Securing P4 Programs by Information Flow Control

Anoud Alshnakat*, Amir M. Ahmadian*, Musard Balliu*, Roberto Guanciale* and Mads Dam*
*KTH Royal Institute of Technology

***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 that combines security types with interval analysis. The proposed type system allows the specification of security policies in terms of input and output packet bit fields rather than program variables. We formalize this type system and prove it sound, guaranteeing that well-typed programs satisfy noninterference. Our prototype implementation, TAP4S, is evaluated on several use cases, demonstrating its effectiveness in detecting security violations and information leakages.

I. INTRODUCTION

Software-Defined Networking (SDN) [1] 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 the setting of 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 [2]. 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) [3] 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 is detailed enough to accurately capture the behavior of the switch. It maintains a level of simplicity and formalism that allows for effective automated analysis [4].

NFVs and SDNs introduce new security challenges that extend beyond the famous and costly outages caused by network misconfigurations [5]. 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 a 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 numbers and TTL fields [6].

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. In the programming languages area, current approaches (including P4BID [7]) 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 subnetwork segments, port ranges, and non-expired TTLs. Therefore, in our approach, in addition to a security label, the security type also keeps track of an interval.

The 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 a manner reminiscent of dynamic information flow control [8] and symbolic execution, cf. [9]. 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 engage 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 a prototype tool TAP4S [10] and evaluate the tool on a test suite and 5 use cases.

II. 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.

We use Program 1 as a running example throughout the paper. The program implements a switch that manages congestion in the network of Fig. 1. In an IPv4 packet, the Explicit Congestion Notification (ECN) field provides the status of congestion experienced by switches while transmitting the packet along the path from source to destination. ECN value 0 indicates that at least one of the traversed switches does not support the ECN capability, values 1 and 2 indicate that all traversed switches support ECN and the packet can be marked if congestion occurs, and 3 indicates that the packet has experienced congestion in at least one of the switches.

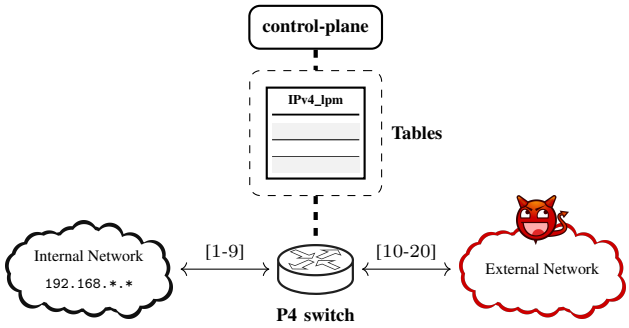

Fig. 1: *Congestion notifier* network layout

We assume for this example that the switch is the *only* ingress and egress point for traffic entering and exiting the internal network, connecting it to external networks as shown in Fig. 1. To illustrate our approach, we assume that the switch is designed to prevent any information leakage about internal network congestion to the external network. In addition to standard packet forwarding, the switch sets ECN to 3 if its queue length exceeds a predefined threshold. This holds only if the packet's destination is within the internal network. Conversely, if the packet is destined to the external network, the switch sets the ECN field to 0. This indicates that ECN is not supported for outbound traffic and ensures that congestion signals experienced within the internal network are not exposed externally.[1]

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 53) 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 Fig. 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), and source and destination IP addresses.

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

[1]In scenarios with multiple ingress and egress points, where external traffic may fully traverse the internal network, the identification of outbound packets cannot rely solely on IP addresses. Instead, classification would need to be based on the forwarding port to allow the use of the ECN field while the packet is inside the internal network.

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 1 starts at line 34. If the IPv4 header is not valid (line 53) the packet is dropped. Otherwise, if the packet's destination (line 54-56) 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. 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.

A. 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 1, additionally to the ECN, 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.

Programming errors and misconfigurations can cause information leakage. Consider the application of the ipv4_lpm table (line 61) which forwards the packet to a table-specified port. A bug in the branch condition on line 54, which checks the least significant bits of the dstAddr (e.g. by mistakenly checking hdr.ipv4.dstAddr[7:0] == 192 instead), would result in setting the ecn field on the packets leaving the internal network, thus causing the packets forwarded to an external network to 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 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 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.

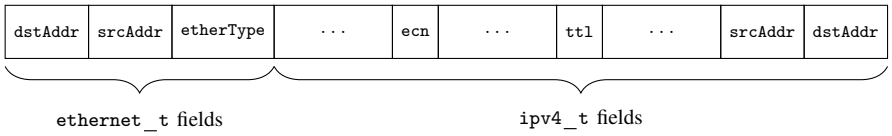

Fig. 2: Packet header

```
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 &&
          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 1: Congestion notifier

> *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 design 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.

III. 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 $\langle$`192.168.0.0, 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:

$$\begin{aligned}
\texttt{hdr.eth.etherType} &: (\langle 0\text{x}0800, 0\text{x}0800\rangle, L)\\
\texttt{hdr.ipv4.srcAddr} &: (\langle \texttt{192.168.0.0, 192.168.255.255}\rangle, L)\\
\texttt{hdr.ipv4.ecn} &: (\langle * \rangle, H)\\
\texttt{standard_metadata.enq_qdepth} &: (\langle * \rangle, H)
\end{aligned}$$

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 the range $\langle$`192.168.0.0, 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 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 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 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 branch 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.

We ensure that the program does not leak sensitive information by checking the final types produced by the type system against an output policy. If these checks succeed, 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 VI.

IV. 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 Fig. 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

$$v ::= \overline{b} \mid \{f_1 = v_1, ..., f_n = v_n\}$$
$$e ::= v \mid x \mid \ominus e \mid e \oplus e' \mid e.f \mid e[\overline{b} : bitv'] \mid \{f_1 = e_1, ..., f_n = e_n\}$$
$$lval ::= x \mid lval.f \mid lval[\overline{b} : \overline{b}']$$
$$s ::= \texttt{skip} \mid lval := e \mid s_1; s_2 \mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \mid \texttt{apply } tbl \mid f(e_1, ..., e_n) \mid \texttt{transition select } e \{v_1 : st_1, ..., v_n : st_n\}\ st$$

Fig. 3: Syntax

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 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 Fig. 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 Fig. 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.

For example, in Program 1 let $m_l = \{\texttt{hdr.ipv4.ttl} \mapsto 2\}$ when invoking `decrease` at line 44. The local state of

$$\text{S-SKIP}\ \frac{}{E : m \xrightarrow{\texttt{skip}} m} \qquad \text{S-ASSIGN}\ \frac{m' = m[lval \mapsto m(e)]}{E : m \xrightarrow{lval := e} m'} \qquad \text{S-SEQ}\ \frac{E : m \xrightarrow{s_1} m' \quad E : m' \xrightarrow{s_2} m''}{E : m \xrightarrow{s_1; s_2} m''}$$

$$\text{S-COND-T}\ \frac{m(e) = \text{true} \quad E : m \xrightarrow{s_1} m'}{E : m \xrightarrow{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} m'} \qquad \text{S-COND-F}\ \frac{m(e) = \text{false} \quad E : m \xrightarrow{s_2} m'}{E : m \xrightarrow{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2} m'}$$

$$\text{S-CALL}\ \frac{(s, \overline{(x, d)}) = E(f) \qquad m_l' = \{x_i \mapsto (m_g, m_l)(e_i)\} \qquad E : (m_g, m_l') \xrightarrow{s} (m_g', m_l'')}{E : (m_g, m_l) \xrightarrow{f(e_1, ..., e_n)} (m_g', m_l)[e_i \mapsto m_l''(x_i) \mid \text{isOut}(d_i)]}$$

$$\text{S-EXTERN}\ \frac{(sem_f, \overline{(x, d)}) = E(f) \qquad m_l' = \{x_i \mapsto (m_g, m_l)(e_i)\} \qquad (m_g', m_l'') = sem_f(m_g, m_l')}{E : m \xrightarrow{f(e_1, ..., e_n)} (m_g', m_l)[e_i \mapsto m_l''(x_i) \mid \text{isOut}(d_i)]}$$

$$\text{S-TRANS}\ \frac{st' = \begin{cases} st_i & \text{if } m(e) = v_i \\ st & \text{otherwise} \end{cases} \qquad E : m \xrightarrow{E(st')} m'}{E : m \xrightarrow{\texttt{transition select } e \{v_1 : st_1, ..., v_n : st_n\}\ st} m'}$$

$$\text{S-TABLE}\ \frac{\begin{array}{c}(\overline{e}, sem_{tbl}) = E(tbl) \\ sem_{tbl}((m_g, m_l)(e_1), ..., (m_g, m_l)(e_n)) = (a, \overline{v}) \\ (s, (x_1, \text{none}), ..., (x_n, \text{none})) = E(a) \\ m_l' = \{x_i \mapsto v_i\} \qquad E : (m_g, m_l') \xrightarrow{s} (m_g', m_l'')\end{array}}{E : (m_g, m_l) \xrightarrow{\texttt{apply } tbl} (m_g', m_l)}$$

Fig. 4: Semantic rules

`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 $m(\texttt{hdr.eth.etherType}) = 0\text{x}0800$ on line 23 of Program 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 $m(\texttt{hdr.ipv4.dstAddr}) = 192.168.2.2$ 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`.

V. 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 bitvector 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 label $\ell \in \{L, H\}$. We use the shorthand I_i^ℓ 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 $\langle f_1 : \tau_1; ...; f_n : \tau_n \rangle$ 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, written as $\gamma \vdash m$, if for every $lval$ in the domain of m, the value $m(lval)$ belongs to the interval specified by $\gamma(lval)$; formally, $\forall\ lval \in \mathrm{domain}(m),\ \ m(lval) : \gamma(lval)$. Note that the typing judgment $\gamma \vdash m$ is based on the interval inclusion and it is independent of any security labels. For example, if $m(x) = 257$ then 257 is considered well-typed wrt. γ if and only if $\gamma(x)$ is an interval that contains 257 (e.g., $\langle 0, 257 \rangle$, $\langle 257, 257 \rangle$, or $\langle 100, 300 \rangle$).

A state type might include a type with an empty interval; we call this state type *empty* and denote it as $\bullet$.

Let $\mathrm{lblOf}(lval, \gamma)$ be the least upper bound of the security labels of all the slices 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 $\mathrm{lblOf}(lval, \gamma) = L$, then $m_1(lval) = m_2(lval)$ holds.

Example 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 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 Fig. 5. This contract models a table that forwards the packets with `hdr.ipv4.dstAddr` = `192.*.*.*` to ports 1-9, the ones with `hdr.ipv4.dstAddr` = `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.

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 $(\overline{\tau})$.

$$
\begin{array}{l}
(\ [\texttt{hdr.ipv4.dstAddr}], \\
\{(\texttt{dstAddr}[31:24] = 192, (\texttt{ipv4\_forward}, [\langle * \rangle^{H}_{48}, \langle 1, 9 \rangle^{L}_{9}])) \\
\ (\texttt{dstAddr}[31:24] = 10, (\texttt{ipv4\_forward}, [\langle * \rangle^{L}_{48}, \langle 10, 20 \rangle^{L}_{9}])) \\
\ (\texttt{dstAddr}[31:24] \neq 192 \wedge \texttt{dstAddr}[31:24] \neq 10, (\texttt{drop}, \ [\,]))\})
\end{array}
$$

Fig. 5: The contract of `ipv4_lpm` table

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 $\mathrm{lbl}(\tau_i) = L$ then $v_{1_i} = v_{2_i}$. Note that $\mathrm{lbl}(\tau_i)$ 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, $\mathrm{lbl}(\tau_i)$ 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 $(\overline{e}_1, sem_{1_{tbl}}) = E_1(tbl)$, $(\overline{e}_2, sem_{2_{tbl}}) = E_2(tbl)$, $(\overline{e}, \mathrm{Cont}_{tbl}) = T(tbl)$ then $\overline{e}_1 = \overline{e}_2 = \overline{e}$, and for all $(\phi, (a, \overline{\tau})) \in \mathrm{Cont}_{tbl}$, and for all arbitrary states m_1 and m_2, such that $m_1(\overline{e}) = m_2(\overline{e}) = \overline{v}$ and $m_1(\phi) \Leftrightarrow m_2(\phi)$, if $m_1(\phi)$ then $\overline{v}_1, \overline{v}_2$ exist 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 explained in Section III the input and output policy cases are expressed by assigning types to program variables. 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 1 (Noninterference)**.** A program s is *noninterfering* wrt. the input and output policy cases γ_i and γ_o, and table contract mapping T, if for all mappings E_1, E_2 and states m_1, m_2, m_1' such that:

- $E_1 \underset{T}{\sim} E_2$,
- $\gamma_i \vdash m_1$, $\gamma_i \vdash m_2$, and $m_1 \underset{\gamma_i}{\sim} m_2$,
- $E_1 : m_1 \xrightarrow{s} m_1'$

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 $m_1' \underset{\gamma_o}{\sim} m_2'$.

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 accepts the cases where the program terminates for both initial states m_1 and m_2.

Intuitively, Definition 1 relies on two different equivalence relations: one induced by the input policy case and one by the output policy case. The former induces a partial equivalence relation (PER) [11], $P_{\gamma_i}(m_1, m_2) = \gamma_i \vdash m_1 \wedge \gamma_i \vdash m_2 \wedge m_1 \underset{\gamma_i}{\sim} m_2$, such that the domain contains only states that satisfy the intervals of γ_i. Similarly, the latter induces $Q_{\gamma_o}(m_1, m_2) = (\gamma_o \vdash m_1 \wedge \gamma_o \vdash m_2 \wedge m_1 \underset{\gamma_i}{\sim} m_2) \vee (\gamma_o \nvdash m_1 \wedge \gamma_o \nvdash m_2)$, which is an equivalence relation (ER). A program is then noninterfering wrt. γ_i and γ_o if every class of the PER P_{γ_i} is mapped to a class of the ER Q_{γ_o}.

This condition implies the following intuitive assumptions: (1) the policy cases are public knowledge, (2) entailment of a state on the intervals of a policy case is public knowledge, (3) the states that do not entail the intervals of an input policy (i.e., those outside the domain of the PER) are considered entirely public, and their corresponding execution is unconstrained, (4) the attacker can observe whether the final state entails the intervals of the output policy, and (5) the attacker cannot observe any additional information about states that do not entail the intervals of the output policy. We use the following examples to further discuss our security condition.

Example 2. Consider the input and output policy cases:

$$\gamma_i = \{a \mapsto \langle 0, 256 \rangle^{H}_{9},\ b \mapsto \langle * \rangle^{L}_{9}\}$$
$$\gamma_o = \{a \mapsto \langle * \rangle^{H}_{9},\ b \mapsto \langle 1025, * \rangle^{L}_{9}\}$$

- The inputs $a_1 = 257$ and $a_2 = 258$ are distinguishable by the attacker, since they do not fall in the H interval $\langle 0, 256 \rangle$ under γ_i.
- The inputs $a_1 = 0$ and $a_2 = 256$ are indistinguishable, since they belong to the H interval $\langle 0, 256 \rangle$ under γ_i.
- Under γ_o, any value $b \geq 1025$ is distinguishable by the attacker, otherwise it is indistinguishable since it falls outside the L interval $\langle 1025, * \rangle$.

We consider pairs of states m_1, m_2 such that $\gamma_i \vdash m_1$, $\gamma_i \vdash m_2$, and $m_1 \underset{\gamma_i}{\sim} m_2$. For example,

1) $m_1(a) = a_1$ and $m_2(a) = a_2$, where $a_1\ ,\ a_2 \in \langle 0, 256 \rangle$.
2) $m_1(b) = m_2(b) = b_0$.

We use the above policies and states to discuss the security condition of the following one-line programs:

- `b=a`
  This program yields $m_1'(b) = a_1$ and $m_2'(b) = a_2$. Since $m_1'(b) \notin \langle 1025, * \rangle$, then $\gamma_o \nvdash m_1'$, hence noninterference is trivially satisfied. Intuitively, despite the variable a being H, the output on the variable b is not observable by the attacker.
- `if (a<=1024) then b=a else skip`
  This program yields $m_1'(b) = a_1$ and $m_2'(b) = a_2$. Since both a_1 and a_2 are in $\langle 0, 256 \rangle$, the program executes b=a

in the true branch. Thus, $\gamma_o \not\vdash m_1'$ and noninterference is trivially satisfied.

- `b=a+1000`
  (i) Let $a_1 = 25$ and $a_2 = 26$. Then $m_1'(b) = 1025$ and $m_2'(b) = 1026$ indicating $\gamma_o \vdash m_1'$ and $\gamma_o \vdash m_2'$, however $m_1' \not\underset{\gamma_o}{\sim} m_2'$, hence the program is interfering.
  (ii) Let $a_1 = 25$ and $a_2 = 0$. Then $m_1'(b) = 1025$ and $m_2'(b) = 1000$ indicating $\gamma_o \vdash m_1'$ and $\gamma_o \not\vdash m_2'$, hence the program is interfering.

Example 3. 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.

VI. 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.

A. 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.

$$\textsc{T-SingleSliceBs}\quad \frac{\gamma \vdash e_1 : \langle I_1 \rangle_i^{\ell_1} \qquad \gamma \vdash e_2 : \langle I_2 \rangle_i^{\ell_2}}{\gamma \vdash e_1 \oplus e_2 : \langle I_1 \oplus I_2 \rangle_i^{\ell_1 \sqcup \ell_2}} \qquad \frac{\gamma \vdash e_1 : \langle I_1 \rangle_i^{\ell_1}}{\gamma \vdash \ominus e_1 : \langle \ominus I_1 \rangle_i^{\ell_1}}$$
$$\gamma \vdash e_1 \otimes e_2 : \langle I_1 \otimes I_2 \rangle_i^{\ell_1 \sqcup \ell_2}$$

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.

$$\textsc{T-AlignedSlice}\quad \frac{\gamma \vdash e : \langle I_n \rangle_{i_n}^{\ell_n} \cdots \langle I_1 \rangle_{i_1}^{\ell_1}}{\gamma \vdash e[\sum_{j=1}^{b} i_j : \sum_{j=1}^{a} i_j] : \langle I_b \rangle_{i_b}^{\ell_b} \cdots \langle I_a \rangle_{i_a}^{\ell_a}}$$

$$\textsc{T-NonAlignedSlice}\quad \frac{\gamma \vdash e : \langle I_n \rangle_{i_n}^{\ell_n} \cdots \langle I_1 \rangle_{i_1}^{\ell_1}}{\gamma \vdash e[b : a] : \langle * \rangle_{b-a}^{\bigsqcup \ell_{i_j}}}$$

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:

Lemma 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 $\mathrm{lbl}(\tau) = \mathrm{L}$, then $m'(e) = v$.*

B. 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 the full version of the paper [12].

$\gamma[lval \mapsto \tau]$ indicates updating the type of *lval*, which can be a part of a variable, in state type γ. $\gamma \uplus \gamma'$ 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 4. Let x be mapped to an interval between 2 and 8, or in binary, bitvectors between $\boxed{0}\boxed{0}\boxed{1}\boxed{0}$ and $\boxed{1}\boxed{0}\boxed{0}\boxed{0}$, 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 $\boxed{0}\boxed{1}\boxed{0}$ and $\boxed{0}\boxed{0}\boxed{0}$, that is every 3-bit value except $\boxed{0}\boxed{0}\boxed{1}$. Such value set cannot be represented by a single continuous interval, hence we overapproximate to the complete interval $\langle * \rangle$.

Similarly, the operation $\text{refine}(\gamma, x[3:3] < 1)$ 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 $\text{join}(\gamma, \{x \mapsto \langle * \rangle_1^H \cdot \langle * \rangle_3^L\})$ 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 V, 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 meet all the output policy cases γ_o in Γ.

In the following rules, we use $\text{raise}(\tau, \ell)$ to return a type where each label ℓ' within τ has been updated to $\ell' \sqcup \ell$.

T-ASSIGN
$$\frac{\gamma \vdash e : \tau \qquad \tau' = \text{raise}(\tau, pc) \qquad \gamma' = \gamma[lval \mapsto \tau']}{T, pc, \gamma \vdash lval := e : \{\gamma'\}}$$

T-ASSIGN rule follows the standard IFC convention. It updates the type of the left-hand-side of the assignment (i.e. $lval$) with the type of expression e while raising its security label to the current security context pc in order to capture indirect information flows.

T-SEQ
$$\frac{T, pc, \gamma \vdash s_1 : \Gamma_1 \qquad \forall \gamma_1 \in \Gamma_1.\ T, pc, \gamma_1 \vdash s_2 : \Gamma_2^{\gamma_1} \qquad \Gamma' = \bigcup_{\gamma_1 \in \Gamma_1} \Gamma_2^{\gamma_1}}{T, pc, \gamma \vdash s_1; s_2 : \Gamma'}$$

T-SEQ types the sequential composition of two statements. This rule type checks the first statement s_1, gathering all possible resulting state types into an intermediate set Γ_1. Then, for each state type in this intermediate set, the rule type checks the second statement s_2, and accumulates all resulting state types into the final state type set Γ'.

T-COND
$$\frac{\gamma \vdash e : \tau \qquad \ell = \text{lbl}(\tau) \qquad pc' = pc \sqcup \ell \qquad T, pc', (\text{refine}(\gamma, e)) \vdash s_1 : \Gamma_1 \qquad T, pc', (\text{refine}(\gamma, \neg e)) \vdash s_2 : \Gamma_2}{T, pc, \gamma \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \text{joinOnHigh}(\Gamma_1 \cup \Gamma_2, \ell)}$$

T-COND 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 5. Consider the conditional statement on line 56 of Program 1, where initially $\gamma = \{\texttt{enq\_qdepth} \mapsto \langle * \rangle_{19}^H, \texttt{hdr.ipv4.ecn} \mapsto \langle * \rangle_2^L, \ldots\}$. 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 $\gamma = \{x \mapsto \langle * \rangle_3^H; b \mapsto \langle * \rangle_1^L\}$. After typing both branches, the two typing state sets are $\Gamma_1 = \{\{x \mapsto \langle * \rangle_2^H \cdot \langle 0 \rangle_1^L\}\}$ and $\Gamma_2 = \{\{x \mapsto \langle * \rangle_2^H \cdot \langle 1 \rangle_1^L\}\}$. 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.

T-TRANS rule types parser transitions. 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.

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,

T-TRANS

$$\frac{\gamma \vdash e : \tau \qquad \ell = \text{lbl}(\tau) \qquad pc' = pc \sqcup \ell \qquad \gamma'_i = \text{refine}(\gamma, e = v_i \wedge \bigwedge_{j<i} e \neq v_j) \qquad T, pc', \gamma'_i \vdash T(st_i) : \Gamma_i \qquad \gamma'_d = \text{refine}(\gamma, \bigwedge_i e \neq v_i) \qquad T, pc', \gamma'_d \vdash T(st) : \Gamma_d \qquad \Gamma' = \Gamma_d \cup (\bigcup_i \Gamma_i) \qquad \Gamma'' = \text{joinOnHigh}(\Gamma', \ell)}{T, pc, \gamma \vdash \texttt{transition select}\, e\, \{v_1 : st_1, ..., v_n : st_n\}\, st : \Gamma''}$$

T-EMPTYTYPE

$$\frac{}{T, L, \bullet \vdash s : \Gamma}$$

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 6. Assume the state type $\gamma = \{\texttt{enq\_qdepth} \mapsto \langle 5 \rangle^H_{19}, \texttt{hdr.ipv4.ecn} \mapsto \langle * \rangle^L_2, \ldots\}$, upon reaching the conditional statement on line 56 of Program 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 $\bullet = \{\texttt{enq\_qdepth} \mapsto \langle\rangle^H_{19}, \ldots\}$, 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 [13], 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 [14], [15]. For instance, in Example 6, 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 the final state types.

T-CALL

$$\frac{\gamma \vdash \vec{e} : \vec{\tau} \qquad \text{tCall}(T, f, pc, \vec{\tau}, \gamma, \Gamma)}{T, pc, \gamma \vdash f(\vec{e}) : \Gamma}$$

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:

$$(s, \overline{(x, d)}) = T(f) \quad \gamma_f = \{x_i \mapsto \tau_i\} \quad T, pc, (\gamma_g, \gamma_f) \vdash s : \Gamma'$$
$$\Gamma = \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\}$$

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 to its 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 7. Assume that at line 44 of Program 1, the `ttl` in the state type is mapped to $\langle 1, 10 \rangle^L_8$. Calling `decrease` entails creating a new local state type and copying in the arguments, which yields $\gamma_{\texttt{decrease}} = \{x \mapsto \langle 1, 10 \rangle^L_8\}$. Typing the function's body (`x = x - 1`) results in the state type $\gamma'_{\texttt{decrease}} = \{x \mapsto \langle 0, 9 \rangle^L_8\}$. 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 [16].

T-TABLE

$$\frac{(\overline{e}, \text{Cont}_{\text{tbl}}) = T(tbl) \qquad \gamma \vdash e_i : \tau_i \qquad \ell = \bigsqcup_i \text{lbl}(\tau_i) \qquad pc' = pc \sqcup \ell \qquad \forall (\phi_j, (a_j, \overline{\tau}_j)) \in \text{Cont}_{\text{tbl}}. \quad \gamma_j = \text{refine}(\gamma, \phi_j) \qquad \text{tCall}(T, a_j, pc', \overline{\tau}_j, (\gamma_g, \gamma_l), \Gamma_j)}{T, pc, \gamma \vdash \texttt{apply}\ tbl : \text{joinOnHigh}(\cup_j \Gamma_j, \ell)}$$

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 V, 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 8. Given the table contract depicted in Fig. 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: $\text{refine}(\gamma, \texttt{dstAddr}[31 : 24] = 192)$. 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.

$$\text{T-EXTERN}$$
$$\frac{\begin{array}{c}(\gamma_g, \gamma_l) \vdash e_i : \tau_i \qquad (\text{Cont}_{\text{E}}, (x_1, d_1), ..., (x_n, d_n)) = T(f) \\ \gamma_f = \{x_i \mapsto \tau_i\} \qquad \forall (\gamma_i, \phi, \gamma_t) \in \text{Cont}_{\text{E}}.\ (\gamma_g, \gamma_f) \sqsubseteq \gamma_i \\ \Gamma' = \{\gamma' +\!\!+ \text{raise}(\gamma_t, pc) \mid (\gamma_i, \phi, \gamma_t) \in \text{Cont}_{\text{E}} \\ \qquad\qquad \wedge\ \text{refine}((\gamma_g, \gamma_f), \phi) = \gamma' \neq \bullet\} \\ \Gamma'' = \{(\gamma'_g, \gamma_l)[e_i \mapsto \gamma'_f(x_i) \mid \text{isOut}(d_i)] \mid (\gamma'_g, \gamma'_f) \in \Gamma'\}\end{array}}{T, pc, (\gamma_g, \gamma_l) \vdash f(e_1, ..., e_n) : \Gamma''}$$

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 $+\!\!+$ 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 9. In Program 1, let the contract for `mark_to_drop` at line 38 be defined as:

$$(\{\texttt{egress\_spec} \mapsto \langle * \rangle^{L}_{9}\}, \text{true}, \{\texttt{egress\_spec} \mapsto \langle 0 \rangle^{L}_{9}\})$$

which indicates that given an input state type $\{\texttt{egress\_spec} \mapsto \langle * \rangle^{L}_{9}\}$ the extern always sets the value of `egress_spec` to zero. Assuming an initial state type $\gamma = \{\texttt{egress\_spec} \mapsto \langle 7 \rangle^{L}_{9}\}$. 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 $\{\texttt{egress\_spec} \mapsto \langle 0 \rangle^{L}_{9}\}$ if pc is L, otherwise $\{\texttt{egress\_spec} \mapsto \langle 0 \rangle^{H}_{9}\}$.

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 $(m'_1 \setminus m_1) \underset{\gamma_t}{\sim} (m'_2 \setminus m_2)$.

C. Soundness

Given initial state types γ_1 and γ_2, and initial states m_1 and m_2, we write $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\sim}} m_2$ 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 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 $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\sim}} m_2$ 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 $m'_1 \overset{\gamma'_2}{\underset{\gamma'_1}{\sim}} m'_2$.*

Lemma 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 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 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 $m_1 \overset{\gamma_b}{\underset{\gamma_a}{\sim}} m_2$ it holds also that $m_1 \overset{\gamma_o}{\underset{\gamma_o}{\sim}} m_2$ 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 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 3 (Sufficient Condition). *Assume for every* $\gamma_1, \gamma_2 \in \Gamma$ *and every* $\gamma_o \in \Gamma_o$ *such that* $\gamma_1 \cap \gamma_o \neq \bullet$ *that*

(1) $\gamma_2 \cap \gamma_o \neq \bullet$ *implies* $\gamma_1 \sqcup \gamma_2 \sqsubseteq \gamma_o$, *and*
(2) for every lval either $\gamma_2(lval) \subseteq \gamma_o(lval)$ *or* $\gamma_1 \sqcup \gamma_2(lval) = \mathrm{L}$.

Then for every $\gamma_1, \gamma_2 \in \Gamma$ *such that* $m_1 \overset{\gamma_2}{\underset{\gamma_1}{\sim}} m_2$, *and every* $\gamma_o \in \Gamma_o$ *such that* $\gamma_o \vdash m_1$ *also* $\gamma_o \vdash m_2$ *and moreover* $m_1 \underset{\gamma_o}{\sim} m_2$.

In the statement of Lemma 3 we use $\gamma_2(lval) \subseteq \gamma_o(lval)$ to indicate that the interval of *lval* in γ_2 is included in the interval specified in γ_o.

Intuitively, Lemma 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.

D. Revisiting the basic congestion program

We revisit Program 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 II, 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 54 entails two possible scenarios. First scenario, when the destination address is in range `192.168.*.*`, as described in Example 5, the two state types resulting from the `if` at line 56 have `hdr.ipv4.ecn` set to H. As in Example 8, these state types satisfy only the first condition of the table's contract, which results in assigning the type $\langle 1, 9\rangle^L_9$ to `egress_spec` and producing the state types γ^1_{int} and γ^2_{int}.

Second scenario, when the destination address on line 54 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 $\langle 0\rangle^L_2$ 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^3_{int}, where the packet has been forwarded to the internal network and `egress_spec` is set to $\langle 1, 9\rangle^L_9$. 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. $\cap\gamma_o \neq \bullet$), 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. $\cap\gamma_o = \bullet$), 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 specific input policy case, Program 1 is non-interfering wrt. the output policy case γ_o.

Our analysis can also detect bugs. Assume a bug on line 54 of Program 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 54. 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. $\cap\gamma_o \neq \bullet$) 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 53 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 $\langle 0 \rangle_9^L$. This state type satisfies the sufficient condition, since `egress_spec` does not intersect $\gamma_o(\texttt{egress\_spec})$ and is L.

VII. Implementation and Evaluation

To evaluate our approach we developed Tap4s [10], a prototype tool which implements the security type system of Section VI. Tap4s is developed in Python and uses the lark parser library [17] 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 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.

Use cases. We evaluate Tap4s on 5 use cases, representing different real-world scenarios. The results of this evaluation are summarized in Table I. Due to space constraints, detailed descriptions of these use cases are provided in the full version of the paper [12]. We also implement and evaluate the use cases from P4BID [7]. These use cases are described in the full version of the paper [12], and their corresponding evaluation results are included in Table I. 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.

TABLE I: 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
Data-plane 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

VIII. Related Work

IFC for P4. Our work draws inspiration from P4BID [7], 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 [19]–[23] 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 [24] 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. [25] 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 [26] 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 [27], [28] are an expressive yet decidable refinement type system [29] to statically express and enforce data-dependent information flow polices. Lifty [30] provides tool support for specifying data-dependent policies and uses Haskell's liquid type-checker [28] to verify and repair the program against these policies. Storm [31] 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 VI-D), 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 [32] 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 [33] 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 [34]. 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 [35].

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 [36] 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 [37] 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 [38] 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 [39] and p4pktgen [40] 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. [41] 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. [42] 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.

IX. 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.

Acknowledgment

Thanks are due to 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 KTH Digital Futures research program, and the Swedish Research Council (VR).

References

[1] P. Goransson, C. Black, and T. Culver, *Software defined networks: a comprehensive approach*. Morgan Kaufmann, 2016.

[2] J. Matias, J. Garay, N. Toledo, J. Unzilla, and E. Jacob, "Toward an SDN-enabled NFV architecture," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 187–193, 2015.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[4] K. D. Albab, J. DiLorenzo, S. Heule, A. Kheradmand, S. Smolka, K. Weitz, M. Timarzi, J. Gao, and M. Yu, "SwitchV: automated SDN switch validation with P4 models," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 365–379.

[5] J. C. C. Chica, J. C. Imbachi, and J. F. B. Vega, "Security in SDN: A comprehensive survey," *Journal of Network and Computer Applications*, vol. 159, p. 102595, 2020.

[6] S. Zander, G. Armitage, and P. Branch, "Covert channels in the IP time to live field," in *Australian Telecommunication Networks and Application Conference (ATNAC) 2006*, 2006.

[7] K. Grewal, L. D'Antoni, and J. 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.

[8] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *Annual Asian Computing Science Conference*, 2006, pp. 75–89.

[9] M. Balliu, M. Dam, and G. Le Guernic, "Encover: Symbolic exploration for information flow security," in *2012 IEEE 25th Computer Security Foundations Symposium*, 2012, pp. 30–44.

[10] A. Alshnakat, A. M. Ahmadian, M. Balliu, R. Guanciale, and M. Dam, "TAP4S," 2025, software release. [Online]. Available: https://github.com/KTH-LangSec/TAP4S

[11] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *High. Order Symb. Comput.*, vol. 14, no. 1, pp. 59–91, 2001.

[12] A. Alshnakat, A. M. Ahmadian, M. Balliu, R. Guanciale, and M. Dam, "Securing p4 programs by information flow control," 2025. [Online]. Available: https://arxiv.org/abs/2505.09221

[13] T. H. Austin and C. 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.

[14] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August, "Rifle: An architectural framework for user-centric information-flow security," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 243–254.

[15] M. Balliu, D. Schoepe, and A. 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*, ser. Lecture Notes in Computer Science, vol. 10492, 2017, pp. 124–145.

[16] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, 2006, pp. 79–90.

[17] "Lark parser." [Online]. Available: https://github.com/lark-parser/lark

[18] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, no. 2–3, pp. 167–187, Jan. 1996.

[19] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 47–60.

[20] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," *Journal of Functional Programming*, vol. 27, p. e5, 2017.

[21] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 85–96, 2012.

[22] M. Guarnieri, M. Balliu, D. Schoepe, D. A. Basin, and A. Sabelfeld, "Information-flow control for database-backed applications," in *IEEE European Symposium on Security and Privacy, EuroS&P 2019*. IEEE, 2019, pp. 79–94.

[23] J. Parker, N. Vazou, and M. Hicks, "LWeb: Information flow security for multi-tier web applications," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[24] L. Lourenço and L. Caires, "Dependent information flow types," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 317–328.

[25] M. Eichholz, E. H. Campbell, M. Krebs, N. Foster, and M. Mezini, "Dependently-typed data plane programming," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–28, 2022.

[26] L. Lourenço and L. Caires, "Information flow analysis for valued-indexed data security compartments," in *International Symposium on Trustworthy Global Computing*, 2013, pp. 180–198.

[27] N. Vazou, P. M. Rondon, and R. Jhala, "Abstract refinement types," in *European Symposium on Programming*, 2013, pp. 209–228.

[28] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, 2014, pp. 269–282.

[29] R. Jhala, N. Vazou *et al.*, "Refinement types: A tutorial," *Foundations and Trends® in Programming Languages*, vol. 6, no. 3–4, pp. 159–317, 2021.

[30] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Liquid information flow control," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–30, 2020.

[31] N. Lehmann, R. Kunkel, J. Brown, J. Yang, N. Vazou, N. Polikarpova, D. Stefan, and R. Jhala, "STORM: Refinement types for secure web applications," in *15th USENIX Symposium on Operating Systems Design and Implementation OSDI 21)*, 2021, pp. 441–459.

[32] N. De Francesco and L. Martini, "Instruction-level security analysis for information flow in stack-based assembly languages," *Information and Computation*, vol. 205, no. 9, pp. 1334–1370, 2007.

[33] A. Cortesi and R. 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*, 2014, pp. 262–274.

[34] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *International Static Analysis Symposium*, 2004, pp. 100–115.

[35] T. A. S. B. A. Banerjee, "A logic for information flow in object-oriented programs."

[36] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 518–532.

[37] Y. Yao, Z. Cui, L. Tian, M. Li, F. Pan, and Y. Hu, "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.

[38] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in P4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.

[39] F. Ruffy, J. Liu, P. Kotikalapudi, V. Havel, H. Tavante, R. Sherwood, V. Dubina, V. Peschanenko, A. Sivaraman, and N. Foster, "P4Testgen: An extensible test oracle for P4-16," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 136–151.

[40] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for P4 programs," in *Proceedings of the Symposium on SDN Research*, 2018, pp. 1–7.

[41] K. Alpernas, R. Manevich, A. Panda, M. Sagiv, S. Shenker, S. Shoham, and Y. Velner, "Abstract interpretation of stateful networks," in *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*, 2018, pp. 86–106.

[42] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Abstract interpretation of distributed network control planes," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–27, 2019.