P4BID: Information Flow Control in P4*

Karuna Grewal
Cornell University
USA

Loris D'Antoni
University of Wisconsin
USA

Justin Hsu
Cornell University
USA

Abstract

Modern programmable network switches can implement custom applications using efficient packet processing hardware, and the programming language P4 provides high-level constructs to program such switches. The increase in speed and programmability has inspired research in *dataplane programming*, where many complex functionalities, e.g., key-value stores and load balancers, can be implemented entirely in network switches. However, dataplane programs may suffer from novel security errors that are not traditionally found in network switches.

To address this issue, we present a new information-flow control type system for P4. We formalize our type system in a recently-proposed core version of P4, and we prove a soundness theorem: well-typed programs satisfy non-interference. We also implement our type system in a tool, P4BID, which extends the type checker in the P4C compiler, the reference compiler for the latest version of P4. We present several case studies showing that natural security, integrity, and isolation properties in networks can be captured by non-interference, and our type system can detect violations of these properties while certifying correct programs.

CCS Concepts: • **Security and privacy → Information flow control**; *Logic and verification*; *Network security*.

Keywords: Information-flow control, programmable networks

ACM Reference Format:
Karuna Grewal, Loris D'Antoni, and Justin Hsu. 2022. P4BID: Information Flow Control in P4. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523717

*This is the conference version of the paper. We defer technical details, secondary definitions, and proofs to the full version of the paper [16].

1 Introduction

The last two decades have seen an ongoing shift in how networks are programmed. The task of programming a network once consisted of manually setting configurations in specialized switch hardware that provided limited customization; low-level programming was the only way to achieve performance. Today, switches are highly programmable and provide rich functionalities for processing network packets. This increased programmability is enabling complex network functionalities, which traditionally run on slower dedicated devices, to run directly on switches and other networking hardware [19, 31]. Furthermore, new programming models and languages make it easier for network operators to define complex functionalities [6].

While the advent of programmable network switches has inspired a large number of practitioners and researchers to write complex functionalities that can run on switches, it has also brought a new level of complexity in a world where bugs can be costly. As is well known, network configuration errors have led to widespread and costly outages (e.g., [13, 34]). The problem of preventing these, and other, types of bugs has received a lot of attention in the programming languages and verification communities. For example, researchers have developed formal tools for verifying that switch configurations guarantee desirable network properties, such as node reachability, the absence of black holes, and resilience to link failures (e.g., [1, 2, 33]). While these tools are extremely useful for network operators, applications running on programmable switches may exhibit errors that are not traditionally associated with networks. In particular, there has been little work on verifying security properties for dataplane programs.

Our work. We develop a new information-flow control (IFC) type system for the network programming language P4 [6], a leading language for programming network switches. P4 is an attractive target: it is actively developed by researchers from academia and industry, and can compile to a variety of networking hardware. *Information flow control* (IFC) is a well-studied, language-based approach to verifying security properties where variables in the program are tagged with security labels, and the type system ensures that no information can flow from high-security variables (secret) to low-security ones (public). IFC is (i) *flexible*: by changing the label usage one can model security properties, like confidentiality and integrity; (ii) *general*: it can accommodate complex programming constructs; and (iii) *lightweight*:

the analysis is simple, type-based, and requires minimal annotations from the programmer. Owing to these strengths, IFC has found wide adoption and has been deployed in real languages [24, 27].

Designing an IFC type system for P4 involves both technical and conceptual challenges. On the technical side, while P4 resembles a standard imperative language, it has a number of features to target the restricted computational model of networking switches. For instance, much of the computation in P4 programs happens via *tables*, which match on data in packet headers and select which actions to run. While a P4 program implements the actions, the table itself is not known until it is installed at runtime by the network controller. A second technical challenge is the size and complexity of the language. Like many languages in real-world use, P4 does not have a formal specification. To firm up the foundations of P4, Doenges et al. [10] developed a formal version of P4, called Core P4, as part of the broader PETR4 project. The formal operational model of Core P4 makes it possible to develop type systems that provably guarantee program properties. However, Core P4 is still quite large—P4 is a language intended for real-world use, with a wide variety of declarations, statements, and expressions, and Core P4 models almost all the features of P4. Our work develops an IFC system that can handle the principal features of Core P4.

On the conceptual side, IFC for dataplane programming has been little-studied and it is not know what useful properties network properties an IFC system can enforce. As part of our work, we present case studies showing that standard properties guaranteed by IFC, like *confidentiality* and *integrity*, are useful security properties for networking applications. We also show how natural network *isolation* properties can also be guaranteed with an IFC system, by adjusting the lattice of security labels.

Outline. After overviewing our approach in Section 2 and providing the necessary background on P4 and Core P4 in Section 3, we present our central contributions:

1. An *Information Flow Control (IFC) type system for Core P4* [10], a core calculus modeling the P4 language, together with a soundness theorem: well-typed programs satisfy *non-interference* (Section 4).
2. P4BID: a *type-checker* implemented on top of p4c, the reference compiler for P4. We evaluate our system through four case studies, demonstrating how properties enforced by IFC, like confidentiality and integrity, can be useful in a networking context. We implement our case studies in P4 and show that P4BID can automatically detect when these properties are violated, while correctly type-checking versions of these programs where the problems are removed (Section 5).

We conclude by surveying related work (Section 6) and outlining possible future directions (Section 7).

2 Overview

A quick introduction to P4. P4 is an actively-developed language for programming the network data plane. Computation is divided into three phases: *parser*, *pipeline*, and *deparser*. The packet processing starts at the parser, where the input packet is extracted into a typed representation given by headers using a finite state machine. The pipeline phase executes the primary logic of the switch by transforming the parsed representation of the input packet. The deparser serializes the parsed typed representation of the input packet into the output packet. Our work focuses on P4 *control blocks*, which implement the pipeline phase. To get a feel for the language, we consider a P4 program for a basic task: converting virtual addresses to physical addresses when packets enter a local network. Listing 1 begins by declaring the types of the *headers* which are carried by packets; P4 programs manipulate the state of packets by modifying the headers. In our case, there are three headers: ipv4 and ethernet carry the routing information in the original packet, while local_hdr carries information specific to the local network.

Listing 1 shows the code for the control block, which implements the core part of the logic. (The full P4 program also describes other stages of the packet-processing pipeline like parsing and deparsing, which we do not consider in our work.) The switch behavior is organized into *tables* and *actions*. Tables match data in headers (the *keys*) and apply actions. For instance, the table ipv4_lpm_forward inspects the value of the header hdr.ipv4.dstAddr and then decides whether to run action ipv4_forward or drop the packet. The concrete mapping is not specified by the P4 program; instead, the switch controller installs these mappings at runtime. Actions can inspect and modify packet headers. Actions can also be parameterized by arguments, which are supplied by the table when the action is applied. For example, the action ipv4_forward accepts a destination address and port as arguments, and then proceeds to update headers. Finally, the apply block specifies the overall behavior of the control block: here, the switch applies table virt2phys to translate virtual addresses to physical addresses, and then ipv4_lpm_forward to forward the packet.

A potential security vulnerability. Listing 1 is designed to process a packet as it enters a local network. The incoming packet refers to a *virtual address*, which must be translated to a physical address. Furthermore, the switch adjusts other packet fields, like the maximum number of hops (time-to-live, ttl), to reflect the topology of the local network. To preserve privacy, details of the local network should not leak into fields that are visible when the packet leaves the network. To accomplish this goal, the program uses a separate header of type local_hdr_t to store local information (Line 1). As the packet is routed in the local network, the switches do not touch the public ipv4 and ethernet headers; instead, they

Listing 1. Translating virtual to physical addresses.

```
1  header local_hdr_t {
2      bit<32> phys_dstAddr;
3      bit<8> phys_ttl;
4      bit<48> next_hop_MAC_addr;
5  }
6
7  header ipv4_t {
8      bit<8> ttl;
9      bit<8> protocol;
10     bit<32> srcAddr;
11     bit<32> dstAddr;
12 }
13
14 header eth_t {
15   bit<48> srcAddr;
16   bit<48> dstAddr;
17 }
18
19 struct headers {
20   ipv4_t ipv4;
21   eth_t eth;
22   local_hdr_t local_hdr;
23 }
24
25 control Obfuscate_Ingress(inout headers hdr,
26         inout standard_metadata_t std_metadata) {
27     table virtual2phys_topology {
28         key = { hdr.ipv4.dstAddr: exact; }
29         actions = { update_to_phys; }
30     }
31     action update_to_phys(bit<32> phys_dstAddr,
32                           bit<8> phys_ttl) {
33         hdr.local_hdr.phys_dstAddr = phys_dstAddr;
34         hdr.ipv4.ttl = phys_ttl;
35     }
36     table ipv4_lpm_forward {
37         key = { hdr.ipv4.dstAddr: lpm; }
38         actions = { ipv4_forward; drop; }
39     }
40     action ipv4_forward(bit<48> dstAddr, bit<9> port) {
41         hdr.eth.dstAddr = dstAddr;
42         standard_metadata.egress_spec = port;
43     }
44     action drop() { mark_to_drop(standard_metadata); }
45     apply {
46         virtual2phys_topology.apply();
47         ipv4_lpm_forward.apply();
48     }
49 }
```

parse local_hdr and update it with the next hop route information. When the packet exits the local network, the header local_hdr is removed.

While the intended behavior is simple to describe, the program in Listing 1 has an error: Line 34 incorrectly stores the local ttl in the ipv4 header, rather than the local_hdr header.

Listing 2. Security-Annotated Version of Listing 1

```
1  header local_hdr_t {
2      <bit<32>, high> phys_dstAddr;
3      <bit<8>, high> phys_ttl;
4      // ...
5  }
6
7  header ipv4_t {
8      <bit<8>, low> ttl;
9      // ...
10 }
11
12 struct headers {
13   ipv4_t ipv4;
14   local_hdr_t local_hdr;
15   // ...
16 }
17
18 control Obfuscate_Ingress(inout headers hdr,
19         inout standard_metadata_t std_metadata) {
20     action update_to_phys(<bit<32>, high> phys_dstAddr,
21                           <bit<8>, high> phys_ttl) {
22         hdr.local_hdr.phys_dstAddr = phys_dstAddr;
23         // !BUG!: low <- high
24         hdr.ipv4.ttl = phys_ttl;
25         // *FIX*: high <- high
26         hdr.local_hdr.phys_ttl = phys_ttl;
27     }
28     // ...
29 }
```

Even when the local header is removed, the ipv4 header will carry private information about the local network. This kind of error unintentionally leaks local information into public headers, but it can be easy to overlook.

Security types to the rescue. We design an information-flow control type system for P4 to catch such bugs. Like standard IFC type systems, our system extends each P4 type with a *security label*: high if the data is secret, and low if the data is public. Listing 2 shows our example program annotated with security types. All data specific to the local network (e.g., phys_dstAddr, phys_ttl) are marked as *high* security. The publicly visible headers (e.g., ipv4, eth) are marked as *low* security. Our type system guarantees that information from high-security data does not influence low-security data. For instance, the information leak we saw before can be flagged in our type system: Line 24 incorrectly assigns a high-security data phys_ttl to a low-security field ipv4.ttl. The problem is corrected by assigning phys_ttl to local_hdr.ttl (Line 26), which is a high-security field.

While this kind of analysis is fairly straightforward, the design of our type system must handle unusual features from P4's programming model (e.g., actions and tables); we discuss these aspects in Section 3 and Section 4. Furthermore, while Listing 1 demonstrates a basic information leak, we will see more interesting applications of our type system to networking applications in Section 5.

3 Syntax and Semantics of Core P4

This section briefly reviews the core P4 calculus presented in the recent work on PETR4 [10], the representation of P4 programs in terms of the core calculus syntax, and the operational semantics and typing judgements for the core calculus.

3.1 Core P4 Syntax

PETR4 formalizes the semantics of various P4 primitives, like control blocks, match-action tables, and statements in a calculus called Core P4. For our information-flow control type system, we focus on the fragment of Core P4 in Figure 1. Expressions and statements are largely standard.

Core P4 programs (*prg*) are represented as a sequence of variable, object, or type declarations followed by a control block. The central construct in a P4 program is the control block, which describes how the switch processes packets in terms of table and action calls inside its apply block. A control block body (*ctrl_body*) is a sequence of declarations and statements. The *stmt* in the control block corresponds to the apply block of a P4 program.

Variable and type declarations (*var_decl*, *typ_decl*) are largely standard; the match_kind enum declares different ways tables can match on packet fields. Object declarations (*obj_decl*) declare P4 objects: tables and actions. These object declarations can have nested ordinary statements (*stmt*) that allow usual imperative primitives like mutation and control flow statements. To get a feel for these features, let's consider how they correspond to parts of the Obfuscate_Ingress control block in Listing 1. The example control block consists of three actions declarations (update_to_phys, ipv4_forward, and drop), and two table declarations (virtual2phys_topology and ipv4_lpm_forward).

Tables. A table declaration, table $x\ \{\overline{key}\ \overline{act}\}$, is composed of a list of expressions (usually packet header fields) that specify the lookup key, $\overline{key}$, and actions, $\overline{act}$, which the lookup table might execute. A table application uses the key to lookup the entries in the table (installed by the control plane) and invokes the action from the matched entry. For example, table virtual2phys_topology in Line 27 contains the key hdr.ipv4.dstAddr: exact (where exact specifies the match pattern, in this case, exact match on the key), and the action update_to_phys action. Applying this table, represented in Core P4 as virtual2phys_topology(), matches the table entries installed by the control plane against the corresponding keys in the current packet and returns an appropriate action to run, with all its arguments. Any optional arguments in the returned action will be supplied by the control plane. The match pattern determines the criterion for choosing a table entry based on the key. For instance, lpm specifies that a key is matched to the entry corresponding to its longest prefix; exact specifies that a key should be exactly matched to some table entry otherwise it is a match failure.

Actions. An action declaration is a special case of a function declaration, function $\tau_{ret}\ x\ (\overline{d\ y : \tau})\{stmt\}$, with no return type. For example, the action update_to_phys on Line 32 in Listing 1 has parameters phys_dstAddr and phys_ttl, of types bit⟨32⟩ and bit⟨8⟩. Parameters can have a *directionality*, d: an *in* expression can only be read from, while an *inout* expression can be both read and written to. Omitted directions in parameters default to the *in* direction; these directionless parameters are optional arguments that can be passed by the control plane. Invoking the action, which can be done directly as a statement or indirectly from a table, runs the statement $stmt$ in the action body. Actions, like all Core P4 functions, do not support recursion.

Differences compared to Core P4. The language in Figure 1 is a significant fragment of Core P4, but it does not handle some of its more specialized features (e.g., generics, constant declarations, slice operation, and native functions). We consider this fragment for simplicity, but we do not foresee difficulties in extending our IFC analysis to full Core P4. We omitted some lesser-used features, like generics, because the core language is already quite large and we believe it is unlikely that omitted features lead to information-flow violations. We focus on programs with a single control block because most P4 programs encode their main functionality in a single ingress control block. Since our system already supports user-defined functions and closures, with all of their technical intricacies, we do not see any obstacle to handling multiple control blocks besides increasing the complexity of our type system.

3.2 Core P4 Semantics

To understand the semantics of Core P4 programs, we will review the evaluation judgement forms for expressions, statements, and declarations from PETR4 [10]. The main judgements are as follows:

$$\langle C, \Delta, \mu, \epsilon, exp\rangle \Downarrow \langle \mu', val\rangle$$
$$\langle C, \Delta, \mu, \epsilon, stmt\rangle \Downarrow \langle \mu', \epsilon', sig\rangle$$
$$\langle C, \Delta, \mu, \epsilon, decl\rangle \Downarrow \langle \Delta', \mu', \epsilon', sig\rangle$$

The contexts used in these judgements are defined in Figure 2. Here, Δ is the partial map from type names to types; ϵ is the partial map between variables and their memory locations; μ is the memory store mapping variable locations to their values. C models the table lookup map provided by the control plane: given a table at location l with $key = val$, and a list of actions described by a list of $PartialActionRef$ (actions with optional arguments missing), C returns an action call expression with all the optional arguments of the action supplied ($ActionRef$). The judgements use val to denote a

$$
\begin{array}{rcll}
exp & ::= & b & \text{Boolean} \\
 & \mid & n_w & \text{integers or bits of width w} \\
 & \mid & x & \text{variable} \\
 & \mid & exp_1[exp_2] & \text{array indexing} \\
 & \mid & exp_1 \oplus exp_2 & \text{binary operation} \\
 & \mid & \{\overline{f_i = exp_i}\} & \text{record} \\
 & \mid & exp.f_i & \text{field projection} \\
 & \mid & exp_1(\overline{exp_2}) & \text{function call}
\end{array}
$$

(a) Expressions

$$
\begin{array}{rcll}
stmt & ::= & exp_1(\overline{exp_2}) & \text{function call} \\
 & \mid & exp_1 := exp_2 & \text{assignment} \\
 & \mid & \text{if } (exp_1)\ stmt_1 \text{ else } stmt_2 & \text{conditional} \\
 & \mid & \{\overline{stmt}\} & \text{sequencing} \\
 & \mid & \text{exit} & \text{exit} \\
 & \mid & \text{return } exp & \text{return} \\
 & \mid & var\_decl & \text{variable declaration}
\end{array}
$$

(b) Statements

$$
\begin{array}{rcl}
prg & ::= & \overline{typ\_decl}\ ctrl\_body \\
ctrl\_body & ::= & \overline{decl}\ stmt \\
decl & ::= & var\_decl \mid obj\_decl \mid typ\_decl \\
var\_decl & ::= & \tau\ x := exp \mid \tau\ x \\
typ\_decl & ::= & \text{match\_kind } \{\overline{f}\} \mid \text{typedef } \tau\ X \\
obj\_decl & ::= & \text{table } x\ \{\overline{key}\ \overline{act}\} \\
 & \mid & \text{function } \tau_{ret}\ x\ (\overline{d\ y : \tau})\{stmt\}
\end{array}
$$

(c) Declarations

$$
\begin{array}{rcl}
d & ::= & in \mid inout \\
lval & ::= & x \\
 & \mid & lval.f \\
 & \mid & lval[n] \\
key & ::= & exp : x \\
act & ::= & x(\overline{exp}, \overline{x : \tau})
\end{array}
$$

(d) Other constructs

Figure 1. Core P4 Expressions (fragment)

$$
\begin{array}{rlrl}
Var & : \text{variables} & Val & : \text{values} \\
TypVar & : \text{type variables} & Typ & : \text{types in Core P4} \\
Loc & : \text{locations} & &
\end{array}
$$

$$
\begin{array}{lll}
\Gamma & : Var \longrightarrow Typ & \Delta \quad : TypVar \longrightarrow Typ \\
\epsilon & : Var \longrightarrow Loc & \mu \quad : Loc \longrightarrow Val \\
C & : Loc \times Val \times \overline{PartialActionRef} \longrightarrow ActionRef &
\end{array}
$$

Figure 2. Typing and Evaluation Contexts

$$
\begin{array}{rcl}
\rho & ::= & bool \mid int \mid bit\langle n\rangle \mid unit \\
 & \mid & \{\overline{f : \rho}\} \mid header\{\overline{f : \rho}\} \mid \rho[n] \\
 & \mid & match\_kind\{\overline{f}\} \\
\kappa & ::= & \rho \mid table \mid \overline{d\ \kappa} \longrightarrow \kappa
\end{array}
$$

Figure 3. Core P4 types

value; and *sig* to denote a *signal*, which indicates whether the program's control flow proceeds normally (cont), returns a value (return *val*), or errors (exit).

Since function calls are expressions, and a function's body can update the memory store, the evaluation judgement for expressions can modify the memory store. Similarly, the statement evaluation judgement captures the updated memory store from evaluating a statement with side-effects and the environment extension on declaring a new variable. A declaration evaluation can reduce to a new memory store and environment when evaluating a variable or object declaration. Additionally, a declaration statement can update the type definition context by introducing a new type alias. Both declarations and statements evaluate to a signal *sig*, representing the result of the control flow in their sequencing blocks.

3.3 Core P4 Type System

Figure 3 recalls the types from Core P4. Core P4 divides the P4 types into two categories: base types, ρ, and general types, κ. The fields of headers and records must be base types. The simplified Core P4 typing judgements for the fragment of Core P4 presented in Figure 1 are as follows:

$$\Gamma, \Delta \vdash exp : \kappa\ goes\ d \qquad \Gamma, \Delta \vdash stmt \dashv \Gamma' \qquad \Gamma, \Delta \vdash decl \dashv \Gamma', \Delta'$$

The expression typing judgement associates a directionality with expressions to indicate if the expression is read only (in) or is both readable and writable (inout). Intuitively, the contexts on the left of $\vdash$ in the statement and declaration typing rule describe the contexts before their execution, while the contexts on the right of $\dashv$ define the context after the execution of the statement and declaration.[1]

4 IFC Type System for P4

This section presents the security-type extension for the Core P4 fragment presented in Figure 1. Before presenting the security-types for our fragment of Core P4, we describe the main idea behind security type systems.

4.1 Background on Security Type Systems

A security type system lifts ordinary types to security types by annotating them with security labels [28]. These security

[1]The original Core P4 typing judgements also have a constant store, to model compile-time constants. We omit this store since our fragment does not include compile-time constants.

labels are drawn from a security lattice, $(\mathbb{L}, \sqsubseteq)$, associated with the type system. We illustrate the key ideas using a simple two point lattice $\{\text{low}, \text{high}\}$. Here, low identifies publicly visible values and high represents secure values, and low $\sqsubseteq$ high.

Consider a well-typed closed expression *exp* with type τ, represented by an ordinary type system as $\vdash exp : \tau$. A security-type system will additionally assign a security label, $\chi \in \mathbb{L}$ to *exp*. This can be represented by the typing judgement $\vdash exp : \langle \tau, \chi \rangle$, where the pair $\langle \tau, \chi \rangle$ is the *security type*. For instance, if *exp* evaluates to *val* and $\chi = \text{high}$, then *val* is considered to be a secure value.

For statements (or expressions) that can mutate variables, a security type system assigns a security label $pc \in \mathbb{L}$ to the typing judgements. This label denotes the security context used to track the security level for variables that can be written at a given program point (*program counter*). Consider a conditional statement that branches on a high security guard expression:

$$\text{if } (h == 1)\ \{\ h := set\_high();\ \}\ \text{else}\ \{\ h := 1;\ \},$$

where the security level of h is high and the *set_high* function call in the true branch writes to only high security variables. Since the guard is at high security level, the pc for both the conditional branches becomes high. Here, both branches need to be well-typed under the high security label, which implies that no variable at security level lower than high can be mutated in either branch. For instance, we must have $\Gamma \vdash_{\text{high}} h := set\_high()$ and $\Gamma \vdash_{\text{high}} (h := 1)$. Without this restriction, there can be an implicit flow of information from the conditional guard into the statement blocks of the conditional, for instance, if the function wrote to a low variable.

The utility of a security-type system lies in the *non-interference* guarantee offered by a well-typed program. To define non-interference, suppose that all low security variables are observable while any high security variable is unobservable. Informally, non-interference can be understood as the property of a program where no unobservable input variable influences the value of any observable output.

4.2 P4 IFC Type System

This section describes our information-flow control type system for the language in Figure 1. We assume the lattice $(\mathbb{L}, \sqsubseteq)$ of security labels has $\top$ and $\bot$ elements, representing the top and bottom elements of the lattice. In our example lattice, $\bot = \text{low}$ and $\top = \text{high}$.

Figure 4 summarizes the security types of our information-flow control system. Core P4 types are lifted to security types using a security label, χ, from the lattice $\mathbb{L}$. We also use pc to denote a security label when it is used as a security context. As in Core P4, we distinguish between base security types ρ and general security types κ. For non-base types, the security label is tracked within the type itself, for instance, the fields of headers and records are assigned security labels

$$\begin{array}{lcl}
\rho & ::= & \langle bool, \chi\rangle \mid \langle int, \chi\rangle \mid \langle bit\langle n\rangle, \chi\rangle \mid \langle unit, \bot\rangle \\
 & \mid & \langle \{\overline{f : \rho}\}, \bot\rangle \mid \langle header\{\overline{f : \rho}\}, \bot\rangle \mid \langle \rho[n], \bot\rangle \\
 & \mid & \langle match\_kind\{\overline{f}\}, \bot\rangle \\
\kappa & ::= & \rho \mid \langle table(pc_{tbl}), \bot\rangle \mid \langle \overline{d\ \rho} \xrightarrow{pc} \rho_{ret}, \bot\rangle \\
\tau & ::= & bool \mid int \mid bit\langle n\rangle \mid unit \\
 & \mid & \{\overline{f : \rho}\} \mid header\{\overline{f : \rho}\} \\
 & \mid & \rho[n] \mid match\_kind\{\overline{f}\} \\
 & \mid & table(pc_{tbl}) \mid \overline{d\ \rho} \xrightarrow{pc} \rho_{ret}
\end{array}$$

Figure 4. IFC Types

instead of the header or record. But to keep the shape of types uniform, we assign the $\bot$ security label for such types. We use the metavariable τ to denote a security type without its outer-most security label; thus, security types are of the form $\langle \tau, \chi \rangle$.

Before describing the judgement forms of the security type system, we introduce the contexts used in the typing judgements. The typing judgements use a typing context, Γ, a type definition context, Δ, and a security context, pc, which are same as Core P4's contexts Figure 2, with the difference that now *Typ* is the set of security types of the form $\langle \tau, \chi \rangle$.

For a given security label pc, variables in a typing context Γ at security level $\chi \sqsubseteq pc$ will be referred as *below-pc* variables, and variables at security level $\chi \not\sqsubseteq pc$ will be referred as *not below-pc* (or sometimes *above-pc*) variables.

Our security type system has three forms of judgements for expressions, statements, and declarations, respectively:

$$\begin{aligned}
\textbf{Expressions} &: \Gamma, \Delta \vdash_{pc} exp : \langle \tau, \chi \rangle \ goes\ d \\
\textbf{Statements} &: \Gamma, \Delta \vdash_{pc} stmt \ \dashv \Gamma' \\
\textbf{Declarations} &: \Gamma, \Delta \vdash_{pc} decl \ \dashv \Gamma', \Delta'
\end{aligned}$$

The direction annotation *goes d* in the typing judgement for expressions is dropped when the direction is not important. The complete security typing rules can be found in Figure 5 (expressions), Figure 6 (statements), and Figure 7 (declarations). Expression typing assumes a typing oracle $\mathcal{T}$, giving the meaning of the binary operations. In statement and declaration typing, the judgement $\Delta \vdash \tau \rightsquigarrow \tau'$ converts τ to a base type by unfolding type definitions [10]. Below, we discuss the most interesting—and technically intricate—typing rules: those for functions, tables, and subtyping.

Typing rules for functions. Our system has rules for function declarations and function calls. These are also the key rules for typing actions, which are functions with no return type. The T-FnDecl rule in Figure 7 typechecks the body of the function to eliminate any leaks in the function body. The pc_{fn} security label on the function's arrow type records the lower bound on the security labels of the variables that the function mutates. For instance, in the following function:

$$\text{function insecure}()\{l := 1; h := 2;\},$$

where the security labels of l and h variable are low and high respectively, pc_{fn} will be low. The T-FnCall rule in Figure 5 enforces that a function will not be invoked in a context that is higher than the function's pc_{fn} because doing so, for instance in the example program, will implicitly flow information from a high guard expression into a low variable.

Typing rules for tables. Since a table matches on the key to select an action to invoke, the key of a table resembles the guard of a conditional. Thus, the value of a key can implicitly leak in the action's body if the invoked action writes to variables at security label lower that that of the key expression. Therefore, to declare a table of type $\langle table(pc_{tbl}), \bot\rangle$, the rule T-TblDecl in Figure 7 ensures that the security label of the most secure key, χ_k, is lower than the label of the least secure assignment, pc_a, in any action. Here, pc_{tbl} records the lower bound on the write effects associated with any keys, actions, or arguments.

The T-TblCall rule in Figure 6 prevents any implicit flow into any of the actions that a table might invoke by allowing a table to be applied only in a pc context lower than the least secure write effect associated with the table application, pc_{tbl}. This prevents implicit leaks during the evaluation of keys, arguments, or the action's body.

Subtyping rule. The T-SubType-In rule in Figure 5 allows only read-only (*in*) expressions to increase their security label. It is not safe to allow *inout* expressions to be subtyped. To see why, consider the following function:

$$\text{write\_to\_high } (inout \text{ h} : \langle bool, high\rangle) \ \{\text{h} := true;\}$$

Suppose we have a low variable $\text{l} : \langle bool, low\rangle$. Since variables are *inout* expressions (T-Var in Figure 5), if *inout* expressions were allowed to increase their label, write_to_high(l) call would have been valid. In this case, the function would have written to a low variable when it should have operated with only a high variable.

4.3 Non-Interference

To define non-interference, consider two program states, $\langle C, \Delta, \mu_a, \epsilon_a\rangle$ and $\langle C, \Delta, \mu_b, \epsilon_b\rangle$, where the environments have equal domains. Suppose every below-pc variable x has equal value under both the memory stores, $\mu_a(\epsilon_a(x)) = \mu_b(\epsilon_b(x))$, but the value of any variables that are not below-pc can differ between the two stores. Non-interference is satisfied if evaluating an expression, statement, or declaration in the two program states results in two final program states that agree on below-pc variables.

The following definition formally describes a pair of below-pc equivalent memory stores and environments. The store typing context Ξ maps locations in a store to security types.

Definition 4.1. Consider two pairs of memory stores and environments $\langle \mu_a, \epsilon_a\rangle$ and $\langle \mu_b, \epsilon_b\rangle$. Then

$$\Xi_a, \Xi_b, \Delta \models_{pc} \langle \mu_a, \epsilon_a\rangle \ \langle \mu_b, \epsilon_b\rangle : \Gamma$$

is satisfied when

$$\Xi_a, \Delta \models \langle \mu_a, \epsilon_a\rangle : \Gamma \qquad \text{and} \qquad \Xi_b, \Delta \models \langle \mu_b, \epsilon_b\rangle : \Gamma$$

and every below-pc variable x in ϵ_a and ϵ_b has equal value *i.e.,* $\mu_a(\epsilon_a(x)) = \mu_b(\epsilon_b(x))$.

Intuitively, $\Xi, \Delta \models \langle \mu, \epsilon\rangle : \Gamma$ states that the store and environment are well-typed: recalling that the location of every variable is described by the environment ϵ and the value at valid locations is described by the memory store μ, the type assigned to a variable using the store typing Ξ must be the same as the type assigned by the typing context Γ.

The following definition of non-interference for statements requires that evaluating a statement under below-pc equivalent pairs of memory stores and environment can only reduce to pairs of final memory stores and environments that are below-pc equivalent. Technically, this is a *termination insensitive* notion of non-interference, since it does not require that both executions terminate. However, P4 programs do not allow recursion and Doenges et al. [10] prove that all well-typed Core P4 programs terminate.

Definition 4.2 (Non-interference for statements)**.** For any security lable $l, \Gamma, \Delta \models_{pc} \text{NI}(stmt) \dashv \Gamma'$ holds for any Ξ_a, Ξ_b, μ_a, μ_b, ϵ_a, ϵ_b, μ'_a, μ'_b, ϵ'_a, ϵ'_b if whenever

1. $\Xi_a, \Xi_b, \Delta \models_l \langle \mu_a, \epsilon_a\rangle \ \langle \mu_b, \epsilon_b\rangle : \Gamma$,
2. $\langle C; \Delta; \mu_a; \epsilon_a; stmt\rangle \Downarrow \langle \mu'_a; \epsilon'_a; sig_1\rangle$,
3. $\langle C; \Delta; \mu_b; \epsilon_b; stmt\rangle \Downarrow \langle \mu'_b; \epsilon'_b; sig_2\rangle$

then there exists Ξ'_a, Ξ'_b, such that

1. $\Gamma, \Delta \vdash_{pc} stmt \ \dashv \Gamma'$,
2. $\Xi'_a, \Xi'_b, \Delta \models_l \langle \mu'_a, \epsilon'_a\rangle \ \langle \mu'_b, \epsilon'_b\rangle : \Gamma'$,
3. $\Xi'_a, \Xi'_b, \Delta \models_l \langle \mu'_a, \epsilon_a\rangle \ \langle \mu'_b, \epsilon_b\rangle : \Gamma$,
4. for any $l_a \in \textsc{dom}(\mu_a)$ and $l_b \in \textsc{dom}(\mu_b)$ such that $\Xi_a, \Delta \vdash \mu_a(l_a) : \langle \tau, \chi\rangle$ and $\Xi_b, \Delta \vdash \mu_b(l_b) : \langle \tau, \chi\rangle$ and $pc \not\sqsubseteq \chi$, we have $\mu'_a(l_a) = \mu_a(l_a)$ and $\mu'_b(l_b) = \mu_b(l_b)$,
5. for any $l_a \in \textsc{dom}(\mu_a)$ such that $\Xi_a, \Delta \vdash \mu_a(l_a) : \tau_{clos}$, where $\tau_{clos} \in \{\tau_{fn}, \tau_{tbl}\}$, we have $\mu'_a(l_a) = \mu_a(l_a)$,
6. for any $l_b \in \textsc{dom}(\mu_b)$ such that $\Xi_b, \Delta \vdash \mu_b(l_b) : \tau_{clos}$, where $\tau_{clos} \in \{\tau_{fn}, \tau_{tbl}\}$, we have $\mu'_b(l_b) = \mu_b(l_b)$,
7. one of the following holds:
   - $sig_1 = sig_2 = cont$; or
   - $sig_1 = sig_2 = exit$; or
   - $sig_1 = $ return val_1 and $sig_2 = $ return val_2 such that $\Xi'_a, \Xi'_b, \Delta \models_l NI(val_1, val_2) : \langle \tau'_{ret}, \chi_{ret}\rangle$, where $\Delta \vdash \tau_{ret} \rightsquigarrow \tau'_{ret}$ and $\Gamma[\text{return}] = \langle \tau_{ret}, \chi_{ret}\rangle$,
8. we have the inclusions:
   - $\Xi_a \subseteq \Xi'_a$ and $\Xi_b \subseteq \Xi'_b$;
   - $\textsc{dom}(\mu_a) \subseteq \textsc{dom}(\mu'_a)$ and $\textsc{dom}(\mu_b) \subseteq \textsc{dom}(\mu'_b)$; and
   - $\textsc{dom}(\epsilon_a) \subseteq \textsc{dom}(\epsilon'_a)$ and $\textsc{dom}(\epsilon_b) \subseteq \textsc{dom}(\epsilon'_b)$.

$$\frac{\Gamma, \Delta \vdash_{pc'} exp : \langle \tau, \chi \rangle \qquad pc \sqsubseteq pc'}{\Gamma, \Delta \vdash_{pc} exp : \langle \tau, \chi \rangle}\ \textsc{T-Subtype-PC} \qquad \frac{\Gamma, \Delta \vdash_{pc} exp : \langle \tau, \chi \rangle\ goes\ in \qquad \chi \sqsubseteq \chi'}{\Gamma, \Delta \vdash_{pc} exp : \langle \tau, \chi' \rangle\ goes\ in}\ \textsc{T-SubType-In}$$

$$\frac{}{\Gamma, \Delta \vdash_{pc} b : \langle bool, \bot \rangle\ goes\ in}\ \textsc{T-Bool} \qquad \frac{}{\Gamma, \Delta \vdash_{pc} n_\infty : \langle int, \bot \rangle\ goes\ in}\ \textsc{T-Int} \qquad \frac{x \in \textsc{dom}(\Gamma) \qquad \Gamma(x) = \langle \tau, \chi \rangle}{\Gamma, \Delta \vdash_{pc} x : \langle \tau, \chi \rangle \text{ goes inout}}\ \textsc{T-Var}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash_{pc} exp_1 : \langle \rho_1, \chi_1 \rangle \qquad \Gamma, \Delta \vdash_{pc} exp_2 : \langle \rho_2, \chi_2 \rangle \\ \mathcal{T}(\Delta; \oplus; \rho_1; \rho_2) = \rho_3 \qquad \chi_1 \sqsubseteq \chi' \qquad \chi_2 \sqsubseteq \chi'\end{array}}{\Gamma, \Delta \vdash_{pc} exp_1 \oplus exp_2 : \langle \rho_3, \chi' \rangle\ goes\ in}\ \textsc{T-BinOP} \qquad \frac{\Gamma, \Delta \vdash_{pc} \{\overline{exp : \langle \tau_i, \chi_i \rangle}\}}{\Gamma, \Delta \vdash_{pc} \{\overline{f : exp}\} : \langle \{\overline{f : \langle \tau_i, \chi_i \rangle}\}, \bot \rangle\ goes\ in}\ \textsc{T-Rec}$$

$$\frac{\Gamma, \Delta \vdash_{pc} exp : \langle \{\overline{f_i : \langle \tau_i, \chi_i \rangle}\}, \bot \rangle\ goes\ d}{\Gamma, \Delta \vdash_{pc} exp.f_i : \langle \tau_i, \chi_i \rangle\ goes\ d}\ \textsc{T-MemRec} \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{pc} exp_1 : \langle \langle \tau, \chi_1 \rangle [n], \bot \rangle\ goes\ d \\ \Gamma, \Delta \vdash_{pc} exp_2 : \langle bit\langle 32 \rangle, \chi_2 \rangle \qquad \chi_2 \sqsubseteq \chi_1\end{array}}{\Gamma, \Delta \vdash_{pc} exp_1[exp_2] : \langle \tau, \chi_1 \rangle\ goes\ d}\ \textsc{T-Index}$$

$$\frac{\Gamma, \Delta \vdash_{pc} exp : \langle header\{\overline{f_i : \langle \tau_i, \chi_i \rangle}\}, \bot \rangle\ goes\ d}{\Gamma, \Delta \vdash_{pc} exp.f_i : \langle \tau_i, \chi_i \rangle\ goes\ d}\ \textsc{T-MemHdr} \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{pc} exp_1 : \langle \overline{d\ \langle \tau_i, \chi_i \rangle} \xrightarrow{pc_{fn}} \langle \tau_{ret}, \chi_{ret} \rangle, \bot \rangle \\ \Gamma, \Delta \vdash_{pc} \overline{exp_2 : \langle \tau_i, \chi_i \rangle\ goes\ d} \qquad pc \sqsubseteq pc_{fn}\end{array}}{\Gamma, \Delta \vdash_{pc} exp_1(\overline{exp_2}) : \langle \tau_{ret}, \chi_{ret} \rangle \text{ goes in}}\ \textsc{T-Call}$$

Figure 5. IFC Typing Rules for Expressions

$$\frac{}{\Gamma, \Delta \vdash_{pc} \{\}\ \dashv \Gamma}\ \textsc{T-Empty} \qquad \frac{}{\Gamma, \Delta \vdash_{\bot} \text{exit}\ \dashv \Gamma}\ \textsc{T-Exit} \qquad \frac{\Gamma, \Delta \vdash_{pc} stmt_1 \dashv \Gamma_1 \qquad \Gamma_1, \Delta \vdash_{pc} \{\overline{stmt_2}\} \dashv \Gamma_2}{\Gamma, \Delta \vdash_{pc} \{stmt_1; \overline{stmt_2}\} \dashv \Gamma_2}\ \textsc{T-Seq}$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash_{pc} exp_1 : \langle \tau, \chi_1 \rangle\ goes\ inout \\ \Gamma, \Delta \vdash_{pc} exp_2 : \langle \tau, \chi_2 \rangle \qquad \chi_2 \sqsubseteq \chi_1 \qquad pc \sqsubseteq \chi_1\end{array}}{\Gamma, \Delta \vdash_{pc} exp_1 := exp_2 \dashv \Gamma}\ \textsc{T-Assign} \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash_{\chi_2} stmt_1\ \dashv \Gamma_1 \qquad \Gamma, \Delta \vdash_{\chi_2} stmt_2\ \dashv \Gamma_2 \\ \Gamma, \Delta \vdash_{pc} exp : \langle bool, \chi_1 \rangle \qquad \chi_1 \sqsubseteq \chi_2 \qquad pc \sqsubseteq \chi_2\end{array}}{\Gamma, \Delta \vdash_{pc} \text{if } (exp)\ stmt_1 \text{ else } stmt_2\ \dashv \Gamma}\ \textsc{T-Cond}$$

$$\frac{\Gamma, \Delta \vdash_{pc} exp : \langle \tau, \chi_{ret} \rangle \qquad \Gamma(\text{return}) = \langle \tau_{ret}, \chi_{ret} \rangle \qquad \Delta \vdash \tau_{ret} \rightsquigarrow \tau}{\Gamma, \Delta \vdash_{\bot} \text{return } exp \dashv \Gamma}\ \textsc{T-Return} \qquad \frac{\Gamma, \Delta \vdash_{pc} var\_decl \dashv \Gamma_1, \Delta}{\Gamma, \Delta \vdash_{pc} var\_decl \dashv \Gamma_1}\ \textsc{T-Decl}$$

$$\frac{\Gamma, \Delta \vdash_{pc} exp_1(\overline{exp_2}) : \langle \tau_{ret}, \chi_{ret} \rangle}{\Gamma, \Delta \vdash_{pc} exp_1(\overline{exp_2}) \dashv \Gamma}\ \textsc{T-FnCallStmt} \qquad \frac{\Gamma, \Delta \vdash_{pc} exp : \langle table(pc_{tbl}), \bot \rangle \qquad pc \sqsubseteq pc_{tbl}}{\Gamma, \Delta \vdash_{pc} exp() \dashv \Gamma}\ \textsc{T-TblCall}$$

Figure 6. IFC Typing Rules for Statements

We present similar non-interference definitions for expressions and declarations in the full version of the paper.

Then, our main soundness theorem states that a well-typed program in our information-flow control type system will be non-interfering.

Theorem 4.3 (Main Soundness Theorem)**.** *If* $\Gamma, \Delta \vdash_{pc} stmt \dashv \Gamma'$, *then* $\Gamma, \Delta \models_{pc} NI(stmt) \Dashv \Gamma'$.

We present similar non-interference theorems for expressions and declarations in the full version of the paper.

Proof Sketch. We prove non-interference theorems for statements, expressions and declarations together as a mutual induction on the typing derivation. The detailed proof of Theorem 4.3 is given in the full version of the paper. The most involved case is the rule for function calls (T-FnCall), where we must slightly strengthen the non-interference definition for expressions, statements, and declarations. □

5 Implementation and Case Studies

To evaluate our type system, we implemented a type-checker for annotated P4 programs and used it to analyze a range of example programs exhibiting different kinds of errors. We call our tool P4BID. Our information-flow control type system is implemented as an extension of the type checker in the p4c compiler [26], the reference compiler for $P4_{16}$ [25].

$$\frac{}{\Gamma, \Delta \vdash_{pc} \langle \tau, \chi \rangle\ x \dashv \Gamma[x : \langle \tau, \chi \rangle], \Delta} \text{ T-VarDecl} \qquad \frac{\Gamma, \Delta \vdash_{pc} exp : \langle \tau', \chi \rangle \qquad \Delta \vdash \tau \rightsquigarrow \tau'}{\Gamma; \Delta \vdash_{pc} \langle \tau, \chi \rangle\ x := exp \dashv \Gamma[x : \langle \tau', \chi \rangle]; \Delta} \text{ T-VarInit}$$

$$\frac{\begin{array}{c} \Gamma, \Delta \vdash_{pc_{tbl}} \overline{exp_k : \langle \tau_k, \chi_k \rangle} \qquad \Gamma, \Delta \vdash_{pc_{tbl}} \overline{x_k : \langle match\_kind, \bot \rangle} \qquad \chi_k \sqsubseteq pc_{tbl} \text{ for all } k \\ \Gamma, \Delta \vdash_{pc_{tbl}} act_{a_j} : \langle \overline{d \langle \tau_{a_{ji}}, \chi_{a_{ji}} \rangle}\ ; \overline{\langle \tau_{c_{ji}}, \chi_{c_{ji}} \rangle} \xrightarrow{pc_{fn_j}} \langle unit, \bot \rangle, \bot \rangle \qquad pc_a \sqsubseteq pc_{fn_j} \text{ for all } j \\ \Gamma, \Delta \vdash_{pc_{tbl}} \overline{exp_{a_{ji}} : \langle \tau_{a_{ji}}, \chi_{a_{ji}} \rangle \textit{ goes } d} \qquad \chi_k \sqsubseteq pc_{fn_j} \text{ for all } j, k \qquad pc_{tbl} \sqsubseteq pc_a \end{array}}{\Gamma, \Delta \vdash_{pc} \text{table } x\ \{\overline{exp_k : x_k}\ \overline{act_{a_j}(\overline{exp_{a_{ji}}})}\} \dashv \Gamma[x : \langle table(pc_{tbl}), \bot \rangle], \Delta} \text{ T-TblDecl}$$

$$\frac{\begin{array}{c} \Gamma_1 = \Gamma[\overline{x_i : \langle \tau'_i, \chi_i \rangle}, \text{return} : \langle \tau'_{ret}, \chi_{ret} \rangle] \qquad \Gamma_1, \Delta \vdash_{pc_{fn}} stmt \dashv \Gamma_2, \\ \Delta \vdash \tau_i \rightsquigarrow \tau'_i \text{ for each } \tau_i \qquad \Delta \vdash \tau_{ret} \rightsquigarrow \tau'_{ret} \qquad \Gamma' = \Gamma[x : \langle \overline{d\ \langle \tau'_i, \chi_i \rangle} \xrightarrow{pc_{fn}} \langle \tau'_{ret}, \chi_{ret} \rangle, \bot \rangle] \end{array}}{\Gamma, \Delta \vdash_{pc} \text{function } \langle \tau_{ret}, \chi_{ret} \rangle\ x\ (\overline{d\ x_i : \langle \tau_i, \chi_i \rangle})\{stmt\} \dashv \Gamma', \Delta} \text{ T-FuncDecl}$$

Figure 7. IFC Typing Rules for Declaration

The target of our type checker is the simple_switch based on the BMv2 behavioral model. Our implementation adds about 700 LOC to P4C and supports the $\mathcal{L} = \{\text{high}, \text{low}\}$ lattice, and a simple diamond lattice from Figure 8b, $\mathcal{L} = \{\text{high}, \text{alice}, \text{bob}, \text{low}\}$ for modeling isolation specifications. Standard P4 types can be annotated with a security label from the lattice; unannotated types default to low.

We evaluate our implementation by comparing the type-checking time of the secure programs presented in the case studies using the P4BID typechecker with the typechecking time of their uninstrumented insecure counterparts using the original P4C compiler. Table 1 shows that our implementation incurs an overhead of 5% (or 30ms) on average in comparison to the reference P4C compiler when evaluated on the instrumented and uninstrumented versions of the same program. We believe this overhead is reasonable for an unoptimized implementation that builds on the stock p4c compiler; developing a more optimized implementation is a direction for future work.

Table 1. Typechecking time in milliseconds.

Program	Unannotated, p4c	Annotated, P4BID
D2R	534	599
App	593	600
Lattice	495	527
Topology	554	591
Cache	538	550
Average	543	573

In the rest of the section, we present our case studies.

5.1 Dataplane Routing with Priorities

In traditional networks, the control plane is responsible for *routing*, determining how to send a packet from source to destination, while the data plane is responsible for *forwarding*, sending a packet to its next hop. Subramanian et al. [31] have shown that using programmable switches, one can handle routing in the data plane, avoiding the control plane entirely. In their scheme, called D2R, when a switch receives a packet, it uses pre-loaded information about the network topology and local knowledge about link failures to perform a breadth-first search (BFS) and find a path to the target destination address. D2R uses P4 mechanisms (e.g., stacks) to perform the BFS computation entirely on the switch, without needing to communicate with the control plane.

We consider an extension of D2R where packets that encounter a higher number of link failures will receive higher priority. Listing 3 gives schematic code for the main headers and control block implementing this variant of data plane routing. The bfs_t headers describe the auxiliary information carried in the packets to perform the BFS, e.g., which links have been tried, while the ipv4_t headers contain information for standard packet forwarding. In the control block D2R_Ingress, the number of failures count (Line 19) can be computed from the vector of links that have been tried, hdr.bfs.tried_links, and the number of traversed links, hdr.bfs.num_hops. The table bfs_step performs one step of BFS; the details are not important for our purposes. Since P4 does not support loops, an iterative search algorithm like BFS is modeled in the apply block on Line 35 by unrolling the loop. If the BFS search has not completed, *i.e.,* the current node in the BFS search is not the destination node (Line 37), the BFS table is applied again (we elide the details of this BFS search algorithm which can be found in [31]). When the BFS search has successfully completed (Line 39), the forwarding table is applied and packet priorities are assigned based on the number of failures encountered by the packet.

Using failure information to prioritize packets may leak information. For instance, there are several potential reasons why hdr.bfs.num_hops could be secret—e.g., the packet could be transiting a private network and one might not want to

Listing 3. D2R: Dataplane Routing

```
1  header bfs_t {
2    <bit<32>, low> curr;
3    <bit<32>, low> tried_links;
4    <bit<32>, high> num_hops;
5    // ...
6  }
7  header ipv4_t {
8    <bit<3>, low> priority;
9    // ...
10 }
11 struct headers {
12   bfs_t bfs;
13   ipv4_t ipv4;
14   // ...
15 }
16
17 control D2R_Ingress(headers hdr) {
18   <bit<32>, high> failures
19   = num_bits_set(hdr.bfs.tried_links) − hdr.bfs.num_hops;
20
21   table bfs_step { ... }
22   table forward {
23     key = { hdr.bfs.next_node: exact; }
24     actions = { forwarding(failures); NoAction; }
25   }
26   action forwarding(in <bit<32>, high> failures) {
27     if (failures >= THRESHOLD) {
28       hdr.ipv4.priority = PRIO_1; // Leak
29     }
30     else {
31       hdr.ipv4.priority = PRIO_2; // Leak
32     }
33     // ... normal forwarding logic ...
34   }
35   apply {
36     if (hdr.bfs.curr != hdr.ipv4.dstAddr) {
37       bfs_step.apply();
38     } else {
39       forward.apply();
40     }
41     // repeat applications of bfs
42   }
43 }
```

reveal whether the network has reliable or unreliable links. If hdr.bfs.num_hops is annotated as high security, the program is rejected by our typechecker because the forwarding action writes data to the low-security priority after branching on the number of the failures, which is high security (Lines 28 and 31). This is an example of an indirect leak: the program branches on the secret, and then writes to public fields.

To remedy this information leak, we can modify the scheme so that the priority is computed based on non-sensitive information. For instance, we can assign priority based on the total number of links that a packet tried to cross. This count is an approximate proxy for the number of failures: as the number of failures rises, the packet tries more links. This change can be implemented by removing hdr.bfs.num_-hops in Line 19, giving a program that is accepted by our typechecker.

A similar kind of leak can manifest in the implementation of NetChain [18], an in-network implementation of chain replication on top of a key-value store. The implementation assigns roles to the various switches in the network to determine the head, tail, or internal nodes of the chain, which among various actions determines if the node sends out a reply or not. If the roles header field is labeled as a secret field, this can give away private topological information. When instrumented with a high label on role, the typechecker flagged implicit leaks in the implementation.

5.2 Modeling Timing for In-Network Caching

Like other IFC systems, our type system can model different notions of adversary-observable data. For an example, we can consider a key-value store with an in-network cache [19]. These systems are a prominent application of data plane computing: switches can quickly retrieve hot items, keep track of which items are frequently requested, and notify the controller about which items should be stored on the switch. While the result of a query should be the same no matter where the item is stored, an observer may be able to detect variations in timing: data that is stored on the switch is returned faster, while data that is stored on the controller takes longer to access. In some cases, this *timing side-channel* may allow an adversary to learn about the state of the system.

While Core P4 does not model timing aspects of program behavior, we can still model timing information leaks by augmenting the program with new variables holding data that a timing-sensitive adversary may be able to observe. For example, Listing 4 gives a schematic P4 program implementing a simple cache. The switch first tries to fetch data locally (Line 16). If the request hits then the table runs action cache_hit, while if the request misses then the table runs action cache_miss. Both actions record the hit or miss in hdr.resp.hit. We mark this field as a low-security (publicly visible) variable, to model an adversary who can distinguish whether a request was serviced by the cache or the controller. If the query is sensitive information, hdr.req.query is declared as high security. Our typechecker rejects this program because of an information leak: the actions cache_hit and cache_miss write to the low-security field hdr.response.hit (Lines 8 and 10), but they are invoked in a table with a high-security key hdr.req.query (Line 12). This is again an indirect leak, modeling a simple timing side-channel.

5.3 Preventing Manipulation in Resource Allocation

The examples we have seen so far use IFC to guarantee confidentiality: secret information (high) should not leak into

Listing 4. In-network cache

```
header request_t { <bit<8>, high> query; }
header response_t { <bool, low> hit; <bit<32>, low> value; }
struct headers { request_t req; response_t resp; eth_t eth; }

control Cache_Ingress(headers hdr) {
  action cache_hit(<bit<32>, low> value) {
    hdr.resp.value = value;
    hdr.resp.hit = true;
  }
  action cache_miss() { hdr.resp.hit = false;  }
  table fetch_from_cache {
    key = { hdr.req.query: exact; }
    actions = { cache_hit; cache_miss; }
  }
  apply {
    fetch_from_cache.apply();
    // ... if miss, try to fetch from controller ...
  }
}
```

publicly visible outputs (low). As is well-known, if we interpret high-security data as "untrusted" and low-security data as "trusted", IFC systems can also ensure *integrity*: untrusted inputs should not affect trusted outputs. To demonstrate, suppose several applications are running on separate subnetworks behind a single gateway switch, which is responsible for forwarding packets to their destination subnetwork and allocate resources to the application flows. We consider a very simple form of resource allocation, where a switch caters to the needs of latency-sensitive applications by increasing the priority of packets belonging to such applications. The P4 program in Listing 5 gives the main logic for a gateway switch that accomplishes this task. In addition to ordinary IP headers, packet headers in this setting also include an application ID hdr.app.appID indicating which application the packet belongs to. In the control block, the table app_resources matches on the application ID, and then calls set_priority with the desired priority level. This action then sets the priority level of the packet by writing to hdr.ipv4.priority (Line 15). Finally, the switch forwards the packet to the destination address hdr.ipv4.dstAddr.

While this program behaves well when clients are honest, a malicious client may manipulate the switch to increase the priority of their packets. Specifically, since hdr.app.appID is used to determine priority but not used to forward the packets, a client may report a false application ID. This issue can be detected by our IFC system if we label hdr.app.appID as untrusted (high) and hdr.ipv4.priority as trusted (low): setting priority based on application ID is an information-flow violation.

To address this problem, we can set the priority based on the destination address instead, by matching on hdr.ipv4.dstAddr instead of hdr.app.appID on Line 18. It is reasonable to model

Listing 5. Resource Allocation

```
1   header app_t { <bit<8>, high> appID; }
2   header ipv4_t {
3     <bit<32>, low> dstAddr;
4     <bit<32>, low> priority;
5     // ...
6   }
7   struct headers {
8     app_t app;
9     ipv4_t ipv4;
10    // ...
11  }
12
13  control App_Ingress(headers hdr) {
14    action set_priority(<bit<3>, low> priority) {
15      hdr.ipv4.priority = priority;
16    }
17    table app_resources {
18      key = { hdr.app.appID: exact; }
19      actions = { set_priority; }
20    }
21    apply {
22      set_priority.apply();
23      // ... forward the packet to hdr.ipv4.dstAddr ...
24    }
25  }
```

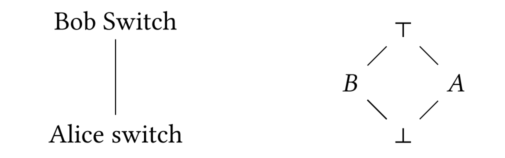
this header as trusted (low) because if a client were to manipulate this data, the packet would be delivered to the wrong destination. In the modified program, the priority is now only computed based on trusted data in hdr.ipv4.dstAddr and the typechecker accepts this program because there is no integrity violation.

5.4 Ensuring Network Isolation

The previous example changes the interpretation of security labels in order to establish different properties with IFC. For our final case study, we show how our type system can use a richer lattice to enforce network isolation properties.

Suppose we have a private network used by two clients, Alice and Bob, who run dataplane programs on two separate nodes (the precise topology is not important, but a sketch can be see in Figure 8a). Nodes pass around a shared packet header with separate fields for Alice and for Bob, and we want to ensure that Alice does not touch Bob's fields, and vice versa. Furthermore, the network operator wants to carry telemetry data alongside the packets (*in-band network telemetry* [17]) this data may depend on Alice or Bob's data, but neither Alice nor Bob should be able to use telemetry data.

We can model this isolation property as non-interference with a four-point diamond lattice with labels $\{A, B, \top, \bot\}$ (Figure 8b). Non-interference ensures that data from level χ can flow to variables labeled χ' if and only if $\chi \sqsubseteq \chi'$. Thus, if we label Alice's fields A and label Bob's fields B, then Alice's



(a) Network Topology **(b)** Security Lattice

Figure 8. Security lattice for a network topology

data cannot influence Bob's fields, and vice versa. Similarly, ⊤-labeled fields can depend on all data, but cannot influence data below ⊤. For instance, telemetry data can be labeled ⊤: both Alice and Bob can accumulate data into ⊤-labeled fields (e.g., increment a counter), but neither Alice nor Bob are able to leak information from ⊤-labeled data into their own fields. Finally, fields labeled ⊥ contain globally visible data that cannot depend on other fields above ⊥. For example, we can pre-configure a packet's route through the private network in ⊥-labeled fields: this ensures that information from Alice or Bob does not influence routing, potentially leading to an indirect leak or isolation failure.

Labeling data from the four-point lattice can already rule out many kinds of leaks. However, it still allows some leaks involving ⊥-labeled data. For instance, Alice may write Bob's fields with ⊥-labeled data, while Bob may use ⊥-labeled data to modify ⊥-labeled data. While potentially undesirable, neither of these actions violates IFC since high data is allowed to depend on low data. To prevent these behaviors, we can additionally typecheck Alice's code with pc label A, and typecheck Bob's code with pc label B. Then, non-interference guarantees that Alice can only write to fields labeled A or ⊤, and Bob can only write to fields labeled B or ⊤.

Listing 6 shows schematic versions of programs implementing the Alice and Bob switches. Both the switches have a single action. The packet header carries one of the four security labels. In this example, we consider that hdr.alice_data and hdr.bob_data are Alice's and Bob's data, respectively; hdr.eth cannot be updated by either switch, but it can be used by both the switches; and hdr.telem can be updated by any switch but it should not be visible to Alice or Bob. Then, isolation can be established by checking two judgements:

$$\Gamma, \Delta \vdash_A \text{update\_by\_alice}() \dashv \Gamma'$$
$$\Gamma, \Delta \vdash_B \text{update\_by\_bob}() \dashv \Gamma'$$

Programs that incorrectly access packet headers will be flagged by the typechecker. For instance, in Alice_Ingress, the switch tries to write to Bob's field, Line 12 and on Line 16 it attempts to use the telemetry field hdr.telem, which can only be written to, not read. Our typechecker flags both leaks. A safe version of Alice's switch program is shown in Listing 7. In contrast, Bob_Ingress is accepted by the typechecker: it applies a table that branches on the ⊥-labeled header hdr.eth,

Listing 6. Network Isolation and Telemetry

```
1  struct headers {
2    <alice_t, A> alice_data;
3    <bob_t, B> bob_data;
4    <telem_t, top> telem;
5    <eth_t, bot> eth;
6  }
7
8  // typed at pc = A
9  control Alice_Ingress(headers hdr) {
10   action set_by_alice(<bob_t, A> value) {
11     // Error: should not have written to Bob's field
12     hdr.bob = value;
13   }
14   table update_by_alice {
15     // Error: should not have used telemetry field
16     key = { hdr.telem: exact; }
17     actions = { set_by_alice; }
18   }
19   apply { update_by_alice.apply(); }
20 }
21
22 // typed at pc = B
23 control Bob_Ingress(headers hdr) {
24   action set_by_bob() {
25     // Allowed: modify telemetry using telemetry
           information
26     hdr.telem = hdr.telem + 1;
27   }
28   table update {
29     key = { hdr.eth.dstAddr: exact; }
30     actions = { set_by_bob; NoAction; }
31   }
32   apply { update_by_bob.apply(); }
33 }
```

Listing 7. Isolation Respecting Switch Program

```
// typed at pc = A
control Alice_Ingress(headers hdr) {
  action set_by_alice(<alice_data, A> value) {
    hdr.alice_data = value;
  }
  table update_by_alice {
    key = { hdr.alice_data: exact; }
    action = { set_by_alice; }
  }
  apply { update_by_alice.apply(); }
}
```

and the action set_by_bob only modifies the ⊤-level header hdr.telem, incrementing a counter.

While our concrete example only involves two switches and two parties, the same idea can be directly generalized to more parties by adding additional labels at the level of A and B. Then, our typechecker can ensure that programs written by different parties act on only their own packet headers. Richer dataflow policies could potentially be enforced by using more complex lattices; this is an interesting direction for future work.

6 Related Work

Security in programmable networks. Recent works explore the security and privacy implications of programmable networks. For instance, in-network systems can be used to defend against denial-of-service attacks [38, 39], obfuscate network topology [23], mitigate covert channels [37], and enforce custom security policies [21, 31]. Tools have also been developed for helping operators test their dataplane programs against adversarial inputs (e.g., [20]). Our work complements these systems by detecting security and privacy bugs in programs running on programmable switches.

Network verification. The network verification literature is too vast to summarize here; methods have have targeted many aspects of networked systems, including routing protocols (e.g., [2–4, 36]), network configurations (e.g., [5, 29]), and network controllers (e.g., [7, 15]). Techniques have also been developed for verifying dataplane programs (e.g., [1, 14]). Some works also allow one to automatically repair faulty configurations [30] or to automatically synthesize policy-compliant ones [32, 33].

Our work focuses on dataplane programs written in the P4 language [6], building on the core version of P4 developed by Doenges et al. [10]. Perhaps the most closely related work is p4v [22], a verification system for P4 programs. Using p4v, a P4 program is verified against a logical specification by extracting a logical formula, which can be dispatched to solvers like Z3. Liu et al. [22] use p4v to verify basic correctness properties, e.g., a program does not read or write invalid headers, or a program implements the desired functionality correctly. While our system cannot verify the general properties established by p4v, our target non-interference property cannot be established in p4v since it relates a program's behavior on pairs of inputs [8]. Furthermore, our type-based analysis is lightweight and does not require automated solvers.

Two closely related type-system based works that explore properties orthogonal to non-interference properties are SafeP4 [11] and Π4 [12]. SafeP4 aims at catching invalid header access bugs, while Π4 presents a dependently-typed extension of P4 for verifying richer properties that SafeP4 could not cover. Unlike Π4, P4BID has a light-weight type-checking algorithm that does not involve constraint solving. Furthermore, our system builds on Core P4, a more realistic formal model of P4. For example, Core P4 models different calling conventions of P4 functions (e.g., pass by value and pass by reference) and control flow signals. These features introduced new opportunities for implicit leaks, which our type system rules out.

Information-flow control. Our approach belongs to a line of research on information-flow control (IFC), a type-based method of expressing and verifying a wide variety of security properties. Starting from work by Denning [9] and Volpano et al. [35], there are now many information-flow control systems ensuring different variants of non-interference against different kinds of adversaries; the survey by Sabelfeld and Myers [28] is a good introduction to this area. Existing systems target general-purpose programming languages (e.g., [24, 27]). Our work brings this idea to languages for programmable networks.

7 Conclusion and Future Directions

We have designed an information-flow control type system for P4 and demonstrated how it can verify networking properties for programs running on programmable switches.

We see several possibilities for further investigation. First, our non-interference theorems treat P4 programs as mapping a single input packet to a single output packet, but,P4 allows programming switches that can maintain internal state and recirculate packets for additional processing. These features could lead to security leaks if an adversary can observe sequences of input and output packets, and it would be interesting to establish non-interference in this richer setting. Second, it could be interesting to refine our analysis with information or assumptions about the control plane [22].

Acknowledgments

This work benefited substantially from discussions about P4 and Core P4 with Eric Campbell, Ryan Doenges, and Nate Foster. We thank the reviewers and our shepherd, Jedidiah McClurg, for their close reading and constructive feedback. This work is partially supported by NSF grants #2152831 and #1943130.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*. 113–126. https://doi.org/10.1145/2535838.2535862

[2] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Los Angeles, California*. 155–168. https://doi.org/10.1145/3098822.3098834

[3] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control plane compression. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Budapest, Hungary*. 476–489. https://doi.org/10.1145/3230543.3230583

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 42:1–42:27. https://doi.org/10.1145/3371110

[5] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin T. Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations. In *USENIX Symposium on Networked Systems*

Design and Implementation (NSDI), Santa Clara, California. 969–984. https://www.usenix.org/conference/nsdi20/presentation/birkner

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

[7] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. 2021. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 133–153. https://www.usenix.org/conference/nsdi21/presentation/campbell

[8] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010), 1157–1210. https://doi.org/10.3233/JCS-2009-0393

[9] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243. https://doi.org/10.1145/360051.360056

[10] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434322

[11] Matthias Eichholz, Eric Hayden Campbell, Nate Foster, Guido Salvaneschi, and Mira Mezini. 2019. How to Avoid Making a Billion-Dollar Mistake: Type-Safe Data Plane Programming with SafeP4. In *European Conference on Object-Oriented Programming (ECOOP), London, England (Leibniz International Proceedings in Informatics, Vol. 134)*. 12:1–12:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.12

[12] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-Typed Data Plane Programming. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 40 (Jan. 2022), 28 pages. https://doi.org/10.1145/3498701

[13] Facebook. 2021. More details about the October 4 outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/

[14] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*. 343–355. https://doi.org/10.1145/2676726.2677011

[15] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: an intermediate language for verification of network control planes. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), London, England*. 958–973. https://doi.org/10.1145/3385412.3386019

[16] Karuna Grewal, Loris D'Antoni, and Justin Hsu. 2022. *P4BID: Information Flow Control in P4 (Extended Version)*. Technical Report. arXiv:2204.03113 [cs.PL] https://arxiv.org/abs/2204.03113

[17] Intel. 2020. *In-band Network Telemetry Detects Network Performance Issues*. Technical Report. Intel. https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI), Renton, Washington*. USA, 35–49.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI), Shanghai, China*. 121–136. https://doi.org/10.1145/3132747.3132764

[20] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic profiling of stateful data planes for adversarial testing. In *International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS)*. 286–301. https://doi.org/10.1145/3445814.3446764

[21] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *USENIX Security Smposium (USENIX)*. 595–612. https://www.usenix.org/conference/usenixsecurity20/presentation/kang

[22] Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), Budapest, Hungary*. 490–503. https://doi.org/10.1145/3230543.3230582

[23] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin T. Vechev. 2018. NetHide: Secure and Practical Network Topology Obfuscation. In *USENIX Security Smposium (USENIX), Baltimore, Maryland*. 693–709. https://www.usenix.org/conference/usenixsecurity18/presentation/meier

[24] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. *Jif 3.0: Java information flow*. http://www.cs.cornell.edu/jif

[25] P4Lang. 2022. P4_16 Spec. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html

[26] P4Lang. 2022. P4c Compiler. https://github.com/p4lang/p4c

[27] François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems* 25, 1 (2003), 117–158. https://doi.org/10.1145/596980.596983

[28] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[29] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2020. Probabilistic Verification of Network Configurations. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 750–764. https://doi.org/10.1145/3387514.3405900

[30] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2020. Detecting network load violations for distributed control planes. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), London, England*. 974–988. https://doi.org/10.1145/3385412.3385976

[31] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2021. D2R: Policy-Compliant Fast Reroute. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*. 148–161. https://doi.org/10.1145/3482898.3483360

[32] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: synthesizing forwarding tables in multi-tenant networks. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France*. 572–585. https://doi.org/10.1145/3009837.3009845

[33] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2018. Synthesis of Fault-Tolerant Distributed Router Configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 22:1–22:26. https://doi.org/10.1145/3179425

[34] Steven J. Vaughan-Nichols. 2021. Google glitch triggers major internet outage. *ZDNet* (Nov. 2021). https://www.zdnet.com/article/google-glitch-triggers-major-internet-outage/

[35] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304

[36] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM*

SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Amsterdam, The Netherlands. 765–780. https://doi.org/10.1145/2983990.2984012

[37] Jiarong Xing, Adam Morrison, and Ang Chen. 2019. NetWarden: Mitigating Network Covert Channels without Performance Loss. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Renton, Washington.* https://www.usenix.org/conference/hotcloud19/presentation/xing

[38] Jiarong Xing, Wenqing Wu, and Ang Chen. 2019. Architecting Programmable Data Plane Defenses into the Network with FastFlex. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), Princeton, New Jersey.* 161–169. https://doi.org/10.1145/3365609.3365860

[39] Jiarong Xing, Wenqing Wu, and Ang Chen. 2021. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *USENIX Security Smposium (USENIX).* 3865–3881. https://www.usenix.org/conference/usenixsecurity21/presentation/xing