m'_f)(x) : (\text{refine}((\gamma_g, \gamma_f), \phi) +\!\!+ \text{raise}(\gamma_t, pc))(x)$$

.

case $(m_g, m_f)(x) \neq (m'_g, m'_f)(x)$: This means that variable x is in the domain of γ_t, thus it is changed, and the new type of it is in γ_t. Therefore, we can trivially conclude that $(m_g, m_f)(x) : \gamma_t(x)$, Consequently, since we know that raise does not change the abstraction, and just change labels, we can therefore conclude that $(m'_g, m'_f)(x) : (\text{refine}((\gamma_g, \gamma_f), \phi) +\!\!+ \text{raise}(\gamma_t, pc))(x)$

These cases show that we can select γ_A such that $\gamma_A \in \Gamma'$ to be $(\text{refine}((\gamma_g, \gamma_f), \phi) +\!\!+ \text{raise}(\gamma_t, pc))$, because it can indeed type (m'_g, m'_f). i.e. $(\text{refine}((\gamma_g, \gamma_f), \phi) +\!\!+ \text{raise}(\gamma_t, pc)) \vdash (m'_g, m'_f)$. For simplicity in the rest of the proof, let us rewrite $\gamma_A = (\gamma_{A_g}, \gamma_{A_f})$, where γ_{A_g} is the global part of the pair $(\text{refine}((\gamma_g, \gamma_f), \phi) +\!\!+ \text{raise}(\gamma_t, pc))$ and γ_{A_f} is the local part of the pair. Thus, we can say that $\gamma_{A_g} \vdash m'_g$ and $\gamma_{A_f} \vdash m'_f$.

Now we need to find $\gamma'' \in \Gamma''$ such that it types $(m'_g, m_l)[e_i \mapsto m'_f(x_i) \mid \text{isOut}(d_i)]$ order to prove Goal 5.

Given from assumptions $\gamma_l \vdash m_l$, and we showed that $\gamma_{A_g} \vdash m'_g$ and $\gamma_{A_f} \vdash m'_f$. In 5 of the typing rules, we pick (γ'_g, γ'_f) such that it is in Γ' to be $(\gamma_{A_g}, \gamma_{A_f})$, Now we conduct cases on the direction of the parameter $\text{isOut}(d_i)$ being `out` or not.

case $\neg\text{isOut}(d_i)$: Then (γ_{A_g}, γ_l) are unchanged, similarly in the semantics (m'_g, m_l) are also unchanged. Thus they can still be typed as $(\gamma_{A_g}, \gamma_l) \vdash (m'_g, m_l)$

case $\text{isOut}(d_i)$: Then (γ_{A_g}, γ_l) are updated with $e_i \mapsto \gamma_{A_f}(x)$, similarly the semantics state (m'_g, m_l) is updated with $e_i \mapsto m'_f(x)$. Previously we showed that $\gamma_{A_f} \vdash m'_f$, this entails by the definition of state typedness $m'_f(x) : \gamma_{A_f}(x)$. This leads to the point that the modifications of e_i's type in state type (γ_{A_g}, γ_l) and value in state (m'_g, m_l) keeps them well typed. Therefore, $(\gamma_{A_g}, \gamma_l) \vdash (m'_g, m_l)$ holds.

We can finally conclude that goal 5 can be resolved by picking the γ'' to be (γ_{A_g}, γ_l), the goal is now proven.

⋄ **Case table application**: Here *stmt* is `apply` *tbl*. From table typing rule we know:

1. $(\overline{e}, \text{Cont}_{\text{tbl}}) = (C, F)(\mathit{tbl})$
2. $\gamma \vdash e_i : \tau_i$
3. $\ell = \bigsqcup_i \text{lbl}(\tau_i)$
4. $pc' = pc \sqcup \ell$
5. $\forall(\phi_j, (a_j, \overline{\tau}_j)) \in \text{Cont}_{\text{tbl}}.\ (\gamma_{g_j}, \gamma_{l_j}) = \text{refine}(\gamma, \phi_j) \ \wedge$ $(s_j, (x_{j_1}, \text{none}), ..., (x_{j_n}, \text{none})) = (C, F)(a_j) \ \wedge$ $\gamma_{a_j} = \{x_{j_i} \mapsto \tau_{j_i}\} \ \wedge \ T, pc', (\gamma_{g_j}, \gamma_{a_j}) \vdash s_j : \Gamma_j$

6. $\Gamma' = \cup_j\{(\gamma'_{g_j}, \gamma_{l_j})|(\gamma'_{g_j}, \gamma'_{a_j}) \in \Gamma_j\}$

7. $\Gamma'' = \begin{cases} \text{join}(\Gamma') & \text{if } \ell = H \\ \Gamma' & \text{otherwise} \end{cases}$

In the following proof, we know that $(\gamma_g, \gamma_l) \vdash (m_g, m_l)$ and $(C, F) \vdash (X, F)$ hold.

We need to prove $\exists m'.\ E : m \xrightarrow{\texttt{apply}\ tbl} m'$ and $\exists \gamma'' \in \Gamma''.\gamma'' \vdash m'$.

And from table reduction rule we can rewrite the goal to:

1. $\exists \overline{e}\ sem_{tbl}.\ (\overline{e}, sem_{tbl}) = (X, F)(tbl)$
2. $\exists (a, \overline{v}).\ sem_{tbl}((m_g, m_l)(e_1), ..., (m_g, m_l)(e_n)) = (a, \overline{v})$
3. $\exists s\ (x_1, ..., x_n).\ (s, (x_1, \text{none}), ..., (x_n, \text{none})) = E(a)$
4. $\exists m_a.\ m_a = \{x_i \mapsto v_i\}$
5. $\exists (m_{g'}, m_{a'}).\ E : (m_g, m_a) \xrightarrow{s} (m_{g'}, m_{a'})$
6. $\exists \gamma'' \in \Gamma''.\gamma'' \vdash (m_{g'}, m_l)$

In this proof, we will get an induction hypothesis for action call $a(\overline{v})$ (formalized as a function call), call it IH.

$$\begin{aligned} &\forall (C, F)\ m\ \gamma\ pc\ \Gamma.\ (C, F), pc, \gamma \vdash a(\overline{v}) : \Gamma \implies \\ &\qquad (C, F) \vdash (X, F)\ \wedge\ \gamma \vdash m \implies \\ &\qquad\quad \exists m'.(X, F) : m \xrightarrow{a(\overline{v})} m'\ \wedge\ \exists \gamma' \in \Gamma.\gamma' \vdash m' \end{aligned}$$

Goal 1: Trivial, by the well-typedness condition $(C, F) \vdash (X, F)$, we know that if the table has a contract (from 1 in typing rule), then indeed there is semantics for it sem_{tbl} and a key list $\overline{e}$ that matches the one in the table typing rule.

Goal 2: From $(C, F) \vdash (X, F)$, we can deduct from condition tblWT $C\ X$ that indeed exists an action and value list pair $(a, \overline{v})$ in the contract Cont_{tbl} correlated to a $\phi_j(m)$ that holds.

Goal 3: From $(C, F) \vdash (X, F)$ we know indeed the actions's a body and signature found in the semantics is the same found in the typing rule.

Goal 4: Trivial, by setting m_a to be $\{x_i \mapsto v_i\}$.

Goal 5: To prove this goal, we need to use IH. And in order to use IH, we must first show that $(\gamma_{g_j}, \gamma_{a_j}) \vdash (m_g, m_a)$ by proving $\gamma_{g_j} \vdash m_g$ where $(\gamma_{g_j}, \gamma_{l_j}) = \text{refine}(\gamma, \phi_j)$ and the copied in is well typed $\gamma_{a_j} \vdash m_a$.

Prove $\gamma_{g_j} \vdash m_g$: Since initially given that $\gamma \vdash m$ i.e. $(\gamma_g, \gamma_l) \vdash (m_g, m_l)$, also we know from tblWT $C\ X$ that there is j such that the predicate ϕ_j is satisfied in (m_g, m_l) i.e. $\phi_j(m_g, m_l)$, thus we can use Hyp E.3 to deduct that refine$(\gamma, \phi_j) \vdash (m_g, m_l)$, i.e. we can infer that the refined state type is able to type the initial state. Given 5 in the typing rule, we know that $(\gamma_{g_j}, \gamma_{l_j}) = \text{refine}(\gamma, \phi_j)$ thus $(\gamma_{g_j}, \gamma_{l_j}) \vdash (m_g, m_l)$, therefore $\gamma_{g_j} \vdash m_g$ holds.

Prove $\gamma_{a_j} \vdash m_a$: This goal can be rewritten as $\{x_i \mapsto \tau_i\} \vdash \{x_i \mapsto v_i\}$. The proof is trivial by WF definition of tables we know that $v_i : \tau_i$, thus the variable x_i is well typed.

Now, we can instantiate IH to using $((C, F), (m_g, m_a), (\gamma_{g_j}, \gamma_{a_j}), pc', \Gamma_j)$, so we can infer that exists m' such that $(X, F) : m \xrightarrow{s} m'$ (thus goal 5 is resolved).

Goal 6: We can also infer from IH that exists $\gamma' \in \Gamma_j$ such that $\gamma' \vdash m'$. Let $(\gamma'_{g_j}, \gamma'_{a_j}) = \gamma'$ and $(m'_g, m'_a) = m'$, thus indeed $(\gamma'_{g_j}, \gamma'_{a_j}) \vdash (m'_g, m'_a)$ holds trivially.

Line 6 of the typing rule iterates over each final state type set and collects the modified global state and the refined local state, thus $(\gamma'_{g_j}, \gamma_{l_j})$ is indeed in Γ'. Since we proved that $\gamma'_{g_j} \vdash m'_g$ holds in the previous step, and also proved $\gamma_{l_j} \vdash m_l$ in goal 5, therefore, $(\gamma'_{g_j}, \gamma_{l_j}) \vdash (m'_g, m_l)$.

Line 7 of the typing rule changes the labels but not the abstraction, thus the abstraction of the state type $(\gamma'_{g_j}, \gamma_{l_j})$ in Γ' indeed exists in Γ'' with labels changed so goal 6 holds.

□

Soundness of Labeling

Theorem E.3

$$
\begin{array}{c}
\forall s\ T\ pc\ m_1\ m_2\ m'_1\ m'_2\ \gamma_1\ \gamma_2\ \Gamma_1\ \Gamma_2\ E_1\ E_2. \\
T, pc, \gamma_1 \vdash s : \Gamma_1 \ \wedge\ T, pc, \gamma_2 \vdash s : \Gamma_2 \implies \\
T \vdash E_1 \ \wedge\ T \vdash E_2 \ \wedge\ E_1 \underset{T}{\sim} E_2 \ \wedge \\
\gamma_1 \vdash m_1 \ \wedge\ \gamma_2 \vdash m_2 \ \wedge\ m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2 \ \wedge \\
E_1 : m_1 \xrightarrow{s} m'_1 \ \wedge\ E_2 : m_2 \xrightarrow{s} m'_2 \implies \\
(\exists \gamma'_1 \in \Gamma_1 \ \wedge\ \gamma'_2 \in \Gamma_2.\ \gamma'_1 \vdash m'_1 \\
\wedge\ \gamma'_2 \vdash m'_2 \ \wedge\ m'_1 \underset{\gamma'_1 \sqcup \gamma'_2}{\sim} m'_2)
\end{array}
$$

Proof. In this proof we assume that the program is well typed and does not get stuck according to HOL4P4 type system. by induction on the typing tree of the program *stmt* i.e. s. Note that, initially $\gamma = (\gamma_g, \gamma_m)$ and $m = (m_g, m_l)$. In the following proof, we know that $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$, we also know that $\gamma_1 \vdash m_1$ and $\gamma_2 \vdash m_2$.

◇ **Case assignment**: Here *stmt* is *lval* $:= e$. From assignment typing rule we know:

1. $\gamma_1 \vdash e : \tau_1$
2. $\tau_1' = \text{raise}(\tau_1, pc)$
3. $\gamma_1' = \gamma_1[\mathit{lval} \mapsto \tau_1']$
4. $\Gamma_1 = \{\gamma_1'\}$
5. $\gamma_2 \vdash e : \tau_2$
6. $\tau_2' = \text{raise}(\tau_2, pc)$
7. $\gamma_2' = \gamma_2[\mathit{lval} \mapsto \tau_2']$
8. $\Gamma_2 = \{\gamma_2'\}$

And from assignment reduction rule we know:

1. $m_1(e) = v_1$
2. $m_1' = m_1[\mathit{lval} \mapsto v_1]$
3. $m_2(e) = v_2$
4. $m_2' = m_2[\mathit{lval} \mapsto v_2]$

Prove $\exists \gamma_1' \in \Gamma_1 \ \wedge \ \gamma_2' \in \Gamma_2.\ \gamma_1' \vdash m_1' \ \wedge \ \gamma_2' \vdash m_2' \ \wedge \ m_1' \underset{\gamma_1' \sqcup \gamma_2'}{\sim} m_2'$. Since that assignment typing rule produces one state type (from 3,4,7,and 8), then from Soundness of Abstraction, we can infer $\gamma_1' \vdash m_1'$ and $\gamma_2' \vdash m_2'$, therefore the first two conjunctions of the goal holds.

Now, the final remaining goal to prove is that $m_1' \underset{\gamma_1' \sqcup \gamma_2'}{\sim} m_2'$. This entails proving that all *lval*$'$ in both m_1' and m_2' are low equivalent with respect to the least upper bound of the final state types that types the final states.

Now we do cases on the label of *lval*$'$ being H or L in $\gamma_1' \sqcup \gamma_2'$.

case label of *lval*$'$ is H: If the label is H, i.e. $\gamma_1' \sqcup \gamma_2' \vdash \mathit{lval}' : \tau \ \wedge \ \text{lbl}(\tau) = H$, then the property of labeling soundness holds after the assignment trivially. That's because soundness property checks the equality of the state type's low ranges only.

case label of lval' is L: If the label is L, i.e. $\gamma_1' \sqcup \gamma_2' \vdash \mathit{lval}' : \tau \ \wedge \ \mathrm{lbl}(\tau) = L$, this implies that in each state type γ_1' and γ_2' individually, the typing label of lval' is L.

In this section, we conduct a case analysis on possible sub-cases relations between of lval being assigned and lval', thus we will have the following subcases: $\mathit{lval}' \subsetneqq \mathit{lval}$, $\mathit{lval} \subsetneqq \mathit{lval}'$, $\mathit{lval}' = \mathit{lval}$, $\mathit{lval}' \subseteqq \mathit{lval}$, and $\mathit{lval} \subseteqq \mathit{lval}'$.

case $\mathit{lval}' \subsetneqq \mathit{lval}$ and $\mathit{lval} \subsetneqq \mathit{lval}'$:
Now for all lval' that are not equal to the lval we assign to, or not sub-lval of it: our goal is to show $m_1'(\mathit{lval}') = m_2'(\mathit{lval}')$ by demonstrating that initially $m_1(\mathit{lval}') = m_2(\mathit{lval}')$ also holds. We know that the assignment doesn't alter those parts of the states. Thus, the semantic update should keep the values of lval' the same, so $m_1(\mathit{lval}') = m_1'(\mathit{lval}')$ and $m_2(\mathit{lval}') = m_2'(\mathit{lval}')$. Likewise, the typing update should keep the type of lval' unchanged, so $\gamma_1'(\mathit{lval}') = \gamma_1(\mathit{lval}')$ and $\gamma_2'(\mathit{lval}') = \gamma_2(\mathit{lval}')$. This indeed mean that the labels of lval' were also L in both γ_1 and γ_2, and given the assumption that $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$, thus indeed $m_1(\mathit{lval}') = m_2(\mathit{lval}')$.

case $\mathit{lval}' = \mathit{lval}$:
For $\mathit{lval}' = \mathit{lval}$, we know that in $\gamma_1' \sqcup \gamma_2'$ we type the lval as L, i.e. $\gamma_1' \sqcup \gamma_2'(\mathit{lval}) = \tau \wedge \mathrm{lbl}(\tau) = L$ thus the same property holds in individual state types $\gamma_1'(\mathit{lval}) = \tau' \ \wedge \ \mathrm{lbl}(\tau') = L$ and also $\gamma_2'(\mathit{lval}) = \tau'' \ \wedge \ \mathrm{lbl}(\tau'') = L$. We need to prove $m_1'(\mathit{lval}) = m_2'(\mathit{lval})$. For lval to be L after the update function in either γ_1' or γ_2', it is necessary for the types of the expression e to be L in both initial state types γ_1 in 1 and γ_2 in 5 in typing rules, i.e. ($\mathrm{lbl}(\tau_1) = L$ and $\mathrm{lbl}(\tau_2) = L$). This condition holds because otherwise, if the typing labels of e were H, then the goal would be trivially true (as the update would make lval H in γ_1' and γ_2', contradicting the assumptions). Since the typing label of e is L in 1 and 5 of typing rule, when reduced to a value in 1 and 3 of the semantics rule, this indicates that they reduce to the same value $v_1 = v_2$ (using Lemma E.8). Since we update lval in m_1 and m_2 with the same value such that we produce m_1' and m_2' respectively, it follows that $m_1'(\mathit{lval}) = m_2'(\mathit{lval})$.

case $\mathit{lval}' \subseteqq \mathit{lval}$:
For $\mathit{lval}' \subseteqq \mathit{lval}$, we know that lval' can be a shorter variation of the lval. The proof is the same as the previous case.

case $\mathit{lval} \subseteqq \mathit{lval}'$:
For $\mathit{lval} \subseteqq \mathit{lval}'$, we know that lval can be a shorter variation of the lval', thus the update of lval affects part of lval' type while the rest of it stays unchanged. Therefore, the proof is straightforward by conducing the same steps of the first two cases.

Given the last four subcases, we can now show that $m_1' \underset{\gamma'}{\sim} m_2'$.

◇ **Case condition**: Here *stmt* is `if` e `then` s_1 `else` s_2. From conditional typing rule we know:

1. $\gamma \vdash e : \tau_1$
2. $\ell_1 = \mathrm{lbl}(\tau_1)$
3. $pc_1 = pc \sqcup \ell_1$
4. $T, pc_1, (\mathrm{refine}(\gamma_1, e)) \vdash s_1 : \Gamma_1$
5. $T, pc_1, (\mathrm{refine}(\gamma_1, \neg e)) \vdash s_2 : \Gamma_2$
6. $\Gamma_3 = \begin{cases} \mathrm{join}(\Gamma_1 \cup \Gamma_2) & \text{if } \ell_1 = H \\ \Gamma_1 \cup \Gamma_2 & \text{otherwise} \end{cases}$
7. $\gamma \vdash e : \tau_2$
8. $\ell_2 = \mathrm{lbl}(\tau_2)$
9. $pc_2 = pc \sqcup \ell_2$
10. $T, pc_2, (\mathrm{refine}(\gamma_2, e)) \vdash s_1 : \Gamma_4$
11. $T, pc_2, (\mathrm{refine}(\gamma_2, \neg e)) \vdash s_2 : \Gamma_5$
12. $\Gamma_6 = \begin{cases} \mathrm{join}(\Gamma_4 \cup \Gamma_5) & \text{if } \ell_2 = H \\ \Gamma_4 \cup \Gamma_5 & \text{otherwise} \end{cases}$

We also know that both initial states are $\gamma_1 \vdash m_1$ and $\gamma_2 \vdash m_2$ and also $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$. And we know that the conditional statement is executed with m_1 and m_2 resulting m_1' and m_2' consequently.

In addition to that, we get induction hypothesis for s_1 IH1 and s_2 IH2 (we only show IH1):

$$\begin{gathered}
\forall\, T\ pc\ m_a\ m_b\ m_a'\ m_b'\ \gamma_a\ \gamma_b\ \Gamma_a\ \Gamma_b\ E_a\ E_b. \\
T, pc, \gamma_a \vdash s_1 : \Gamma_a \ \wedge\ T, pc, \gamma_b \vdash s_1 : \Gamma_b \implies \\
T \vdash E_a \ \wedge\ T \vdash E_b \ \wedge\ E_a \underset{T}{\sim} E_b \ \wedge \\
\gamma_a \vdash m_a \ \wedge\ \gamma_b \vdash m_b \ \wedge\ m_a \underset{\gamma_a \sqcup \gamma_b}{\sim} m_b \ \wedge \\
E_a : m_a \xrightarrow{s_1} m_a' \ \wedge\ E_b : m_b \xrightarrow{s_1} m_b' \\
\implies \\
\left(\exists \gamma_a' \in \Gamma_a \ \wedge\ \gamma_b' \in \Gamma_b.\ \gamma_a' \vdash m_a' \ \wedge\ \gamma_b' \vdash m_b' \ \wedge\ m_a' \underset{\gamma_a' \sqcup \gamma_b'}{\sim} m_b'\right)
\end{gathered}$$

We start by cases on labels ℓ_1 and ℓ_2 of e.

case $\ell_1 = \ell_2 = L$: We need to prove that $\exists \gamma_1' \in \Gamma_3$ and $\exists \gamma_2' \in \Gamma_6$ it holds $m_1' \underset{\gamma_1' \sqcup \gamma_2'}{\sim} m_2'$. We can directly use Lemma E.8 to infer that e is evaluation is indistinguishable in the states, and since e is assumed to be typed as boolean, thus we get two subcases where: $m_1(e) = true$ and $m_2(e) = true$, or $m_1(e) = false$ and $m_2(e) = false$.

case $m_1(e) = true$ and $m_2(e) = true$: when looking into the reduction rule of the both if statements in the assumption, we only reduce the first branch of each. Hence, we have $E_1 : m_1 \xrightarrow{s_1} m_1'$ and $E_2 : m_2 \xrightarrow{s_1} m_2'$.

From Hyp E.4, we can show that $m_1 \underset{(\text{refine}(\gamma_1,e)) \sqcup (\text{refine}(\gamma_2,e))}{\sim} m_2$. Additionally, we can infer that $(\text{refine}(\gamma_1, e)) \vdash m_1$ and $(\text{refine}(\gamma_2, e)) \vdash m_2$ using Hyp E.1.

Now we can directly instantiate and apply IH1 using the following $(T, pc \sqcup L, m_1, m_2, m_1', m_2', (\text{refine}(\gamma_1, e)), (\text{refine}(\gamma_2, e)), \Gamma_1, \Gamma_4, E_1, E_2)$ to infer that the states after executing s_1 are low equivalent, i.e. exists $\gamma_1'' \in \Gamma_1$ and exists $\gamma_2'' \in \Gamma_4$ such that $\gamma_1'' \vdash m_1'$ and $\gamma_2'' \vdash m_2'$ and $m_1' \underset{\gamma_1'' \sqcup \gamma_2''}{\sim} m_2'$. Since $\Gamma_1 \subseteq \Gamma_3$ and $\Gamma_4 \subseteq \Gamma_6$, thus the goal holds.

case $m_1(e) = false$ and $m_2(e) = false$ same proof as the previous case.

case $\ell_2 = H$: We initiate the proof by fixing ℓ_2 to be H, and the value of e to be reduced to $false$, thus it executes s_2 (starting from configuration m_2, and yields m_2', note that if e reduces to $true$ the proof is identical as this case). In this proof, we refer to these as the second configuration.

Now, consider the following scenario where we start from m_1 in the semantics rule and γ_1 in typing rule, we generalize the proof for any boolean expression e_i such that i ranges over $true$ and $false$, where e_{true} is e, e_{false} is $\neg e$, s_{true} is the first branch s_1, and s_{false} is the second branch s_2. In this sub-case of the proof, ℓ_1 denotes the label associated e_i's typing label, and let the refinement of the initial typing scope γ_1 to be represented as $\text{refine}(\gamma_1, e_i)$. Suppose the executed branch is s_i, yielding a final set of state types denoted as Γ_i. In this proof, we refer to these as the first configuration.

Given the previous generalizations, we can rewrite the assumptions to:

(a) $T, \ell_1, (\text{refine}(\gamma_1, e_i)) \vdash s_i : \Gamma_i$

(b) $E_1 : m_1 \xrightarrow{s_i} m_1'$

(c) $T, H, (\text{refine}(\gamma_2, \neg e)) \vdash s_2 : \Gamma_5$

(d) $T, H, (\text{refine}(\gamma_2, e)) \vdash s_1 : \Gamma_4$

(e) $E_2 : m_2 \xrightarrow{s_2} m_2'$

What we aim to prove is the existence of $\gamma_1' \in \Gamma_3$ such that $\gamma_1' \vdash m_1'$. Additionally, we need to establish the existence of $\gamma_2' \in \Gamma_6$ where $\Gamma_6 = \text{join}(\Gamma_4 \cup \Gamma_5)$, such that $\gamma_2' \vdash m_2'$. Furthermore, we must prove that $m_1' \underset{\gamma_1' \sqcup \gamma_2'}{\sim} m_2'$.

From the Soundness of Abstraction, we know that for the second configuration indeed exists $\gamma_2'' \in \Gamma_5$ such that it types m_2' (i.e. $\gamma_2'' \vdash m_2'$).

Given that $\gamma_2'' \in \Gamma_5$ and $\Gamma_6 = \text{join}(\Gamma_4 \cup \Gamma_5)$, we can deduce (by Lemma E.7) the existence of $\overline{\gamma_2''} \in \text{join}(\Gamma_4 \cup \Gamma_5)$ such that it is more restrictive than γ_2'', denoted as $\gamma_2'' \sqsubseteq \overline{\gamma_2''}$. Using the same lemma, we conclude that $\overline{\gamma_2''} \vdash m_2'$. Now on, we choose $\overline{\gamma_2''}$ to be used in the proof and resolve the second conjunction of the goal.

From the Soundness of Abstraction, we know that for the first configuration indeed exists $\gamma_i'' \in \Gamma_i$ such that it types m_1' (i.e. $\gamma_i'' \vdash m_1'$).

In the first configuration, we generalized the proof according to the evaluation of e_i. Consequently, the final state type set Γ_3 can be either a union (if $\ell_1 = L$) or a join (if $\ell_1 = H$) of all final state type sets $\forall i \leq 1$. Γ_i resulting from typing their corresponding s_i. In either case (union or join), we can establish the existence of $\overline{\gamma_i''} \in \Gamma_3$ such that $\gamma_i'' \sqsubseteq \overline{\gamma_i''}$ and indeed $\overline{\gamma_i''} \vdash m_1'$. Note that if Γ_3 resulted from a join, we infer this using Lemma E.7; otherwise, if it resulted from a union, it is trivially true. In fact, we can directly choose $\gamma_i'' = \overline{\gamma_i''}$ when union the final state types sets.

Next, we proceed to implement cases based on whether an *lval*'s type label is H or L.

case $(\overline{\gamma_i''} \sqcup \overline{\gamma_2''}(\mathit{lval}) = \tau) \ \wedge \ \text{lbl}(\tau) = H$: the goal holds trivially.

case $(\overline{\gamma_i''} \sqcup \overline{\gamma_2''}(\mathit{lval}) = \tau) \ \wedge \ \text{lbl}(\tau) = L$: this case entails that each state type individually holds $\overline{\gamma_i''}(\mathit{lval}) = \tau_1' \ \wedge \ \text{lbl}(\tau_1') = L$ and also $\overline{\gamma_2''}(\mathit{lval}) = \tau_2'' \ \wedge \ \text{lbl}(\tau_2'') = L$.

Given that the *lval*'s type is L in $\overline{\gamma_2''}$, and considering $\overline{\gamma_2''} \in \text{join}(\Gamma_4 \cup \Gamma_5)$, it follows that the *lval* is also L in any state type within $\text{join}(\Gamma_4 \cup \Gamma_5)$. Consequently, the *lval* is L in the state types of both Γ_5 and Γ_4 (if not empty) individually before the join operation. Hence, since $\gamma_2'' \in \Gamma_5$, it implies that *lval* is also L in γ_2'' expressed as $\gamma_2''(\mathit{lval}) = \tau_2''' \wedge \text{lbl}(\tau_2''') = L$ (using Lemma E.11). We also know that typing a statement in a H context in (d) entails that the final Γ_4 is not empty (using Lemma E.14), thus *lval*'s type label is also L in all the state types in Γ_4.

In the second configuration, we type the statement s_2 with a H context in (c), and s_2 reduces to m_2' in (e). Furthermore, from the previous step, we inferred that the *lval*'s type is L in γ_2''. Hence, we can use Lemma E.10 to infer that the initial and final states remain unchanged for L lvalues, which means $m_2(\mathit{lval}) = m_2'(\mathit{lval})$. Then we can use Lemma

E.12 to infer that $\mathrm{refine}(\gamma_2, \neg e) \sqsubseteq \gamma_2''$, this entails that the *lval*'s type label is indeed L in the refined state $\mathrm{refine}(\gamma_2, \neg e)$. It is easy to see that $\gamma_2 \sqsubseteq \mathrm{refine}(\gamma_2, \neg e)$, thus *lval*'s type is also L in the initial state type γ_2, i.e. $\gamma_2(\mathit{lval}) = \tau_2' \ \wedge \ \mathrm{lbl}(\tau_2') = L$.

For the first configuration, in this sub-case, we have $\overline{\gamma_i''}(\mathit{lval}) = \tau_1' \ \wedge \ \mathrm{lbl}(\tau_1') = L$, and $\overline{\gamma_i''} \in \Gamma_3$. We previously showed $\gamma_i'' \in \Gamma_i$ and $\gamma_i'' \sqsubseteq \overline{\gamma_i''}$, where Γ_i such that is the final state type of typing s_i according to the evaluation of e_i. Since *lval*'s typing label is L in $\overline{\gamma_i''}$ in Γ_3 and we know that the state types in $\overline{\gamma_i''}$ are more restrictive than the state types in γ_i'', we can conclude that $\gamma_i'' \in \Gamma_i$ also types *lval* as L $\gamma_i''(\mathit{lval}) = \tau_1'' \ \wedge \ \mathrm{lbl}(\tau_1'') = L$.

Previously, we demonstrated that the *lval*'s typing label is L in all final state types in Γ_4 and Γ_5, and we showed that Γ_4 is not empty. Consequently, neither s_1 nor s_2 can modify *lval*. Considering the assumptions (a) and (b) (related to the first configuration), where s_i can be either s_1 or s_2, we conclude that the *lval* remains unchanged there as well. Here, we can apply Lemma E.13 to deduce that the *lval*'s typing label in $\gamma_i'' \in \Gamma_i$ are more restrictive than the one we find in the refined state $\mathrm{refine}(\gamma_1, e_i)$, thus $\mathrm{refine}(\gamma_1, e_i)(\mathit{lval}) = \tau_i' \ \wedge \ \mathrm{lbl}(\tau_i') = L$. Now we can apply Lemma E.10 for any s_i to show that $m_1(\mathit{lval}) = m_1'(\mathit{lval})$.

Finally, since the typing label of *lval* in $\mathrm{refine}(\gamma_1, e_i)$ is L, then trivially we know that the typing label of *lval* in γ_1 is also L because $\gamma_1 \sqsubseteq \mathrm{refine}(\gamma_1, e_i)$. We previously showed that *lval*'s typing label is L in γ_2, thus we now can show that $\gamma_1 \sqcup \gamma_2(\mathit{lval}) = \tau' \ \wedge \ \mathrm{lbl}(\tau') = L$. Now, we can deduce that $m_1(\mathit{lval}) = m_2(\mathit{lval})$ from the definition of the assumption $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$. Thus, the goal holds.

case $\ell_1 = H$: when fixing the first configuration, we implement same proof as previous case.

◇ **Case sequence**: Here *stmt* is $s_1; s_2$. From sequence typing rule we know:

1. $T, pc, \gamma_1 \vdash s_1 : \Gamma_1$
2. $\forall \gamma_1' \in \Gamma_1.\ T, pc, \gamma_1' \vdash s_2 : \Gamma_2^{\gamma_1'}$
3. $\Gamma' = \bigcup_{\gamma_1' \in \Gamma_1} \Gamma_2^{\gamma_1'}$
4. $T, pc, \gamma_2 \vdash s_1 : \Gamma_3$
5. $\forall \gamma_2' \in \Gamma_3.\ T, pc, \gamma_2' \vdash s_2 : \Gamma_4^{\gamma_2'}$
6. $\Gamma'' = \bigcup_{\gamma_2' \in \Gamma_3} \Gamma_4^{\gamma_2'}$

And from sequence reduction rule we know:

1. $E_1 : m_1 \xrightarrow{s_1} m'_1$
2. $E_1 : m'_1 \xrightarrow{s_2} m''_1$
3. $E_2 : m_2 \xrightarrow{s_1} m'_2$
4. $E_2 : m'_2 \xrightarrow{s_2} m''_2$

We initially know that : $\gamma_1 \vdash m_1$, $\gamma_2 \vdash m_2$, and $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$.

In addition to that, we get induction hypothesis for s_1 IH1 and s_2 IH2 (we only show IH1):

$$\begin{gathered}
\forall\ T\ pc\ m_a\ m_b\ m'_a\ m'_b\ \gamma_a\ \gamma_b\ \Gamma_a\ \Gamma_b\ E_a\ E_b. \\
T, pc, \gamma_a \vdash s_1 : \Gamma_a \ \wedge \ T, pc, \gamma_b \vdash s_1 : \Gamma_b \implies \\
T \vdash E_a \ \wedge \ T \vdash E_b \ \wedge \ E_a \underset{T}{\sim} E_b \ \wedge \\
\gamma_a \vdash m_a \ \wedge \ \gamma_b \vdash m_b \ \wedge \ m_a \underset{\gamma_a \sqcup \gamma_b}{\sim} m_b \ \wedge \\
E_a : m_a \xrightarrow{s_1} m'_a \ \wedge \ E_b : m_b \xrightarrow{s_1} m'_b \\
\implies \\
\big(\exists \gamma'_a \in \Gamma_a \ \wedge \ \gamma'_b \in \Gamma_b.\ \gamma'_a \vdash m'_a \ \wedge \ \gamma'_b \vdash m'_b \ \wedge \ m'_a \underset{\gamma'_a \sqcup \gamma'_b}{\sim} m'_b\big)
\end{gathered}$$

We need to prove there are two state types $\gamma''_1 \in \Gamma'$ and $\gamma''_2 \in \Gamma''$ such they type the final states $\gamma''_1 \vdash m''_1$ and $\gamma''_2 \vdash m''_2$ and indeed $m''_1 \underset{\gamma''_1 \sqcup \gamma''_2}{\sim} m''_2$ holds.

We start by using IH1, and instantiating it with $(T, pc, m_1, m_2, m'_1, m'_2, \gamma_1, \gamma_2, \Gamma_1, \Gamma_3, E_1, E_2)$ to infer that there exists $\gamma'_1 \in \Gamma_1$ and $\gamma'_2 \in \Gamma_3$ such that $\gamma'_1 \vdash m'_1 \ \wedge \ \gamma'_2 \vdash m'_2$ and also $m'_1 \underset{\gamma'_1 \sqcup \gamma'_2}{\sim} m'_2$.

Then, in the typing rule, we instantiate 2 with γ'_1 and 5 with γ'_2. Now we can use IH2, and instantiating it with $(T, pc, m'_1, m'_2, m''_1, m''_2, \gamma'_1, \gamma'_2, \Gamma_2^{\gamma'_1}, \Gamma_4^{\gamma'_2}, E_1, E_2)$. From that we can infer that indeed there exists state types $\gamma''_1 \in \Gamma_2^{\gamma'_1}$ and $\gamma''_2 \in \Gamma_4^{\gamma'_2}$ such that they type the final states $\gamma''_1 \vdash m''_1$ and $\gamma''_2 \vdash m''_2$, where they keep the states low equivalent as $m''_1 \underset{\gamma''_1 \sqcup \gamma''_2}{\sim} m''_2$.

We know that the final set of state type of interest is simply the union of all state types that can type the second statement in 3 and 6. It is easy to see that since $\gamma''_1 \in \Gamma_2^{\gamma'_1}$ then $\gamma''_1 \in \Gamma'$. Similarly, $\gamma''_2 \in \Gamma_4^{\gamma'_2}$ then $\gamma''_2 \in \Gamma''$. Thus, the goal is proven.

◇ **Case function call**: Here *stmt* is $f(e_1, ..., e_n)$. From call typing rule we know (note that here we explicitly write the global and local state type):

1. $(\gamma_{g1}, \gamma_{l1}) \vdash e_i : \tau_{i1}$
2. $(s, (x_1, d_1), ..., (x_n, d_n)) = (C, F)(f)$
3. $\gamma_{f1} = \{x_i \mapsto \tau_{i1}\}$
4. $(C, F), pc, (\gamma_{g1}, \gamma_{f1}) \vdash s : \Gamma'_1$
5. $\Gamma''_1 = \{(\gamma'_{g1}, \gamma_{l1})[e_i \mapsto \gamma'_{f1}(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_{g1}, \gamma'_{f1}) \in \Gamma'_1\}$
6. $(\gamma_{g2}, \gamma_{l2}) \vdash e_i : \tau_{i2}$
7. $\gamma_{f2} = \{x_i \mapsto \tau_{i2}\}$
8. $(C, F), pc, (\gamma_{g2}, \gamma_{f2}) \vdash s : \Gamma'_2$
9. $\Gamma''_2 = \{(\gamma'_{g2}, \gamma_{l2})[e_i \mapsto \gamma'_{f2}(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_{g2}, \gamma'_{f2}) \in \Gamma'_2\}$

And from call reduction rule we know:

1. $(s, (x_1, d_1), ..., (x_n, d_n)) = (X_1, F)(f)$
2. $m_{f1} = \{x_i \mapsto (m_{g1}, m_{l1})(e_i)\}$
3. $(X_1, F) : (m_{g1}, m_{f1}) \xrightarrow{s} (m'_{g1}, m'_{f1})$
4. $m''_1 = (m'_{g1}, m_{l1})[e_i \mapsto m'_{f1}(x_i) \mid \text{isOut}(d_i)]$
5. $(s, (x_1, d_1), ..., (x_n, d_n)) = (X_2, F)(f)$
6. $m_{f2} = \{x_i \mapsto (m_{g2}, m_{l2})(e_i)\}$
7. $(X_2, F) : (m_{g2}, m_{f2}) \xrightarrow{s} (m'_{g2}, m'_{f2})$
8. $m''_2 = (m'_{g2}, m_{l2})[e_i \mapsto m'_{f2}(x_i) \mid \text{isOut}(d_i)]$

Let $m_1 = (m_{g1}, m_{l1})$, $m_2 = (m_{g2}, m_{l2})$, let $\gamma_1 = (\gamma_{g1}, \gamma_{l1})$, and $\gamma_1 = (\gamma_{g2}, \gamma_{l2})$. We initially know that: $\gamma_1 \vdash m_1$, $\gamma_2 \vdash m_2$, and $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$.

In addition to that, we get induction hypothesis for s IH:

$$\begin{gathered}
\forall\ T\ pc\ m_a\ m_b\ m'_a\ m'_b\ \gamma_a\ \gamma_b\ \Gamma_a\ \Gamma_b\ E_a\ E_b. \\
T, pc, \gamma_a \vdash s : \Gamma_a \ \wedge\ T, pc, \gamma_b \vdash s : \Gamma_b \implies \\
T \vdash E_a \ \wedge\ T \vdash E_b \ \wedge\ E_a \underset{T}{\sim} E_b \ \wedge \\
\gamma_a \vdash m_a \ \wedge\ \gamma_b \vdash m_b \ \wedge\ m_a \underset{\gamma_a \sqcup \gamma_b}{\sim} m_b \ \wedge \\
E_a : m_a \xrightarrow{s} m'_a \ \wedge\ E_b : m_b \xrightarrow{s} m'_b \\
\implies \\
\left(\exists \gamma'_a \in \Gamma_a \ \wedge\ \gamma'_b \in \Gamma_b.\ \gamma'_a \vdash m'_a \ \wedge\ \gamma'_b \vdash m'_b \ \wedge\ m'_a \underset{\gamma'_a \sqcup \gamma'_b}{\sim} m'_b\right)
\end{gathered}$$

We need to prove there are two state types $\gamma_1'' \in \Gamma_1''$ and $\gamma_2'' \in \Gamma_2''$ such they type the final states $\gamma_1'' \vdash m_1''$ and $\gamma_2'' \vdash m_2''$ and indeed $m_1'' \underset{\gamma_1'' \sqcup \gamma_2''}{\sim} m_2''$ holds.

First we need to prove that the resulted copy-in map is also well-typed i.e. $\gamma_{f1} \vdash m_{f1}$. Note that the same proof applies to prove $\gamma_{f2} \vdash m_{f2}$:

Given that initially the typing state types the state $(\gamma_{g1}, \gamma_{l1}) \vdash (m_{g1}, m_{l1})$ from assumptions and given that the expressions e_i have a type τ_{i1} from typing rule 1, now we can use Lemma E.4 to infer that for all $v_i : \tau_i$ such that v_i is the evaluation of $(m_{g1}, m_{l1})(e_i)$. This leads us to trivially infer that $\forall i.\ \{x_i \mapsto \tau_i\} \vdash \{x_i \mapsto v_i\}$ holds, thus $\gamma_{f1} \vdash m_{f1}$.

Now we want to show that $(m_{g1}, m_{f1}) \underset{(\gamma_{g1}, \gamma_{f1}) \sqcup (\gamma_{g2}, \gamma_{f2})}{\sim} (m_{g2}, m_{f2})$. This can be broken down and rewritten into two goals according to Lemma E.15:

Goal 1. $m_{g1} \underset{\gamma_{g1} \sqcup \gamma_{g2}}{\sim} m_{g2}$:
We know that the domains of m_{g1} and m_{l1} are distinct and do not intersect (similarly for m_{g2} and m_{l2}), and given that they are initially low equivalent with respect to $(\gamma_{g1}, \gamma_{l1}) \sqcup (\gamma_{g2}, \gamma_{l2})$ as $(m_{g1}, m_{l1}) \underset{(\gamma_{g1}, \gamma_{l1}) \sqcup (\gamma_{g2}, \gamma_{l2})}{\sim} (m_{g2}, m_{l2})$. Then we can use Lemma E.15 to show that the goal holds.

Goal 2. $m_{f1} \underset{\gamma_{f1} \sqcup \gamma_{f2}}{\sim} m_{f2}$:
First we start by rewriting the goal as following:
$\{x_i \mapsto (m_{g1}, m_{l1})(e_i)\} \underset{\{x_i \mapsto (\gamma_{g1}, \gamma_{l1})(e_i)\} \sqcup \{x_i \mapsto (\gamma_{g2}, \gamma_{l2})(e_i)\}}{\sim} \{x_i \mapsto (m_{g2}, m_{l2})(e_i)\}$.
It is easy to see that the empty map is low equivalent as: $\{\} \underset{\{\} \sqcup \{\}}{\sim} \{\}$. Now we can use Lemma E.16 to show that the goal holds.

Now we can use IH, and instantiate it with: $((C, F), pc, (m_{g1}, m_{f1}), (m_{g2}, m_{f2}), (m'_{g1}, m'_{f1}), (m'_{g2}, m'_{f2}), (\gamma_{g1}, \gamma_{f1}), (\gamma_{g2}, \gamma_{f2}), \Gamma_1', \Gamma_2', (X_1, F), (X_2, F))$, so that we can deduct that exists $\overline{\gamma_1'} \in \Gamma_1'$ such that $\overline{\gamma_1'} \vdash (m'_{g1}, m'_{f1})$, also exists $\overline{\gamma_2'} \in \Gamma_2'$ such that $\overline{\gamma_2'} \vdash (m'_{g2}, m'_{f2})$. We can also infer that $(m'_{g1}, m'_{f1}) \underset{\overline{\gamma_1'} \sqcup \overline{\gamma_2'}}{\sim} (m'_{g2}, m'_{f2})$.

Let $\overline{\gamma_1'} = (\overline{\gamma'_{g1}}, \overline{\gamma'_{f1}})$ and $\overline{\gamma_2'} = (\overline{\gamma'_{g2}}, \overline{\gamma'_{f2}})$ then we can also conclude from the previous:

(a) $\overline{\gamma'_{g1}} \vdash m'_{g1}$

(b) $\overline{\gamma'_{g2}} \vdash m'_{g2}$

(c) $\overline{\gamma'_{f1}} \vdash m'_{f1}$

(d) $\overline{\gamma'_{f2}} \vdash m'_{f2}$

(e) $m'_{g1} \underset{\gamma'_{g1} \sqcup \gamma'_{g2}}{\sim} m'_{g2}$

(f) $m'_{f1} \underset{\gamma'_{f1} \sqcup \gamma'_{f2}}{\sim} m'_{f2}$

Since the final goal is to prove there are two state types $\gamma''_1 \in \Gamma''_1$ and $\gamma''_2 \in \Gamma''_2$ such they type the final states $\gamma''_1 \vdash m''_1$ and $\gamma''_2 \vdash m''_2$ and indeed $m''_1 \underset{\gamma''_1 \sqcup \gamma''_2}{\sim} m''_2$ holds, then we can select γ''_1 to be $(\overline{\gamma'_{g1}}, \overline{\gamma'_{f1}})$, i.e. the copied-out map is $(\overline{\gamma'_{g1}}, \gamma_{l1})[e_i \mapsto \overline{\gamma'_{f1}}(x_i) \mid \text{isOut}(d_i)]$.

Similarly, we can select γ''_2 to be $(\overline{\gamma'_{g2}}, \overline{\gamma'_{f2}})$, i.e. the copied-out map is $(\overline{\gamma'_{g2}}, \gamma_{l2})[e_i \mapsto \overline{\gamma'_{f2}}(x_i) \mid \text{isOut}(d_i)]$.

Goal 1:

$$(\overline{\gamma'_{g1}}, \gamma_{l1})[e_i \mapsto \overline{\gamma'_{f1}}(x_i)] \vdash (m'_{g1}, m_{l1})[e_i \mapsto m'_{f1}(x_i)]$$

Since we know $\overline{\gamma'_{g1}} \vdash m'_{g1}$ ((a) from previous step) and $\gamma_{l1} \vdash m_{l1}$ from assumptions rewrites, this entails that $(\overline{\gamma'_{g1}}, \gamma_{l1}) \vdash (m'_{g1}, m_{l1})$. We also proved that $\overline{\gamma'_{f1}} \vdash m'_{f1}$ ((c) from previous step), thus for all $x \in \text{domain}(\gamma'_{f1})$ holds $m'_{f1}(x) : \overline{\gamma'_{f1}}(x)$. Now we can use Lemma E.5 to show that the goal holds.

Goal 2:

$$(\overline{\gamma'_{g2}}, \gamma_{l2})[e_i \mapsto \overline{\gamma'_{f2}}(x_i)] \vdash (m'_{g2}, m_{l2})[e_i \mapsto m'_{f2}(x_i)]$$

Same proof as the previous sub goal, using (b) and (d) the previous step, and the initial assumption $\gamma_{l2} \vdash m_{l2}$.

Goal 3:

$$(m'_{g1}, m_{l1})[e_i \mapsto m'_{f1}(x_i)] \underset{((\overline{\gamma'_{g1}}, \gamma_{l1})[e_i \mapsto \overline{\gamma'_{f1}}(x_i)]) \sqcup ((\overline{\gamma'_{g2}}, \gamma_{l2})[e_i \mapsto \overline{\gamma'_{f2}}(x_i)])}{\sim} (m'_{g2}, m_{l2})[e_i \mapsto m'_{f2}(x_i)]$$

We know that $m'_{g1} \underset{\gamma'_{g1} \sqcup \gamma'_{g2}}{\sim} m'_{g2}$ (from (e) of previous step), we also know that $m'_{l1} \underset{\gamma_{l1} \sqcup \gamma_{l2}}{\sim} m_{l2}$ (from assumptions), now using Lemma E.15 we can combine them to infer an equivalence before copying out or (no copy-out because there are not out directed parameters), i.e. :

$(m'_{g1}, m_{l1}) \underset{\overline{(\gamma'_{g1}, \gamma_{l1})} \sqcup \overline{(\gamma'_{g2}, \gamma_{l2})}}{\sim} (m'_{g2}, m_{l2})$.

We previously proved that $m_{f1} \underset{\gamma_{f1} \sqcup \gamma_{f2}}{\sim} m_{f2}$, now we can prove this goal directly using Lemma E.16.

◇ **Case extern**: Here *stmt* is $f(e_1, ..., e_n)$. From call typing rule we know (Note that here we explicitly write the global and local state type):

1. $(\gamma_{g1}, \gamma_{l1}) \vdash e_i : \tau_{i1}$
2. $(\mathrm{Cont}_\mathrm{E}, (x_1, d_1), ..., (x_n, d_n)) = (C, F)(f)$
3. $\gamma_{f1} = \{x_i \mapsto \tau_{i1}\}$
4. $\forall (\gamma_i, \phi, \gamma_t) \in \mathrm{Cont}_\mathrm{E}.\ (\gamma_{g1}, \gamma_{l1}) \sqsubseteq \gamma_i$
5. $\Gamma'_1 = \{\gamma'_1 {+\!\!+} \mathrm{raise}(\gamma_t, pc) \mid (\gamma_i, \phi, \gamma_t) \in \mathrm{Cont}_\mathrm{E}\ \wedge\ \mathrm{refine}((\gamma_{g1}, \gamma_{f1}), \phi) = \gamma'_1 \neq \bullet\}$
6. $\Gamma''_1 = \{(\gamma'_g, \gamma_{l1})[e_i \mapsto \gamma'_f(x_i) \mid \mathrm{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'_1\}$
7. $(\gamma_{g2}, \gamma_{l2}) \vdash e_i : \tau_{i2}$
8. $\gamma_{f2} = \{x_i \mapsto \tau_{i2}\}$
9. $\forall (\gamma_i, \phi, \gamma_t) \in \mathrm{Cont}_\mathrm{E}.\ (\gamma_{g2}, \gamma_{l2}) \sqsubseteq \gamma_i$
10. $\Gamma'_2 = \{\gamma'_2 {+\!\!+} \mathrm{raise}(\gamma_t, pc) \mid (\gamma_i, \phi, \gamma_t) \in \mathrm{Cont}_\mathrm{E}\ \wedge\ \mathrm{refine}((\gamma_{g2}, \gamma_{f2}), \phi) = \gamma'_2 \neq \bullet\}$
11. $\Gamma''_2 = \{(\gamma'_g, \gamma_{l2})[e_i \mapsto \gamma'_f(x_i) \mid \mathrm{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'_2\}$

And from extern reduction rule we know:

1. $(sem_f, (x_1, d_1), ..., (x_n, d_n)) = (X_1, F)(f)$
2. $m_{f1} = \{x_i \mapsto (m_{g1}, m_{l1})(e_i)\}$
3. $(m'_{g1}, m'_{f1}) = sem_f(m_{g1}, m_{f1})$
4. $m''_1 = (m'_{g1}, m_{l1})[e_i \mapsto m'_{f1}(x_i) \mid \mathrm{isOut}(d_i)]$
5. $(sem_f, (x_1, d_1), ..., (x_n, d_n)) = (X_2, F)(f)$
6. $m_{f2} = \{x_i \mapsto (m_{g2}, m_{l2})(e_i)\}$
7. $(m'_{g2}, m'_{f2}) = sem_f(m_{g2}, m_{f2})$
8. $m''_2 = (m'_{g2}, m_{l2})[e_i \mapsto m'_{f2}(x_i) \mid \mathrm{isOut}(d_i)]$

Let $m_1 = (m_{g1}, m_{l1})$, $m_2 = (m_{g2}, m_{l2})$, let $\gamma_1 = (\gamma_{g1}, \gamma_{l1})$, and $\gamma_1 = (\gamma_{g2}, \gamma_{l2})$. We initially know that : $\gamma_1 \vdash m_1$, $\gamma_2 \vdash m_2$, and $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$.

We need to prove there are two state types $\gamma_1'' \in \Gamma_1''$ and $\gamma_2'' \in \Gamma_2''$ such they type the final states $\gamma_1'' \vdash m_1''$ and $\gamma_2'' \vdash m_2''$ and indeed $m_1'' \underset{\gamma_1'' \sqcup \gamma_2''}{\sim} m_2''$ holds.

First we need to prove that the resulted copy-in map is also well-typed i.e. $\gamma_{f1} \vdash m_{f1}$ and $\gamma_{f2} \vdash m_{f2}$: same proof as the function call case.

Now we want to show that $(m_{g1}, m_{f1}) \underset{(\gamma_{g1}, \gamma_{f1}) \sqcup (\gamma_{g2}, \gamma_{f2})}{\sim} (m_{g2}, m_{f2})$: same proof as the function call case.

We know from extWT C X_1 and extWT C X_2 relation in the well-typedness $(X_1, F) \vdash (C, F)$ and $(X_2, F) \vdash (C, F)$ that indeed there exists an input state type, condition and output state type i.e. $(\gamma_i, \phi, \gamma_t)$ in the contract of the extern such that the condition satisfies the input state $\phi(m_{g1}, m_{f1})$.

From Soundness of Abstraction proof, we know that $(\gamma_{g1}, \gamma_{f1}) \vdash (m_{g1}, m_{f1})$. Using 4 of extern's typing rule we know that $(\gamma_{g1}, \gamma_{f1})$ is no more restrictive than γ_i, i.e. $(\gamma_{g1}, \gamma_{f1}) \sqsubseteq \gamma_i$ holds. Using the same strategy and 9 of extern's typing rule, we can also prove $(\gamma_{g2}, \gamma_{f2}) \sqsubseteq \gamma_i$. We can also deduct that $(m_{g1}, m_{f1}) \underset{\gamma_i}{\sim} (m_{g2}, m_{f2})$ using Lemma E.9.

Let the variables $\{x_1, ..., x_n\}$ be the ones used in the condition ϕ. Using the definition of extern's well-typedness again, we know that the least upper bound of typing label of $\{x_1, ..., x_n\}$ in γ_i is no more restrictive that the lower bound of the output state type γ_t. The entails that since ϕ holds on m_1 (i.e. $\phi(m_1)$)then it indeed holds for m_2 (i.e. $\phi(m_2)$).

Now we split the proof into two cases:

A. For the changed variables by extern in the state: given 3 and 7 of the extern reduction rule, and using $\phi(m_1)$ we can use the definition of extern's well-typedness again to infer that the variables that are changed by the semantics are a subset of the domain of γ_t and low equivalent with respect to γ_t:

$$((m'_{g1}, m'_{f1}) \setminus (m_{g1}, m_{f1})) \underset{\gamma_t}{\sim} ((m'_{g2}, m'_{f2}) \setminus (m_{g2}, m_{f2}))$$

B. Now for the unchanged variables by extern in the state: it is easy to see that the refined state refine$((\gamma_{g1}, \gamma_{f1}), \phi)$ is more restrictive than $(\gamma_{g1}, \gamma_{f1})$, thus we can say $(\gamma_{g1}, \gamma_{f1}) \sqsubseteq$ refine$((\gamma_{g1}, \gamma_{f1}), \phi)$. Now we can use Lemma E.9 to show:

$$(m_{g1}, m_{f1}) \underset{\text{refine}((\gamma_{g1}, \gamma_{f1}), \phi) \sqcup \text{refine}((\gamma_{g1}, \gamma_{f1}), \phi)}{\sim} (m_{g2}, m_{f2})$$

This property also holds on the variable names x that are unchanged by the behavior of the extern, i.e. :

$$(m'_{g1}, m'_{f1}) \underset{\text{refine}((\gamma_{g1},\gamma_{f1}),\phi)\sqcup\text{refine}((\gamma_{g1},\gamma_{f1}),\phi)}{\sim} (m'_{g2}, m'_{f2})$$

We can rename $\text{refine}((\gamma_{g1}, \gamma_{f1}), \phi) \mathbin{+\!\!+} \gamma_t$ to $\overline{\gamma_3}$ and $\text{refine}((\gamma_{g2}, \gamma_{f2}), \phi) \mathbin{+\!\!+} \gamma_t$ to $\overline{\gamma_4}$, and we can easily infer from A and B that : $(m'_{g1}, m'_{f1}) \underset{\overline{\gamma_3}\sqcup\overline{\gamma_4}}{\sim} (m'_{g2}, m'_{f2})$.

The rest of the proof is the same as the function call case.

◇ **Case table application**: Here *stmt* is `apply` *tbl*. From table rule we know:

1. $(\overline{e}, \text{Cont}_{\text{tbl}}) = (C, F)(tbl)$
2. $\gamma_1 \vdash e_i : \tau_{i1}$
3. $\ell_1 = \bigsqcup_i \text{lbl}(\tau_{i1})$
4. $pc'_1 = pc \sqcup \ell_1$
5. $\forall (\phi_j, (a_j, \overline{\tau}_j)) \in \text{Cont}_{\text{tbl}}.\ (\gamma_{g_j}, \gamma_{l_j}) = \text{refine}(\gamma_1, \phi_j)\ \wedge$ $(s_j, (x_{j_1}, \text{none}), ..., (x_{j_n}, \text{none})) = (C, F)(a_j)\ \wedge$ $\gamma_{a_j} = \{x_{j_i} \mapsto \tau_{j_i}\}\ \wedge\ (C, F), pc'_1, (\gamma_{g_j}, \gamma_{a_j}) \vdash s_j : \Gamma_j$
6. $\Gamma'_1 = \cup_j \{(\gamma'_{g_j}, \gamma_{l_j}) | (\gamma'_{g_j}, \gamma'_{a_j}) \in \Gamma_j\}$
7. $\Gamma''_1 = \begin{cases} \text{join}(\Gamma'_1) & \text{if } \ell_1 = H \\ \Gamma'_1 & \text{otherwise} \end{cases}$
8. $\gamma_2 \vdash e_i : \tau_{i2}$
9. $\ell_2 = \bigsqcup_i \text{lbl}(\tau_{i2})$
10. $pc'_2 = pc \sqcup \ell_2$
11. $\forall (\phi_k, (a_k, \overline{\tau}_k)) \in \text{Cont}_{\text{tbl}}.\ (\gamma_{g_k}, \gamma_{l_k}) = \text{refine}(\gamma_2, \phi_k)\ \wedge$ $(s_k, (x_{k_1}, \text{none}), ..., (x_{k_n}, \text{none})) = (C, F)(a_k)\ \wedge$ $\gamma_{a_k} = \{x_{k_i} \mapsto \tau_{k_i}\}\ \wedge\ (C, F), pc'_2, (\gamma_{g_k}, \gamma_{a_k}) \vdash s_k : \Gamma_k$
12. $\Gamma'_2 = \cup_k \{(\gamma'_{g_k}, \gamma_{l_k}) | (\gamma'_{g_k}, \gamma'_{a_k}) \in \Gamma_k\}$
13. $\Gamma''_2 = \begin{cases} \text{join}(\Gamma'_2) & \text{if } \ell_2 = H \\ \Gamma'_2 & \text{otherwise} \end{cases}$

And from table reduction rule we know:

1. $(\overline{e}, sem_{tbl1}) = (X_1, F)(tbl)$

2. $sem_{tbl1}((m_{g1}, m_{l1})(e_1), ..., (m_{g1}, m_{l1})(e_n)) = (a_1, \overline{v})$

3. $(s_1, (x_1, \text{none}), ..., (x_n, \text{none})) = (X_1, F)(a_1)$

4. $m_{a1} = \{x_i \mapsto v_i\}$

5. $(X_1, F) : (m_{g1}, m_{a1}) \xrightarrow{s_1} (m'_{g1}, m'_{a1})$

6. $m_{final1} = (m'_{g1}, m_{l1})$

7. $(\overline{e}', sem_{tbl2}) = (X_2, F)(tbl)$

8. $sem_{tbl2}((m_{g2}, m_{l2})(e'_1), ..., (m_{g2}, m_{l2})(e'_n)) = (a_2, \overline{v}')$

9. $(s_2, (x'_2, \text{none}), ..., (x'_n, \text{none})) = (X_2, F)(a_2)$

10. $m_{a2} = \{x'_i \mapsto v'_i\}$

11. $(X_2, F) : (m_{g2}, m_{a2}) \xrightarrow{s_2} (m'_{g2}, m'_{a2})$

12. $m_{final2} = (m'_{g2}, m_{l2})$

In addition to that, we get induction hypothesis for s_1 IH1 (similarly for s_2 IH2):

$$\begin{gathered}
\forall\ T\ pc\ m_a\ m_b\ m'_a\ m'_b\ \gamma_a\ \gamma_b\ \Gamma_a\ \Gamma_b\ E_a\ E_b. \\
T, pc, \gamma_a \vdash s : \Gamma_a\ \wedge\ T, pc, \gamma_b \vdash s : \Gamma_b \implies \\
T \vdash E_a\ \wedge\ T \vdash E_b\ \wedge\ E_a \underset{T}{\sim} E_b\ \wedge \\
\gamma_a \vdash m_a\ \wedge\ \gamma_b \vdash m_b\ \wedge\ m_a \underset{\gamma_a \sqcup \gamma_b}{\sim} m_b\ \wedge \\
E_a : m_a \xrightarrow{s_1} m'_a\ \wedge\ E_b : m_b \xrightarrow{s_1} m'_b \\
\implies \\
\left(\exists \gamma'_a \in \Gamma_a\ \wedge\ \gamma'_b \in \Gamma_b.\ \gamma'_a \vdash m'_a\ \wedge\ \gamma'_b \vdash m'_b\ \wedge\ m'_a \underset{\gamma'_a \sqcup \gamma'_b}{\sim} m'_b\right)
\end{gathered}$$

Let $m_1 = (m_{g1}, m_{l1})$, $m_2 = (m_{g2}, m_{l2})$, let $\gamma_1 = (\gamma_{g1}, \gamma_{l1})$, and $\gamma_2 = (\gamma_{g2}, \gamma_{l2})$. We initially know that : $\gamma_1 \vdash m_1$, $\gamma_2 \vdash m_2$, and $m_1 \underset{\gamma_1 \sqcup \gamma_2}{\sim} m_2$.

The final goal is to prove there are two state types $\gamma''_1 \in \Gamma''_1$ and $\gamma''_2 \in \Gamma''_2$ such they type the final states $\gamma''_1 \vdash (m'_{g1}, m_{l1})$ and $\gamma''_2 \vdash (m'_{g2}, m_{l2})$ and indeed $(m'_{g1}, m_{l1}) \underset{\gamma''_1 \sqcup \gamma''_2}{\sim} (m'_{g2}, m_{l2})$ holds.

Initially, we prove expression (table keys) found in 1 in reduction rule, with 1 and 7 of typing rule are the same. We use $(X_1, F) \underset{(C,F)}{\sim} (X_2, F)$ to show that 1 and 7 in the typing rule have the same expression (i.e. $\overline{e} = \overline{e}'$). Then, using tblWT C X_1 and tblWT C X_2, we confirm that the expression in rule 1 of the reduction rule matches those in rules 1 and 7 of the typing rule. Therefore, all relevant expressions are equivalent.

First, we conduct a case analysis on ℓ_1 and ℓ_2 being equivalent:

case $\ell_1 = \ell_2 = L$: Given that ℓ_1 and ℓ_2 are L in 3 and 9 of the typing rule, respectively, we can conclude that the evaluation of the key expressions $\overline{e}$ in states m_1 and m_2 are identical. This follows directly from Lemma E.8, which establishes that $m_1(e_i) = m_2(e_i)$ for all e_i.

By the definition of $(X_1, F) \underset{(C,F)}{\sim} (X_2, F)$, we know that for any memory states m_1 and m_2, if $m_1(e_i) = m_2(e_i)$ for all table's keys e_i, then $m_1(\phi) \Leftrightarrow m_2(\phi)$. This implies that the condition for both tables to match is identical. Applying this definition again, we deduct that the actions a_1 and a_2 in 2 and 8 of the reduction rule are identical i.e. $a_1 = a_2 = a$, therefore their corresponding action bodies and signatures must also be the same in (X_1, F) and (X_2, F), thus $s_1 = s_2 = s$ and $\overline{x} = \overline{x}'$. Applying this definition $(X_1, F) \underset{(C,F)}{\sim} (X_2, F)$, yet again, we can also deduct that the action's values $\overline{v}$ and $\overline{v}'$ are low equivalent wrt. $\overline{\tau}$ i.e. $\overline{v} \underset{\overline{\tau}}{\sim} \overline{v}'$.

Now we can rewrite the table reduction rule to:

a) $(\overline{e}, sem_{tbl1}) = (X_1, F)(tbl)$

b) $sem_{tbl1}((m_{g1}, m_{l1})(e_1), ..., (m_{g1}, m_{l1})(e_n)) = (a, \overline{v})$

c) $(s, (x_1, \text{none}), ..., (x_n, \text{none})) = (X_1, F)(a)$

d) $m_{a1} = \{x_i \mapsto v_i\}$

e) $(X_1, F) : (m_{g1}, m_{a1}) \xrightarrow{s} (m'_{g1}, m'_{a1})$

f) $m_{final1} = (m'_{g1}, m_{l1})$

g) $(\overline{e}, sem_{tbl2}) = (X_2, F)(tbl)$

h) $sem_{tbl2}((m_{g2}, m_{l2})(e_1), ..., (m_{g2}, m_{l2})(e_n)) = (a, \overline{v}')$

i) $(s, (x_2, \text{none}), ..., (x_n, \text{none})) = (X_2, F)(a)$

j) $m_{a2} = \{x_i \mapsto v'_i\}$

k) $(X_2, F) : (m_{g2}, m_{a2}) \xrightarrow{s} (m'_{g2}, m'_{a2})$

l) $m_{final2} = (m'_{g2}, m_{l2})$

Given $(C, F) \vdash (X_1, F)$ that indeed exists condition ϕ_j in Cont_{tbl} that satisfies the table input state (m_{g1}, m_{l1}), and also exists a list of types $\overline{\tau}$ such that it can type that values of the table's semantics (in 2 of the typing rule) i.e. $v_i : \tau_i$. Similarly, from $(C, F) \vdash (X_2, F)$, we know that exists ϕ_k that satisfies the table input state (m_{g2}, m_{l2}), and exists $\overline{\tau}'$ such that $v'_i : \tau'_i$.

We can instantiate 5 from the table rule with $(\phi_j, (a, \overline{\tau}))$, and instantiate 11 with $(\phi_k, (a, \overline{\tau}'))$.

We need to prove $(m_{g1}, m_{a1}) \underset{(\gamma_{g_j},\gamma_{a_j})\sqcup(\gamma_{g_k},\gamma_{a_k})}{\sim} (m_{g2}, m_{a2})$ using the following sub-goals:

Goal 1. prove $m_{g1} \underset{\gamma_{g_j}\sqcup\gamma_{g_k}}{\sim} m_{g2}$ and $m_{l1} \underset{\gamma_{l_j}\sqcup\gamma_{l_k}}{\sim} m_{l2}$:

It is easy to see that $\gamma_1 \sqsubseteq \text{refine}(\gamma_1, \phi_j)$ and $\gamma_2 \sqsubseteq \text{refine}(\gamma_2, \phi_k)$ trivially hold, and can be rewritten to $\gamma_1 \sqsubseteq (\gamma_{g_j}, \gamma_{l_j})$ and $\gamma_2 \sqsubseteq (\gamma_{g_k}, \gamma_{l_k})$. Since initially we know that $(m_{g1}, m_{l1}) \underset{\gamma_1\sqcup\gamma_2}{\sim} (m_{g2}, m_{l2})$, therefore using Lemma E.9 $(m_{g1}, m_{l1}) \underset{(\gamma_{g_j},\gamma_{l_j})\sqcup(\gamma_{g_k},\gamma_{l_k})}{\sim} (m_{g2}, m_{l2})$. This entails using Lemma E.15 that $m_{g1} \underset{\gamma_{g_j}\sqcup\gamma_{g_k}}{\sim} m_{g2}$ and also $m_{l1} \underset{\gamma_{l_j}\sqcup\gamma_{l_k}}{\sim} m_{l2}$ hold.

Additionally, note that Hyp E.3 states also that $(\gamma_{g_j}, \gamma_{l_j})$ can still type the state (m_{g1}, m_{l1}), i.e. $(\gamma_{g_j}, \gamma_{l_j}) \vdash (m_{g1}, m_{l1})$. Similarly, $(\gamma_{g_k}, \gamma_{l_k}) \vdash (m_{g2}, m_{l2})$.

Goal 2. prove $m_{a1} \underset{\gamma_{a_j}\sqcup\gamma_{a_k}}{\sim} m_{a2}$: From $(X_1, F) \underset{(C,F)}{\sim} (X_2, F)$ we know it holds $\overline{v} \underset{\overline{\tau}}{\sim} \overline{v}'$, note that these are the table's semantic output (i.e. will be action's arguments). Thus, whenever τ_i is L, then the values of arguments are equivalent $v_i = v_i'$. This entails that constructing states $m_{a1} = \{x_i \mapsto v_i\}$ and $m_{a2} = \{x_i \mapsto v_i'\}$ must be low equivalent with respect to $\gamma_{a_j} = \{x_i \mapsto \tau_i\}$, i.e. $m_{a1} \underset{\gamma_{a_j}}{\sim} m_{a2}$. Similarly, from $(X_1, F) \underset{(C,F)}{\sim} (X_2, F)$ and whenever τ_i' is L, we can also deduct that $\gamma_{a_k} = \{x_i \mapsto \tau_i'\}$, i.e. $m_{a1} \underset{\gamma_{a_k}}{\sim} m_{a2}$. Therefore, this entails (using Lemma E.15) that $m_{a1} \underset{\gamma_{a_j}\sqcup\gamma_{a_k}}{\sim} m_{a2}$.

From the last two sub-goals, we can use Lemma E.15 to deduct

$$(m_{g1}, m_{a1}) \underset{(\gamma_{g_j},\gamma_{a_j})\sqcup(\gamma_{g_k},\gamma_{a_k})}{\sim} (m_{g2}, m_{a2})$$

Now we can use IH and instantiate it with $\big((C,F), pc \sqcup \ell, (m_{g1}, m_{a1}), (m_{g2}, m_{a2}), (m'_{g1}, m'_{a1}), (m'_{g2}, m'_{a2}), (\gamma_{g_j}, \gamma_{a_j}), (\gamma_{g_k}, \gamma_{a_k}), \Gamma_j, \Gamma_k, (X_1, F), (X_2, F)\big)$ to deduct that indeed exist $\overline{\gamma_1} \in \Gamma_j$ and $\overline{\gamma_2} \in \Gamma_k$ such that $\overline{\gamma_1} \vdash (m'_{g1}, m'_{a1})$ and $\overline{\gamma_2} \vdash (m'_{g2}, m'_{a2})$ and indeed $(m'_{g1}, m'_{a1}) \underset{\overline{\gamma_1}\sqcup\overline{\gamma_2}}{\sim} (m'_{g2}, m'_{a2})$. In the following, let $\overline{\gamma_1} = (\overline{\gamma_{1_g}}, \overline{\gamma_{1_l}})$ and $\overline{\gamma_2} = (\overline{\gamma_{2_g}}, \overline{\gamma_{2_l}})$. We can rewrite the IH results as following: exist $(\overline{\gamma_{1_g}}, \overline{\gamma_{1_l}}) \in \Gamma_j$ and $(\overline{\gamma_{2_g}}, \overline{\gamma_{2_l}}) \in \Gamma_k$ such that $(\overline{\gamma_{1_g}}, \overline{\gamma_{1_l}}) \vdash (m'_{g1}, m'_{a1})$ and $(\overline{\gamma_{2_g}}, \overline{\gamma_{2_l}}) \vdash (m'_{g2}, m'_{a2})$ and indeed $(m'_{g1}, m'_{a1}) \underset{(\overline{\gamma_{1_g}},\overline{\gamma_{1_l}})\sqcup(\overline{\gamma_{2_g}},\overline{\gamma_{2_l}})}{\sim} (m'_{g2}, m'_{a2})$.

Clearly, from 6 in the typing rule, and we know that Γ_1' is the union of all the changed global state types by the action's body, with the refined starting local state type $(\gamma_{g_j}, \gamma_{l_j}) = \text{refine}(\gamma_1, \phi_j)$. Thus, we know that indeed $(\overline{\gamma_{1_g}}, \gamma_{l_j}) \in \Gamma'_1$. Similarly, from 12 in the typing rule, we know that $(\overline{\gamma_{2_g}}, \gamma_{l_k}) \in \Gamma'_2$.

We choose $(\overline{\gamma_{1_g}}, \gamma_{l_j})$ and $(\overline{\gamma_{2_g}}, \gamma_{l_k})$ to finish the proof of the goal in this subcase.

Since $\ell_1 = \ell_2$ and is L in this sub-case, then trivially $\Gamma''_1 = \Gamma'_1$ in 7 of the typing rule, and $\Gamma''_2 = \Gamma'_2$ in 13 of the typing rule. Since we proved that $\gamma_{l_j} \vdash m_{l1}$ and $\gamma_{l_k} \vdash m_{l2}$, therefore $(\overline{\gamma_{1_g}}, \gamma_{l_j}) \vdash (m'_{g1}, m_{l1})$ and $(\overline{\gamma_{2_g}}, \gamma_{l_k}) \vdash (m'_{g2}, m_{l2})$ hold. Additionally, using Lemma E.15 $(m'_{g1}, m_{l1}) \underset{(\overline{\gamma_{1_g}}, \gamma_{l_j}) \sqcup (\overline{\gamma_{2_g}}, \gamma_{l_k})}{\sim} (m'_{g2}, m_{l2})$.

case $\ell_1 \neq \ell_2$: This proof is similar to the conditional case. We initiate the proof by fixing ℓ_2 to be H while ℓ_1 can be either H or L, thus the evaluation of $\overline{e}$ in the states (m_{g1}, m_{l1}) and (m_{g2}, m_{l2}) differs. Consequently, we (possibly) end up with two different actions and their corresponding arguments $(a_1, \overline{v})$ and $(a_2, \overline{v}')$.

From $(C, F) \vdash (X_1, F)$ we know that indeed exists condition ϕ_j in Cont_{tbl} that satisfies the table input state (m_{g1}, m_{l1}), and also exists a list of types $\overline{\tau}$ such that it can type that values of the table's semantics (in 2 of the typing rule) i.e. $v_i : \tau_i$. Similarly, from $(C, F) \vdash (X_2, F)$, we know that exists ϕ_k that satisfies the table input state (m_{g2}, m_{l2}), and exists $\overline{\tau}'$ such that $v'_i : \tau'_i$.

We can instantiate 5 (we refer to these as the first configuration) from the table rule with $(\phi_j, (a_1, \overline{\tau}))$, and instantiate 11 (we refer to these as the second configuration) with $(\phi_k, (a_2, \overline{\tau}'))$. Since the actions are different, then we let s_1 be the body of action a_1, and s_2 be the body of action a_2 (we refer to these as the first configuration).

Using the same steps in the previous sub-case, we know that $m_{g1} \underset{\gamma_{g_j} \sqcup \gamma_{g_k}}{\sim} m_{g2}$ and $m_{l1} \underset{\gamma_{l_j} \sqcup \gamma_{l_k}}{\sim} m_{l2}$.

The final goal is to prove there are two state types $\gamma''_1 \in \Gamma''_1$ and $\gamma''_2 \in \Gamma''_2$ such they type the final states $\gamma''_1 \vdash (m'_{g1}, m_{l1})$ and $\gamma''_2 \vdash (m'_{g2}, m_{l2})$ and indeed $(m'_{g1}, m_{l1}) \underset{\gamma''_1 \sqcup \gamma''_2}{\sim} (m'_{g2}, m_{l2})$ holds.

From the Soundness of Abstraction, we know that for the second configuration indeed exists $\gamma'_2 \in \Gamma_k$ such that it types the action's resulted state (m'_{g2}, m'_{a2}) (i.e. $\gamma'_2 \vdash (m'_{g2}, m'_{a2})$). In the following, let $(\gamma'_{g2}, \gamma'_{l2}) = \gamma'_2$, thus indeed $(\gamma'_{g2}, \gamma'_{l2}) \vdash (m'_{g2}, m'_{a2})$. Given that Γ'_2 in 12 of the typing rule is a union of all Γ_i; thus indeed it includes Γ_k that has the resulted global state type γ'_{g2}, and the refined caller's local state type γ_{l_k} while removing the callee's resulted local state type γ_{a_k}. In short, we know that indeed for the second configuration will find $(\gamma'_{g2}, \gamma_{l_k}) \in \Gamma'_2$. Note that in the Soundness of Abstraction previously we proved that $\gamma_{l_k} \vdash m_{l2}$, therefore $(\gamma'_{g2}, \gamma_{l_k}) \vdash (m'_{g2}, m_{l2})$. Since ℓ_2 is H, then $\Gamma''_2 = \text{join}(\Gamma'_2)$, so we can deduce (by Lemma E.7) the existence of $\overline{\gamma'_2} \in \text{join}(\Gamma'_2)$ such that it is more restrictive than γ'_2, denoted as $\gamma'_2 \sqsubseteq \overline{\gamma'_2}$.

Using the same lemma, we conclude that $\overline{\gamma'_2} \vdash (m'_{g2}, m_{l2})$. Now on, we choose $\overline{\gamma'_2}$ to be used in the proof and resolve the second conjunction of the goal.

From the Soundness of Abstraction, we know that for the first configuration indeed exists $\gamma'_1 \in \Gamma_j$ such that it types action's resulted state (m'_{g1}, m_{a1}) (i.e. $\gamma'_1 \vdash (m'_{g1}, m_{a1})$). In the first configuration, the final state type set Γ''_1 can be either a union (if $\ell_1 = L$) or a join (if $\ell_1 = H$) of all final state type sets and including Γ'_1. In either case (union or join), we can establish the existence of $\overline{\gamma'_1} \in \Gamma''_1$ such that $\gamma'_1 \sqsubseteq \overline{\gamma'_1}$ and indeed $\overline{\gamma'_1} \vdash (m'_{g1}, m_{l1})$. Note that if Γ''_1 resulted from a join, we follow the same steps of the (second configuration) in the previous step; otherwise, if it resulted from a union, it is trivially true. Thus, the first conjunction of the final goal is proved.

The final goal left to prove is $(m'_{g1}, m_{l1}) \underset{\overline{\gamma'_1} \sqcup \overline{\gamma'_2}}{\sim} (m'_{g2}, m_{l2})$ holds. In the following, let $(\overline{\gamma'_{g1}}, \overline{\gamma'_{l1}}) = \overline{\gamma'_1}$ and $(\overline{\gamma'_{g2}}, \overline{\gamma'_{l2}}) = \overline{\gamma'_2}$.

Next, we proceed to implement cases based on whether an *lval*'s type label is H or L.

case $(\overline{\gamma'_1} \sqcup \overline{\gamma'_2}(lval) = \tau) \ \wedge \ \mathrm{lbl}(\tau) = H$: holds trivially.

case $(\overline{\gamma'_1} \sqcup \overline{\gamma'_2}(lval) = \tau) \ \wedge \ \mathrm{lbl}(\tau) = L$: this case entails that each state type individually holds $\overline{\gamma'_1}(lval) = \tau'_1 \ \wedge \ \mathrm{lbl}(\tau'_1) = L$ and also $\overline{\gamma'_2}(lval) = \tau''_2 \ \wedge \ \mathrm{lbl}(\tau''_2) = L$.

Given that the *lval*'s type is L in $\overline{\gamma'_2}$, and considering $\overline{\gamma'_2} \in \mathrm{join}(\Gamma'_2)$, it follows that the *lval* is also L in any state type within (Γ'_2). Consequently, the *lval* is L in Γ_k, thus L in $(\gamma'_{g2}, \gamma_{l_k})$.

In the second configuration, we type the action's body s_2 with a H context, where s_2 reduces to (m'_{g2}, m'_{a2}). And since we previously showed that *lval* is L in $(\gamma'_{g2}, \gamma_{l_k})$ such that $(\gamma'_{g2}, \gamma_{l_k}) \vdash (m'_{g2}, m_{l2})$, then *lval* is in m'_{g2} or m_{l2}, however not in m'_{a2}.

Since *lval*'s type is L in γ'_{g2}, we can use Lemma E.10 to infer that the global initial and final states remain unchanged for L lvalues, which means $m_{g2}(lval) = m'_{g2}(lval)$. Then we can use Lemma E.12 to infer that $\mathrm{refine}(\gamma_2, \phi_k) \sqsubseteq (\gamma'_{g2}, \gamma'_{l2})$, this entails that the *lval*'s type label is indeed L in the global of refined state γ_{g_k}. It is easy to see that $\gamma_2 \sqsubseteq \mathrm{refine}(\gamma_2, \phi_k)$, thus $(\gamma_{g2}, \gamma_{l2}) \sqsubseteq (\gamma_{g_k}, \gamma_{l_k})$, therefore *lval*'s type is also L in the global initial state type γ_{g2}, i.e. $\gamma_{g2}(lval) = \tau'_2 \ \wedge \ \mathrm{lbl}(\tau'_2) = L$.

For the first configuration, in this sub-case, we have $\overline{\gamma'_1}(lval) = \tau'_1 \ \wedge \ \mathrm{lbl}(\tau'_1) = L$, and $\overline{\gamma'_1} \in \Gamma''_1$. We previously showed $\gamma'_1 \sqsubseteq \overline{\gamma'_1}$. Since *lval*'s typing label is L in $\overline{\gamma'_1}$ and we know that the state types in $\overline{\gamma'_1}$ are more restrictive than the state types in γ'_1, we can conclude that $\gamma'_1 \in \Gamma_j$ also types *lval* as L.

Now, we will prove that $m'_{g1} \underset{\overline{\gamma'_{g1}} \sqcup \overline{\gamma'_{g2}}}{\sim} m'_{g2}$ by conducting cases on ℓ_1:

case If ℓ_1 is H:

We can replicate the same exact steps done for the second configuration to deduct $m_{g1}(\mathit{lval}) = m'_{g1}(\mathit{lval})$ and $(\gamma_{g1}, \gamma_{l1}) \sqsubseteq (\gamma_{g_j}, \gamma_{l_j})$ and the *lval*'s type is L in the global initial state type γ_{g1}, i.e. $\gamma_{g1}(\mathit{lval}) = \tau'_2 \ \wedge \ \mathrm{lbl}(\tau'_2) = L$.

case If ℓ_1 is L:

Previously, we demonstrated that the *lval*'s typing label is L in all final state types in all Γ_i in Γ_2. Consequently, none of the actions' bodies can modify *lval*, this is true because all actions in the first and second semantics and the contracts are identical, therefore $m_{g1}(\mathit{lval}) = m'_{g1}(\mathit{lval})$. Thus, we know that indeed s_1 is typed under a high *pc* in the second configuration, we conclude that the *lval* remains unchanged there as well. Consequently, we can now apply Lemma E.13 to deduce that the *lval*'s typing label in the first configuration $\gamma'_1 \in \Gamma_j$ are more restrictive than the one we find in the refined state refine(γ_1, ϕ_j), and because $(\gamma_{g_j}, \gamma_{l_j}) = \mathrm{refine}(\gamma_1, \phi_j)$ then $(\gamma_{g_j}, \gamma_{l_j})(\mathit{lval}) = \tau'_1 \ \wedge \ \mathrm{lbl}(\tau'_1) = L$. Therefore, *lval*'s type is also L in the global initial state type γ_{g2}, i.e. $\gamma_{g2}(\mathit{lval}) = \tau'_2 \ \wedge \ \mathrm{lbl}(\tau'_2) = L$

Then, we need to prove $m_{l1} \underset{\overline{\gamma'_{l1}} \sqcup \overline{\gamma'_{l2}}}{\sim} m_{l2}$. First, we prove that $\overline{\gamma'_{l1}} = \gamma_{l_j}$ directly from the definition of join as $\mathrm{join}(\gamma_{l_j}, \gamma_{l_j}) = \gamma_{l_j}$ and we know $\mathrm{join}(\gamma_{l_j}, \gamma_{l_j}) = \overline{\gamma'_{l1}}$ thus $\overline{\gamma'_{l1}} = \gamma_{l_j}$ holds. Similarly, we know that $\overline{\gamma'_{l2}} = \gamma_{l_k}$ holds. Since *lval* is L in $\overline{\gamma'_{l2}}$ then it is also L in γ_{l_k}. And since *lval* is L in $\overline{\gamma'_{l2}}$, then it is also L in γ_{l_j}. From the previous subgoal, we proved $m_{l1} \underset{\gamma_{l_j} \sqcup \gamma_{l_k}}{\sim} m_{l2}$. This is property hold.

Finally, since we proved $m'_{g1} \underset{\overline{\gamma'_{g1}} \sqcup \overline{\gamma'_{g2}}}{\sim} m'_{g2}$ and $m_{l1} \underset{\overline{\gamma'_{l1}} \sqcup \overline{\gamma'_{l2}}}{\sim} m_{l2}$ now we can use Lemma E.15 to deduct that $(m'_{g1}, m_{l1}) \underset{\overline{\gamma'_1} \sqcup \overline{\gamma'_2}}{\sim} (m'_{g2}, m_{l2})$ holds.

◇ **Case transition**: similar to the conditional statement proof.

□

T-EmptyType

$$\frac{}{T, L, \bullet \vdash s : \Gamma}$$

T-Assign

$$\frac{\begin{array}{c} \gamma \vdash e : \tau \\ \tau' = \mathrm{raise}(\tau, pc) \\ \gamma' = \gamma[\mathit{lval} \mapsto \tau'] \end{array}}{T, pc, \gamma \vdash \mathit{lval} := e : \{\gamma'\}}$$

T-Call

$$\frac{\begin{array}{c} \gamma \vdash \vec{e} : \vec{\tau} \\ \mathrm{tCall}(T, f, pc, \vec{\tau}, \gamma, \Gamma) \end{array}}{T, pc, \gamma \vdash f(\vec{e}) : \Gamma}$$

T-Seq

$$\frac{\begin{array}{c} T, pc, \gamma \vdash s_1 : \Gamma_1 \\ \forall \gamma_1 \in \Gamma_1.\ T, pc, \gamma_1 \vdash s_2 : \Gamma_2^{\gamma_1} \\ \Gamma' = \bigcup_{\gamma_1 \in \Gamma_1} \Gamma_2^{\gamma_1} \end{array}}{T, pc, \gamma \vdash s_1; s_2 : \Gamma'}$$

T-Cond

$$\frac{\begin{array}{c} \gamma \vdash e : \tau \\ \ell = \mathrm{lbl}(\tau) \qquad pc' = pc \sqcup \ell \\ T, pc', (\mathrm{refine}(\gamma, e)) \vdash s_1 : \Gamma_1 \\ T, pc', (\mathrm{refine}(\gamma, \neg e)) \vdash s_2 : \Gamma_2 \\ \Gamma' = \Gamma_1 \cup \Gamma_2 \qquad \Gamma'' = \mathrm{joinOnHigh}(\Gamma', \ell) \end{array}}{T, pc, \gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Gamma''}$$

T-Table

$$\frac{\begin{array}{c} (\overline{e}, \mathrm{Cont}_{\mathrm{tbl}}) = T(\mathit{tbl}) \qquad \gamma \vdash e_i : \tau_i \\ \ell = \bigsqcup_i \mathrm{lbl}(\tau_i) \qquad pc' = pc \sqcup \ell \\ \forall(\phi_j, (a_j, \overline{\tau}_j)) \in \mathrm{Cont}_{\mathrm{tbl}}.\ \gamma_j = \mathrm{refine}(\gamma, \phi_j) \\ \mathrm{tCall}(T, a_j, pc', \overline{\tau}_j, (\gamma_g, \gamma_l), \Gamma_j) \end{array}}{T, pc, \gamma \vdash \texttt{apply}\ \mathit{tbl} : \mathrm{joinOnHigh}(\cup_j \Gamma_j, \ell)}$$

T-Trans

$$\frac{\begin{array}{c} \gamma \vdash e : \tau \qquad \ell = \mathrm{lbl}(\tau) \qquad pc' = pc \sqcup \ell \\ \gamma'_i = \mathrm{refine}(\gamma, e = v_i \wedge \bigwedge_{j<i} e \neq v_j) \qquad T, pc', \gamma'_i \vdash T(st_i) : \Gamma_i \\ \gamma'_d = \mathrm{refine}(\gamma, \bigwedge_i e \neq v_i) \qquad T, pc', \gamma'_d \vdash T(st) : \Gamma_d \\ \Gamma' = \Gamma_d \cup (\bigcup_i \Gamma_i) \qquad \Gamma'' = \mathrm{joinOnHigh}(\Gamma', \ell) \end{array}}{T, pc, \gamma \vdash \texttt{transition select}\, e\, \{v_1 : st_1, ..., v_n : st_n\}\ st : \Gamma''}$$

T-Extern

$$\frac{\begin{array}{c} (\gamma_g, \gamma_l) \vdash e_i : \tau_i \\ (\mathrm{Cont}_{\mathrm{E}}, (x_1, d_1), ..., (x_n, d_n)) = T(f) \\ \gamma_f = \{x_i \mapsto \tau_i\} \\ \forall(\gamma_i, \phi, \gamma_t) \in \mathrm{Cont}_{\mathrm{E}}.\ (\gamma_g, \gamma_f) \sqsubseteq \gamma_i \\ \Gamma' = \{\gamma' {+\!\!+} \mathrm{raise}(\gamma_t, pc) \mid\ (\gamma_i, \phi, \gamma_t) \in \mathrm{Cont}_{\mathrm{E}} \wedge \\ \mathrm{refine}((\gamma_g, \gamma_f), \phi) = \gamma' \neq \bullet\} \\ \Gamma'' = \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \mathrm{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\} \end{array}}{T, pc, (\gamma_g, \gamma_l) \vdash f(e_1, ..., e_n) : \Gamma''}$$

Figure E.8: Typing rules of statements

Bibliography

[1] D Elliot Bell, Leonard J LaPadula, et al. *Secure computer systems: Mathematical foundations*. Tech. rep. Citeseer, 1973.

[2] Butler W Lampson. "A note on the confinement problem". In: *Communications of the ACM* 16.10 (1973), pp. 613–615.

[3] Jerold Whitmore, Andre Bensoussan, and Paul Green. "Design for Multics security enhancements". In: *National Institute of Standards and Technology (US)*. ESD-TR-74-176. National Institute of Standards and Technology (US). 1973.

[4] Dorothy E Denning. "A lattice model of secure information flow". In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[5] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[6] Kenneth J Biba et al. "Integrity considerations for secure computer systems". In: (1977).

[7] Leslie Lamport. "Proving the correctness of multiprocess programs". In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[8] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *S&P*. 1982, pp. 11–20.

[9] Danny Dolev and Andrew Yao. "On the security of public key protocols". In: *IEEE Transactions on information theory* 29.2 (1983), pp. 198–208.

[10] United States. Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*. Vol. 83. 1. Department of Defense, 1987.

[11] Norm Hardy. "The Confused Deputy: (or why capabilities might have been invented)". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38.

[12] David FC Brewer and Michael J Nash. "The Chinese Wall Security Policy." In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 1989, pp. 206–214.

[13] Jaisook Landauer and Timothy Redmond. “A lattice of information”. In: *Proceedings Computer Security Foundations Workshop*. IEEE. 1993, pp. 65–70.

[14] Ravi S Sandhu and Pierangela Samarati. “Access control: principle and practice”. In: *IEEE communications magazine* 32.9 (1994), pp. 40–48.

[15] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1995.

[16] Ronald Fagin et al. *Reasoning about knowledge*. Cambridge, Mass.: MIT Press, 1995.

[17] William Murphy and Willy Hereman. “Determination of a position in three dimensions using trilateration and approximate distances”. In: *Department of Mathematical and Computer Sciences, Colorado School of Mines, Golden, Colorado, MCS-95* 7 (1995), p. 19.

[18] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. “A sound type system for secure flow analysis”. In: *Journal of computer security* 4.2-3 (1996), pp. 167–187.

[19] Andrew C. Myers. “JFlow: Practical Mostly-Static Information Flow Control”. In: *POPL*. San Antonio, Texas, USA, 1999, pp. 228–241.

[20] Andrew C Myers and Barbara Liskov. “Protecting privacy using the decentralized label model”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000), pp. 410–442.

[21] I. Kaplansky. *Set Theory and Metric Spaces*. AMS Chelsea Publishing, 2001.

[22] Heiko Mantel. “Information Flow Control and Applications - Bridging a Gap”. In: *FME*. Springer. 2001, pp. 153–172.

[23] Steve Zdancewic and Andrew C Myers. “Robust Declassification.” In: *csfw*. Vol. 1. 2001, pp. 15–23.

[24] Steve Zdancewic et al. “Untrusted Hosts and Confidentiality: Secure Program Partitioning”. In: *SOSP’01*. 2001, pp. 1–14.

[25] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge University Press, 2002.

[26] Latanya Sweeney. “k-anonymity: A model for protecting privacy”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-based Systems* 10.05 (2002), pp. 557–570.

[27] Steve Zdancewic et al. “Secure Program Partitioning”. In: *ACM Trans. Comput. Syst.* (Aug. 2002), pp. 283–328.

[28] Andrei Sabelfeld and Andrew C Myers. “Language-based information-flow security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19.

[29] Andrei Sabelfeld and Andrew C. Myers. “A Model for Delimited Information Release”. In: *ISSS*. Vol. 3233. Springer, 2003, pp. 174–191.

[30] Willem Visser et al. “Model checking programs”. In: *Autom. Softw. Eng.* 10.2 (2003), pp. 203–232.

[31] Lantian Zheng et al. “Using Replication and Partitioning to Build Secure Distributed Systems”. In: *S&P’03*. 2003, p. 236.

[32] Torben Amtoft and Anindya Banerjee. “Information flow analysis in logical form”. In: *International Static Analysis Symposium*. Springer. 2004, pp. 100–115.

[33] Roberto Giacobazzi and Isabella Mastroeni. “Abstract non-interference: Parameterizing non-interference by abstract interpretation”. In: *ACM SIGPLAN Notices* 39.1 (2004), pp. 186–197.

[34] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. “Enforcing robust declassification”. In: *CSF Workshop*. IEEE. 2004, pp. 172–186.

[35] Corina S Păsăreanu and Willem Visser. “Verification of Java programs using symbolic execution and invariant generation”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 2004, pp. 164–181.

[36] N. Vachharajani et al. “RIFLE: An Architectural Framework for User-Centric Information-Flow Security”. In: *37th International Symposium on Microarchitecture (MICRO-37’04)*. 2004, pp. 243–254.

[37] Steve Zdancewic. “Challenges for information-flow security”. In: *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID’04)*. Vol. 6. 2004.

[38] Elisa Bertino and Ravi Sandhu. “Database security-concepts, approaches, and challenges”. In: *IEEE Transactions on Dependable and Secure Computing* 2.1 (2005), pp. 2–19.

[39] Ádám Darvas, Reiner Hähnle, and David Sands. “A theorem proving approach to analysis of secure information flow”. In: *Security in Pervasive Computing: Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005. Proceedings 2*. Springer. 2005, pp. 193–209.

[40] Michael Hicks et al. “Dynamic updating of information-flow policies”. In: *Proc. of Foundations of Computer Security Workshop*. Vol. 20. 2005.

[41] Peng Li and Steve Zdancewic. “Practical information flow control in web-based information systems”. In: *18th IEEE Computer Security Foundations Workshop (CSFW’05)*. IEEE. 2005, pp. 2–15.

[42] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. “A logic for information flow in object-oriented programs”. In: *ACM SIGPLAN Notices* 41.1 (2006), pp. 91–102.

[43] Niklas Broberg and David Sands. “Flow locks: Towards a core calculus for dynamic flow policies”. In: *European Symposium on Programming*. Springer. 2006, pp. 180–196.

[44] Ezra Cooper et al. “Links: Web Programming without Tiers”. In: FMCO ’06. Springer-Verlag, 2006, pp. 266–296.

[45] Cynthia Dwork. “Differential privacy”. In: *Automata, Languages and Programming: 33rd International Colloquium*. Springer. 2006, pp. 1–12.

[46] Sebastian Hunt and David Sands. “On flow-sensitive security types”. In: vol. 41. 1. ACM New York, NY, USA, 2006, pp. 79–90.

[47] Gurvan Le Guernic et al. “Automata-based confidentiality monitoring”. In: *Annual Asian Computing Science Conference*. Springer. 2006, pp. 75–89.

[48] Manuel Serrano, Erick Gallesio, and Florian Loitsch. “Hop, A Language for Programming the Web 2.0”. In: OOPSLA Companion ’06. ACM, 2006.

[49] Nikhil Swamy et al. “Managing policy updates in security-typed languages”. In: *CSF Workshop*. IEEE. 2006, 13–pp.

[50] Sebastian Zander, Grenville Armitage, and Philip Branch. “Covert channels in the IP time to live field”. In: *Australian Telecommunication Networks and Application Conference (ATNAC) 2006*. 2006.

[51] A. Askarov and A. Sabelfeld. “Gradual Release: Unifying Declassification, Encryption and Key Release Policies”. In: *S&P*. 2007.

[52] Stephen Chong, Krishnaprasad Vikram, Andrew C Myers, et al. “SIF: Enforcing Confidentiality and Integrity in Web Applications”. In: *16th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2007, pp. 1–16.

[53] Stephen Chong et al. “Secure Web Applications via Automatic Partitioning”. In: *SOSP’07*. 2007, pp. 31–44.

[54] Nicoletta De Francesco and Luca Martini. “Instruction-level security analysis for information flow in stack-based assembly languages”. In: *Information and Computation* 205.9 (2007), pp. 1334–1370.

[55] Aslan Askarov et al. “Termination-insensitive noninterference leaks more than just a bit”. In: *ESORICS*. Springer. 2008, pp. 333–348.

[56] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. “Expressive Declassification Policies and Modular Static Enforcement”. In: *S&P*. 2008, pp. 339–353.

[57] David Clark and Sebastian Hunt. “Non-interference for deterministic interactive programs”. In: *FAST*. Springer. 2008, pp. 50–66.

[58] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer. 2008, pp. 337–340.

[59] Nikhil Swamy, Brian J Corcoran, and Michael Hicks. “Fable: A language for enforcing user-defined security policies”. In: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2008, pp. 369–383.

[60] ARM. *Building a Secure System using TrustZone® Technology*. 2009.

[61] Aslan Askarov and Andrei Sabelfeld. “Tight enforcement of information-release policies for dynamic languages”. In: *CSF*. IEEE. 2009, pp. 43–59.

[62] Thomas H Austin and Cormac Flanagan. “Efficient purely-dynamic information flow analysis”. In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. 2009, pp. 113–124.

[63] Musard Balliu and Isabella Mastroeni. “A weakest precondition approach to active attacks analysis”. In: *PLAS*. 2009, pp. 59–71.

[64] Niklas Broberg and David Sands. “Flow-sensitive semantics for dynamic information flow policies”. In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. 2009, pp. 101–112.

[65] Brian J Corcoran, Nikhil Swamy, and Michael Hicks. “Cross-tier, label-based security enforcement for web applications”. In: *ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 269–282.

[66] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. “Cross-tier, label-based security enforcement for web applications.” In: *SIGMOD*. 2009.

[67] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. “A Security-Preserving Compiler for Distributed Programs: From Information-Flow Policies to Cryptographic Mechanisms”. In: *CCS’09*. 2009, pp. 432–441.

[68] Ana Almeida Matos and Gérard Boudol. “On declassification and the non-disclosure policy”. In: *J. Comput. Secur.* 17.5 (2009), pp. 549–597.

[69] A. Sabelfeld and D. Sands. “Declassification: Dimensions and principles”. In: *JCS* (2009).

[70] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. “Predictive black-box mitigation of timing channels”. In: *ACM CCS*. 2010.

[71] Musard Balliu and Isabella Mastroeni. “A Weakest Precondition Approach to Robustness”. In: *Trans. Comput. Sci.* 10 (2010), pp. 261–297.

[72] Niklas Broberg and David Sands. “Paralocks: role-based information flow control and beyond”. In: *POPL*. 2010, pp. 431–444.

[73] Adam Chlipala. “Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications”. In: *9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 105–118.

[74] Michael R Clarkson and Fred B Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[75] Dominique Devriese and Frank Piessens. “Noninterference through secure multi-execution”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 109–124.

[76] Alan Nash, Luc Segoufin, and Victor Vianu. “Views and queries: Determinacy and rewriting”. In: *ACM Transactions on Database Systems (TODS)* 35.3 (2010), pp. 1–41.

[77] O. Purdila, L. A. Grijincu, and N. Tapus. “LKL: The Linux Kernel Library”. In: *9th RoEduNet IEEE International Conference*. 2010, pp. 328–333.

[78] Alejandro Russo and Andrei Sabelfeld. “Dynamic vs. static flow-sensitive security analysis”. In: *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE. 2010, pp. 186–199.

[79] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Hot Topics in Cloud Computing*. 2010, p. 10.

[80] Aslan Askarov and Andrew C. Myers. “Attacker Control and Impact for Confidentiality and Integrity”. In: *LMCS’11* (2011).

[81] Musard Balliu, Mads Dam, and Gurvan Le Guernic. “Epistemic Temporal Logic for Information Flow Security”. In: *PLAS*. 2011.

[82] G. Barthe, P. R. D’Argenio, and T. Rezk. “Secure information flow by self-composition”. In: *MSCS* (2011).

[83] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Relational verification using product programs”. In: *International Symposium on Formal Methods*. Springer. 2011, pp. 200–214.

[84] Gilles Barthe, Pedro R D’argenio, and Tamara Rezk. “Secure information flow by self-composition”. In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[85] Sebastian Hunt and David Sands. “From exponential to polynomial-time security typing via principal types”. In: *European Symposium on Programming*. Springer. 2011, pp. 297–316.

[86] Mauro Jaskelioff and Alejandro Russo. “Secure multi-execution in Haskell”. In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer. 2011, pp. 170–178.

[87] Deian Stefan et al. “Flexible dynamic information flow control in Haskell”. In: *SIGPLAN*. 2011, pp. 95–106.

[88] Aslan Askarov and Stephen Chong. “Learning is change in knowledge: Knowledge-based security for dynamic policies”. In: *CSF*. IEEE. 2012, pp. 308–322.

[89] Thomas H Austin and Cormac Flanagan. “Multiple facets for dynamic information flow”. In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2012, pp. 165–178.

[90] Musard Balliu, Mads Dam, and Gurvan Le Guernic. “Encover: Symbolic exploration for information flow security”. In: *IEEE 25th Computer Security Foundations Symposium (CSF)*. IEEE. 2012, pp. 30–44.

[91] Musard Balliu and Gurvan Le Guernic. *ENCoVer*. Software release. 2012. URL: http://www.nada.kth.se/~musard/encover.

[92] Willem De Groef et al. “FlowFox: a web browser with flexible and precise information flow control”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 748–759.

[93] Rayna Dimitrova et al. “Model checking information flow in reactive systems”. In: *Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings 13*. Springer. 2012, pp. 169–185.

[94] Daniel B Giffin et al. “Hails: Protecting data privacy in untrusted web applications”. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 47–60.

[95] Dimiter Milushev, Wim Beck, and Dave Clarke. “Noninterference via symbolic execution”. In: *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 152–168.

[96] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. “A language for automatically enforcing privacy policies”. In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 85–96.

[97] Thomas H Austin et al. “Faceted execution of policy-agnostic programs”. In: *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*. 2013, pp. 15–26.

[98] Musard Balliu. “A logic for information flow analysis of distributed programs”. In: *NordSec*. 2013.

[99] Gabriel M Bender et al. “Fine-grained disclosure control for app ecosystems”. In: *ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 869–880.

[100] Luísa Lourenço and Luís Caires. “Information flow analysis for valued-indexed data security compartments”. In: *International Symposium on Trustworthy Global Computing*. Springer. 2013, pp. 180–198.

[101] Frank McKeen et al. “Innovative Instructions and Software Model for Isolated Execution”. In: HASP ’13. 2013.

[102] Corina S Păsăreanu et al. “Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis”. In: *Automated Software Engineering* 20 (2013), pp. 391–425.

[103] Niki Vazou, Patrick M Rondon, and Ranjit Jhala. “Abstract refinement types”. In: *European Symposium on Programming*. Springer. 2013, pp. 209–228.

[104] Gabriel Bender, Lucja Kot, and Johannes Gehrke. “Explainable security for relational databases”. In: *ACM SIGMOD International Conference on Management of data*. ACM, 2014, pp. 1411–1422.

[105] Pat Bosshart et al. “P4: Programming protocol-independent packet processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[106] Agostino Cortesi and Raju Halder. “Information-flow analysis of hibernate query language”. In: *Future Data and Security Engineering: First International Conference, FDSE 2014, Ho Chi Minh City, Vietnam, November 19-21, 2014, Proceedings*. Springer. 2014, pp. 262–274.

[107] Zakir Durumeric et al. “The matter of heartbleed”. In: *Proceedings of the 2014 conference on internet measurement conference*. 2014, pp. 475–488.

[108] Intel. *Intel Software Guard Extensions Developer Guide*. Accessed 2021-05-20. 2014. URL: https://software.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-developer-guide.pdf.

[109] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. “SeLINQ: tracking information across application-database boundaries”. In: *19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2014, pp. 25–38.

[110] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. “SeLINQ: tracking information across application-database boundaries”. In: *ICFP*. 2014.

[111] Gregor Snelting et al. “Checking probabilistic noninterference using JOANA”. In: *it Inf. Technol.* 56 (2014), pp. 280–287.

[112] Niki Vazou et al. “Refinement types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 2014, pp. 269–282.

[113] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *ACM Transactions on Computer Systems (TOCS)* (2015), p. 8.

[114] Niklas Broberg, Bart van Delft, and David Sands. “The anatomy and facets of dynamic policies”. In: *CSF*. IEEE. 2015, pp. 122–136.

[115] Adam Chlipala. “Ur/Web: A Simple Model for Programming the Web”. In: *POPL ’15*. Association for Computing Machinery, 2015, pp. 153–165. ISBN: 9781450333009.

[116] Bart van Delft, Sebastian Hunt, and David Sands. “Very static enforcement of dynamic policies”. In: *International Conference on Principles of Security and Trust*. Springer. 2015, pp. 32–52.

[117] Luísa Lourenço and Luís Caires. “Dependent information flow types”. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015, pp. 317–328.

[118] Jon Matias et al. “Toward an SDN-enabled NFV architecture”. In: *IEEE Communications Magazine* 53.4 (2015), pp. 187–193. DOI: 10.1109/MCOM.2015.7081093.

[119] Rohit Sinha et al. “Moat: Verifying Confidentiality of Enclave Programs”. In: *CCS’15*. 2015, pp. 1169–1184.

[120] José Bacelar Almeida et al. “Verifying Constant-Time Implementations”. In: *USENIX Security*. 2016.

[121] Sergei Arnautov et al. “SCONE: Secure Linux Containers with Intel SGX”. In: *OSDI’16*. 2016, pp. 689–703.

[122] Musard Balliu et al. “Jslinq: Building secure applications across tiers”. In: *6th ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 307–318.

[123] Nataliia Bielova and Tamara Rezk. “Spot the difference: Secure multi-execution and multiple facets”. In: *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*. Springer. 2016, pp. 501–519.

[124] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086. 2016.

[125] Victor Costan, Ilia Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: 2016, pp. 857–874.

[126] Anitha Gollamudi and Stephen Chong. “Automatic Enforcement of Expressive Security Policies Using Enclaves”. In: *OOPSLA’16*. 2016, pp. 494–513.

[127] Paul Goransson, Chuck Black, and Timothy Culver. *Software defined networks: a comprehensive approach*. Morgan Kaufmann, 2016.

[128] Marco Guarnieri, Srdjan Marinovic, and David Basin. “Strong and provably secure database access control”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 163–178.

[129] David Kaplan, Jeremy Powell, and Tom Woller. “AMD Memory Encryption”. In: (2016).

[130] Rohit Sinha et al. “A Design and Verification Methodology for Secure Isolated Regions”. In: *PLDI’16*. 2016, pp. 665–681.

[131] Jean Yang et al. “Precise, dynamic information flow for database-backed applications”. In: *ACM Sigplan Notices* 51.6 (2016), pp. 631–647.

[132] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. “We Are Family: Relating Information-Flow Trackers”. In: *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Vol. 10492. Lecture Notes in Computer Science. 2017, pp. 124–145.

[133] Niklas Broberg, Bart van Delft, and David Sands. “Paragon - Practical programming with information flow control”. In: *J. Comput. Secur.* 25 (2017), pp. 323–365.

[134] Ethan Cecchetti, Andrew C Myers, and Owen Arden. “Nonmalleable information flow control”. In: *CCS*. 2017, pp. 1875–1891.

[135] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. “Automatic detection and demonstrator generation for information flow leaks in object-oriented programs”. In: *computers & security* 67 (2017), pp. 335–349.

[136] Peixuan Li and Danfeng Zhang. “Towards a Flow- and Path-Sensitive Information Flow Analysis”. In: *30th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 53–67.

[137] Joshua Lind et al. “Glamdring: Automatic Application Partitioning for Intel SGX”. In: *USENIX ATC’17*. 2017, pp. 285–298.

[138] Jed Liu et al. “Fabric: Building open distributed systems securely by construction”. In: *Journal of Computer Security* (2017), pp. 367–426.

[139] Shen Liu, Gang Tan, and Trent Jaeger. “PtrSplit: Supporting General Pointers in Automatic Program Partitioning”. In: *CCS’17*. 2017, pp. 2359–2371.

[140] Aastha Mehta et al. “Qapla: Policy compliance for database-backed systems”. In: *26th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2017, pp. 1463–1479.

[141] Checkpoint Research. *EternalBlue: Everything You Need to Know*. Accessed: 2024-12-03. 2017. URL: https://research.checkpoint.com/2017/eternalblue-everything-know/.

[142] Shweta Shinde et al. “Panoply: Low-TCB Linux Applications With SGX Enclaves”. In: *NDSS’17*. 2017.

[143] R. Silva, P. Barbosa, and A. Brito. “DynSGX: A Privacy Preserving Toolset for Dinamically Loading Functions into Intel(R) SGX Enclaves”. In: *CloudCom’17*. 2017, pp. 314–321.

[144] Rohit Sinha. “Secure Computing using Certified Software and Trusted Hardware”. PhD thesis. 2017.

[145] Deian Stefan et al. “Flexible dynamic information flow control in the presence of exceptions”. In: *Journal of Functional Programming* 27 (2017), e5.

[146] Pramod Subramanyan et al. “A Formal Foundation for Secure Remote Execution of Enclaves”. In: *CCS’17*. 2017, pp. 2435–2450.

[147] Hongliang Tian et al. “SGXKernel: A Library Operating System Optimized for Intel SGX”. In: *Proceedings of the Computing Frontiers Conference*. 2017, pp. 35–44.

[148] Chia-Che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *USENIX ATC’17*. 2017, pp. 645–658.

[149] Pascal Weisenburger et al. "Quality-Aware Runtime Adaptation in Complex Event Processing". In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* 2017, pp. 140–151.

[150] Abbas Acar et al. "A survey on homomorphic encryption schemes: Theory and implementation". In: *ACM Computing Surveys (Csur)* 51.4 (2018), pp. 1–35.

[151] Kalev Alpernas et al. "Abstract interpretation of stateful networks". In: *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25.* Springer. 2018, pp. 86–106.

[152] Andrey Chudnov and David A Naumann. "Assuming you know: Epistemic semantics of relational annotations for expressive flow policies". In: *CSF.* IEEE. 2018, pp. 189–203.

[153] Lucas Freire et al. "Uncovering bugs in P4 programs with assertion-based verification". In: *Proceedings of the Symposium on SDN Research.* 2018, pp. 1–7.

[154] Andres Nötzli et al. "P4pktgen: Automated test case generation for P4 programs". In: *Proceedings of the Symposium on SDN Research.* 2018, pp. 1–7.

[155] Radu Stoenescu et al. "Debugging P4 programs with Vera". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* 2018, pp. 518–532.

[156] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. "Distributed System Development with ScalaLoci". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). URL: https://doi.org/10.1145/3276499.

[157] Musard Balliu, Iulia Bastys, and Andrei Sabelfeld. "Securing IoT Apps". In: *IEEE Security & Privacy Magazine* (2019).

[158] Ryan Beckett et al. "Abstract interpretation of distributed network control planes". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–27.

[159] Thomas Bourgeat et al. "MI6: Secure Enclaves in a Speculative Out-of-Order Processor". In: *MICRO'52.* 2019, pp. 42–56.

[160] Luigi Coppolino et al. "A Comparative Analysis of Emerging Approaches for Securing Java Software with Intel SGX". In: *Future Generation Computer Systems* (2019), pp. 620–633.

[161] Adrien Ghosn, James R. Larus, and Edouard Bugnion. "Secured Routines: Language-Based Construction of Trusted Execution Environments". In: *USENIX ATC'19.* 2019, pp. 571–585.

[162] A. Gollamudi, S. Chong, and O. Arden. "Information Flow Control for Distributed Trusted Execution Environments". In: *CSF'19.* 2019, pp. 304–30414.

[163] Marco Guarnieri et al. "Information-flow control for database-backed applications". In: *IEEE European Symposium on Security and Privacy (EuroS&P).* IEEE. 2019, pp. 79–94.

[164] Shen Liu et al. "Program-mandering: Quantitative Privilege Separation". In: *CCS'19.* 2019, pp. 1023–1040.

[165] James Parker, Niki Vazou, and Michael Hicks. "LWeb: information flow security for multi-tier web applications". In: *Proc. ACM Program. Lang.* 3.POPL'19 (2019), 75:1–75:30.

[166] Christian Priebe et al. *SGX-LKL: Securing the Host OS Interface for Trusted Execution.* 2019.

[167] Huibo Wang et al. "Towards Memory Safe Enclave Programming with Rust-SGX". In: *CCS'19.* 2019, pp. 2333–2350.

[168] Apple. *Apple Platform Security.* https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf. Accessed 2021-05-20. 2020.

[169] Juan Camilo Correa Chica, Jenny Cuatindioy Imbachi, and Juan Felipe Botero Vega. "Security in SDN: A comprehensive survey". In: *Journal of Network and Computer Applications* 159 (2020), p. 102595.

[170] Jianyu Jiang et al. "Uranus: Simple, Efficient SGX Programming and its Applications". In: *ASIA CCS'20.* 2020.

[171] Elisavet Kozyri and Fred B Schneider. "RIF: Reactive information flow labels". In: *J. Comput. Secur.* 28 (2020), pp. 191–228.

[172] Dayeol Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *EuroSys'20.* 2020.

[173] Yi Lu and Chenyi Zhang. "Nontransitive security types for coarse-grained information flow control". In: *CSF.* IEEE. 2020, pp. 199–213.

[174] Meni Orenbach, Andrew Baumann, and Mark Silberstein. "Autarky: Closing Controlled Channels with Self-Paging Enclaves". In: *EuroSys'20.* 2020.

[175] Sandro Pinto and Cesare Garlati. *Multi Zone Security for Arm Cortex-M Devices.* https://hex-five.com/wp-content/uploads/2020/02/Multi-Zone-Security-White-Paper-20200224.pdf. 2020.

[176] Nadia Polikarpova et al. "Liquid information flow control". In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–30.

[177] Youren Shen et al. "Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX". In: *ASPLOS'20.* 2020, pp. 955–970.

[178] Chia-Che Tsai et al. "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves". In: *USENIX Security'20*. 2020.

[179] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. "A Survey of Multitier Programming". In: *ACM Comput. Surv.* (Sept. 2020).

[180] Ryan Doenges et al. "Petr4: formal foundations for P4 data planes". In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–32.

[181] Sebastian Hunt and David Sands. "A Quantale of Information". In: *IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE. 2021, pp. 1–15.

[182] Ranjit Jhala, Niki Vazou, et al. "Refinement types: A tutorial". In: *Foundations and Trends® in Programming Languages* 6.3–4 (2021), pp. 159–317.

[183] Nico Lehmann et al. "STORM: Refinement types for secure web applications". In: *15th USENIX Symposium on Operating Systems Design and Implementation OSDI 21)*. 2021, pp. 441–459.

[184] Peixuan Li and Danfeng Zhang. "Towards a General-Purpose Dynamic Information Flow Policy". In: *arXiv preprint* (2021).

[185] Aditya Oak et al. "Language Support for Secure Software Development with Enclaves". In: 2021.

[186] Aditya Oak et al. "Language Support for Secure Software Development with Enclaves". In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE. 2021, pp. 1–16. DOI: 10.1109/CSF51468.2021.00037.

[187] Kausik Subramanian et al. "D2r: Policy-compliant fast reroute". In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 2021, pp. 148–161.

[188] Amir M Ahmadian and Musard Balliu. *DynCoVer*. Software release. Mar. 2022. URL: https://github.com/amir-ahmadian/jpf-dyncover.

[189] Kinan Dak Albab et al. "SwitchV: automated SDN switch validation with P4 models". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 365–379.

[190] Matthias Eichholz et al. "Dependently-typed data plane programming". In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), pp. 1–28.

[191] Douglas Everson, Long Cheng, and Zhenkai Zhang. "Log4shell: Redefining the web attack surface". In: *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*. 2022, pp. 1–8.

[192] Karuna Grewal, Loris D'Antoni, and Justin Hsu. "P4BID: information flow control in P4". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 46–60.

[193] Qichen Wang and Ke Yi. “Conjunctive Queries with Comparisons”. In: *International Conference on Management of Data*. ACM, 2022, pp. 108–121.

[194] Amir M. Ahmadian, Matvey Soloviev, and Musard Balliu. *DiVerT*. Software release. 2023. URL: https://github.com/KTH-LangSec/DiVerT.

[195] Fabian Ruffy et al. “P4Testgen: An Extensible Test Oracle For P4-16”. In: *Proceedings of the ACM SIGCOMM 2023 Conference*. 2023, pp. 136–151.

[196] Ying Yao et al. “Scaver: A Scalable Verification System for Programmable Network”. In: *Proceedings of the 2024 SIGCOMM Workshop on Formal Methods Aided Network Operation*. 2024, pp. 14–19.

[197] Anjuna. *Anjuna Enterprise Enclaves*. https://www.anjuna.io/enterprise-enclaves. Accessed 2021-05-20.

[198] Fortanix. *The Fortanix Runtime Encryption*. https://fortanix.com/products/runtime-encryption. Accessed 2021-05-20.

[199] GraalVM. *GraalVM Native Image*. https://www.graalvm.org/docs/reference-manual/native-image. Accessed 2021-05-20.

[200] JavaParser. *JavaParser*. https://javaparser.org. Accessed 2021-08-24.

[201] *Lark Parser*. URL: https://github.com/lark-parser/lark.

[202] Oracle. *Java Native Interface*. https://docs.oracle.com/javase/8/docs/technotes/guides/jni. Accessed 2021-05-20.

[203] Oracle. *Java Remote Method Invocation - Distributed Computing for Java*. https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html. Accessed 2021-05-20.

[204] SGX-LKL. *SGX-LKL Library OS for Running Linux Applications Inside of Intel SGX Enclaves*. https://github.com/lsds/sgx-lkl. Accessed 2021-05-20.