Foundationally Verified Data Plane Programming

Qinshi Wang

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Prof. Andrew W. Appel

September 2023

Abstract

P4 is a major standardized programming language for programming and specifying the network data plane. P4 is widely used in a variety of network functionalities, including monitoring, traffic management, forwarding, and security. Recently, stateful applications have been emerging in this area, as supported by programmable hardware. Typical stateful applications include network telemetry (heavy hitters, distributed denial-of-service (DDoS) detection, performance monitoring), middleboxes (firewalls, network address translation (NAT), load balancers, intrusion detection), and distributed services (in-network caching, lock management, conflict detection). Their complexity and rich properties are beyond the ability of existing P4 verifiers.

In this thesis, we propose Verifiable P4: a new framework for P4 program verification based on interactive theorem proving that is (1) capable of proving multi-packet properties, (2) modular in terms of the structure of P4 programs, and (3) foundationally sound with respect to a mechanized formal semantics of P4. In order to achieve these goals, we built (1) a mechanized formal semantics of P4 more comprehensive and convenient than existing formal semantics, (2) a set of program logic rules that are proven sound, and (3) an interactive verification system based on the program logic and Coq tactic mechanism. We verified a stateful firewall fully implemented in P4 using a sliding-window Bloom filter with Verifiable P4 and evaluated its utility.

Acknowledgements

It was a long and challenging journey to pursue a Ph.D. degree, and it is my fortune to have received support from so many people.

First, I would like to express my gratitude my advisor, Andrew W. Appel, who led me to the fantastic field of software verification, and provided great guidance during my Ph.D. studies. In Spring 2018, when I was looking for a research direction that is both theoretically interesting and practically useful, I took his course on theorem proving and programming languages and fell in love with this field. As my advisor, he is a smart guide on research, a patient mentor on writing, and an invaluable cornerstone in my academic journey.

I appreciate my thesis committee, Nate Foster, David Walker, Jennifer Rexford, and Zachary Kincaid. Their insightful feedback and constructive criticism have significantly enhanced my research and the quality of this thesis.

I am grateful to Qinxiang Cao, who has been a great model of success and mentor for me in both competitions and research since my high school days.

I would like to acknowledge my collaborators, Andrew W. Appel, Lennart Beringer, Qinxiang Cao, Joshua Cohen, Ryan Doenges, Mengying Pan, Rudy Peterson, Vilhelm Sjöberg, and Shengyi Wang. It was my honor and pleasure to have worked with them.

I am thankful to my friends, Tingting Cai, Xiaoqi Chen, Jiaxin Guan, Guanhua He, Zhongtian He, Junnan Hu, Renzhi Jing, Changshuo Liu, Shang Mu, Yuqi Nie, Liqun Peng, Zhiyi Ren, Weiyi Tang, Chenggong Wang, Lianyong Wang, Gaoyuan Wu, Yantao Wu, Zhelun Wu, Dazhi Xi, Zhaojian Xu, Weikun Yang, Conghao Yi, Qiang Zhang, Wenda Zhang, Zidong Zhang, Yuyan Zhao, and Yuqing Zhu, who have made my life brighter and more colorful.

I would like to express my gratitude to my family friends Lan Bin and Fred Ling, who have offered guidance and care as if I were their nephew.

Finally, I wish to express my most profound appreciation to my parents, Qian Xu and Weibing Wang. Their unwavering love, boundless support, endless patience, and constant belief in my dreams have been the cornerstone of my academic journey. Especially when the COVID-19 pandemic brought unprecedented challenges that destabilized my life, their guidance and reassurance were a beacon of hope, enabling me to persevere and continue pursuing my goals with determination in those trying times.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001120C0160.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, computer networking is indispensable almost everywhere. Network hosts are connected through network switches. Traditionally, network switches are fixed-function devices, whose behavior is specified and implemented by their manufacturers. In recent years, people have become interested in programming languages for switches, for two reasons: first, in order to secure the network, it is worth specifying and reasoning about networks, including the behavior of switches and links; second, programmable switches are prevalent in scenarios demanding flexibility, especially since programmable switches have evolved to be as fast as fixed-function switches, we need a programming language for them.

P4 [5, 12] is the most widespread programming language for specifying and programming switches. [22] But P4 is a low-level language, and P4 programs are often contorted to fit within the constraints of resources in a particular target architecture, which is designed to process billions of packets per second. So the correctness of these programs has become a concern. To address that concern, there are several verification tools for P4 programs (see Section 1.2).

In many classic P4 applications, processing a packet does not typically change the state of the switch. But recently there are new applications for programmable

data planes, in which each packet changes the state of the switch and affects how the following packets are processed. These applications include network telemetry systems (SketchLib [31], BeauCoup [9], FlowRadar [27]), network functions (SilkRoad [30]), and distributed services (NetCache [24], NetLock [46]). Misconfiguration in such programs may lead to serious network failures. But the existing verification tools cannot reason about the packet-to-packet state changes of these *stateful* programs.

For example, consider a stateful firewall that protects the internal network from unsolicited traffic. External packets should be permitted to enter through the firewall only if they are responses to recent outgoing requests to the same IP address. The stateful firewall remembers recent outgoing packet headers. We need to verify a property about multiple packets: *no valid incoming responses are blocked*. A small rate of false positives is tolerated, i.e., allowing incoming packets that are not responses.

The multi-packet property that we want to verify can be formally written as follows. Let T be the valid response time window, h be the list of historical packets, p be the current packet and r be the action on p (forwarding or dropping it). We want that for every integer i, we have

$$p.\text{dir} = \text{in} \ \wedge \ h[i].\text{dir} = \text{out} \ \wedge \ h[i].\text{dst} = p.\text{src} \ \wedge \\ h[i].\text{src} = p.\text{dst} \ \wedge \ p.t - h[i].t \leq T \implies r = \text{forward}.$$

To verify such programs, we need stateful reasoning. None of the existing verifiers can properly characterize the way the switch's state changes per packet, either because they don't handle state at all, or because their specification languages are too weak to properly relate the pre-state to the post-state after processing a packet (see Section 1.2); and none of existing verifiers can reason about such multi-packet properties.

Arbitrary multi-packet properties cannot be proved fully automatically.[1] So we embed our verifier in an interactive proof assistant (Coq)—this allows a very general-purpose logic in which almost any kind of mathematics can be expressed. Inevitably, in such general math one cannot always get 100% proof automation. To minimize proof workload, we use Coq's programmability to automate where possible (matching P4 to functional models) and we use interactive proof where necessary (proving multi-packet properties from per-packet state changes).

Modular proofs clarify protocols between modules and make modules reusable. In order to modularize P4 programs with stateful objects, we propose a hierarchical representation of states used in semantics, specification, and verification that improves modularity; unlike some previous P4 semantics, we enforce a phase distinction between *instantiation* (that populates this hierarchy) and *run-time packet processing*.

All programs, including verification tools, can have bugs. P4 verifiers with bugs can "verify" something that isn't true. The correctness concern of P4 verifiers has not been fully addressed in the previous work (see Section 1.2). Our verifier has a once-and-for-all machine-checked *soundness* guarantee: if it can prove a property of your program, then your program behaves that way in the P4 semantics.

Contributions We have built Verifiable P4, which is a P4 verification system in Coq that supports very rich specifications, especially for stateful programs. The verification system also makes modular verification possible, and the verification result is foundational. We have applied Verifiable P4 to a stateful firewall purely implemented in P4 using a sliding-window Bloom filter, and proved the multi-packet property that solicited packets are never dropped. In particular, we make the following technical contributions:

[1] P4 programs does not have loops or recursions for a single packet, so it might be possible to automatically reason about single-packet processing. But lifting from single-packet processing to multi-packet processing involves induction, which is known to be difficult.

1. We revisit the formal semantics in Petr4 [18], and propose a phase distinction between *instantiation* (compile-time allocation) and *execution* (run-time packet processing). This phase distinction makes the semantics clearer and easier to reason about.
2. We built a mechanized semantics for P4 in Coq, guided by the $P4_{16}$ Language Specification [12].
3. We propose a hierarchy for states, semantics, specification, and verification that improves modular verification. We define *hierarchical predicates* to specify state of extern objects.
4. We demonstrate the program logic and verification tool by verifying a high-performance implementation of a stateful firewall that uses a sliding-window Bloom filter.

1.1 Introduction to P4

P4 is a standardized programming language for both programming programmable network switches and specifying nonprogrammable network switches. This section gives a brief introduction to readers not familiar with P4, by giving a walkthrough example and pointing out some key aspects of P4.

P4 runs on different hardware/software platforms, each of which is called a P4 *target*. A P4 *architecture* is the interface that describes the programming model of a target. A target may support multiple architectures, which makes the target compatible with programs written for different architectures. An architecture may be implemented by multiple targets, too.

Figure 1.1 illustrates the Very Simple Switch (VSS) architecture (adapted from the $P4_{16}$ Specification [12]), which is a P4 architecture for introductory purposes. It has three P4-programmable components: the parser, the match-action pipeline, and the

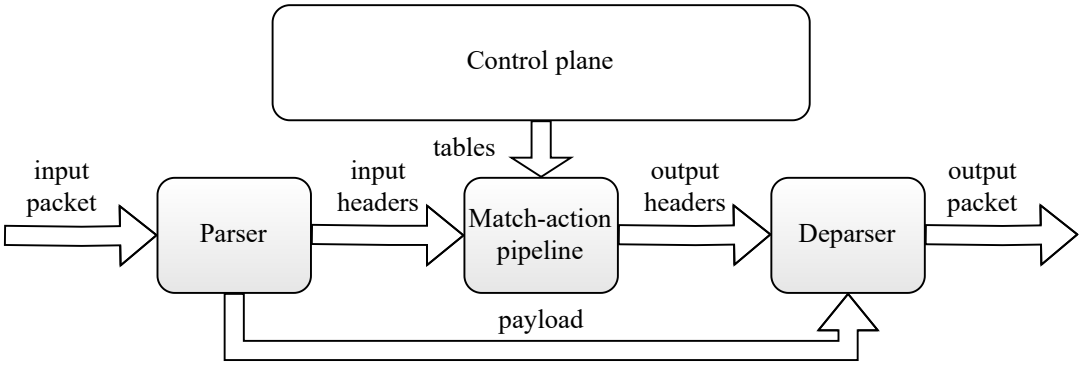

Figure 1.1: Very Simple Switch (VSS) architecture

deparser. The remaining parts of the switch are nonprogrammable. When a packet is processed by a VSS switch, it is first parsed by the programmable parser, and the headers are passed to the match-action pipeline while the rest of the packet, called the payload, is directly passed to the deparser. The match-action pipeline produces output headers by performing operations determined by the P4 program and tables. These tables can be dynamically updated by the control plane. The deparser then reassembles the output headers with the payload to form the output packet.

Figure 1.2 is the *architecture header file* of the VSS architecture, which should be included by every P4 program on VSS. For a real architecture, such a file should be provided by the manufacturer. It defines the *interface*[2] of each P4-programmable block, namely Parser, Pipe, and Deparser. Each P4-programmable block is an instance, which will be introduced later, but its entrance can be viewed as a P4 function,[3] and its function signature has to match the corresponding interface. These interfaces do not have return values. In fact, although there are some other functions in P4 that have return values, a different value-passing mechanism, called *copy-in/copy-out*, is consistently used instead of return values.

[2]They are called parser types/control types in the P4 specification.

[3]We use a more general definition of "function" than the P4 Specification. Any entity callable at runtime is regarded as a function.

```
#include <core.p4> // P4's standard core library

struct std_meta_t {
    bit<9> ingress_port;
    bit<9> egress_port;
}

parser Parser<H>(packet_in pkt, out H hdrs, inout std_meta_t std_meta);
control Pipe<H>(inout H hdrs, inout std_meta_t std_meta);
control Deparser<H>(packet_out pkt, in H hdrs);

package VSS<H>(Parser<H> p, Pipe<H> pipe, Deparser<H> dp);

extern Checksum16 {
    Checksum16(); // constructor
    void clear(); // reset checksum
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

Figure 1.2: Architecture header file of VSS

In the copy-in/copy-out mechanism, function parameters are usually directed, including in, out, inout. in parameters are normal parameters passed from the caller to the function; out parameters are similar to return values and passed from the function to the caller; inout parameters serve the role of both in and out parameters. In addition, there are *directionless* parameters, such as pkt in the interfaces Parser and Deparser. Directionless parameters are evaluated at compile time, so they must be compile-time known values. Unlike directional parameters, they can be objects, such as packet_in and packet_out, which represent the input packet and the output packet, respectively. Package interface VSS defines that a P4 program for a VSS switch consists of a Parser, a Pipe, and a Deparser.

The architecture header file also contains the interface of extern objects and functions available in the architecture. While the expressiveness of P4 is limited, each target hardware provides various additional functionalities, which are modeled as

extern objects and functions in P4. For example, VSS supports an extern object, Checksum16. The user can create an instance of Checksum16 and use its methods to calculate the checksum of a header. packet_in and packet_out are also extern objects, which are defined in core.p4. The library core.p4 is a standard file. It defines standard types of P4, and should be included by every P4 program.

The architecture can impose different restrictions on each programmable block, tailored to the hardware capabilities. For example, in VSS, Checksum16 can only be used in the parser and the deparser, and tables can only be defined in the match-action pipeline. Although these restrictions are not described in the header file, the compiler may reject programs that violate such restrictions. Also, the hardware has limited resources, so programs may be rejected by the compiler because it cannot find a layout to fit the program in the hardware.

In the following paragraphs, we show an example P4 program on the VSS architecture that performs the standard IPv4 forwarding, offering a concrete example for the fundamental concepts and structure of P4 programs.

```
1  #include <vss.p4>
2
3  header Ethernet_h {
4      EthernetAddress dstAddr;
5      EthernetAddress srcAddr;
6      bit<16> etherType;
7  }
8
9  struct hdrs_t {
10     Ethernet_h ethernet;
11     IPv4_h ip;
12 }
```

The P4 program starts by including the architecture description and defining header types. Ethernet_h is the Ethernet header, which consists of three fields. A header in P4 is like a struct,[4] except that it includes an extra bit indicating its

[4] struct in P4 is similar to struct in C.

validity. The fields of a header are meaningful only when the header is valid. We omit the definition of IPv4 header (IPv4_h). The type hdrs_t is a struct that includes the headers to manipulate for a packet.

```
13 parser MyParser(packet_in pkt, out hdrs_t hdrs, inout std_meta_t std_meta) {
14     Checksum16() ck; // instantiate checksum unit
15
16     state start {
17         pkt.extract(hdrs.ethernet);
18         transition select(p.ethernet.etherType) {
19             0x0800: parse_ipv4;
20         }
21     }
22
23     state parse_ipv4 {
24         b.extract(p.ip);
25         ck.clear();
26         ck.update(p.ip);
27         // Verify that packet checksum is zero
28         verify(ck.get() == 16w0, error.IPv4ChecksumError);
29         transition accept;
30     }
31 }
```

Then comes the parser declaration MyParser. A P4 parser is a state machine in which each state executes a program block and transitions to another state. It begins from the state called start. In this example, the parser extracts the Ethernet header from the packet, and examines the EtherType. It transitions to the parse_ipv4 state if the EtherType is 0x0800. Otherwise, the transition fails and the packet is rejected. In parse_ipv4, it extracts the IPv4 header from the packet and verifies its checksum. The headers as result of parsing are passed out through the out parameter hdrs.

```
32 control MyPipe(inout hdrs_t hdrs, inout std_meta_t std_meta) {
33     IPv4Address _nextHop;
34     action forward(IPv4Address nextHop, bit<9> port) {
35         _nextHop = nextHop;
36         std_meta.egress_port = port;
37         hdrs.ipv4.ttl = hdrs.ipv4.ttl − 1;
```

```
38      }
39
40      action drop() {
41          std_meta.egress_port = DROP_PORT;
42      }
43
44      table routing_table {
45          key = {
46              hdrs.ipv4.dstAddr: lpm;
47          }
48          actions = {
49              forward();
50              drop();
51          }
52          default_action = drop();
53      }
54
55      apply {
56          if (hdrs.ip.ttl ≤ 1)
57              drop();
58          routing_table.apply();
59          dmac_table.apply();
60          smac_table.apply();
61      }
62  }
```

We then have a control declaration MyPipe, which programs the match-action pipeline block. It analyzes the input headers and determines the packet's forwarding behavior. While it is simple in this example, this block is where the core logic is implemented in more sophisticated applications.

The match-action pipeline consists of match-action *tables*. Each table contains some entries, and invoking a table involves finding the entry that matches the keys, and executing the action specified by the entry. The entries may be either maintained dynamically by the control plane, or hardcoded in the P4 program as constants. In this example, routing_table is the conventional IPv4 routing table, whose entries are maintained by the control plane, so the entries are not in the P4 program and may be dynamically changed by the control plane between packets. The table examines the

destination address in the IPv4 header (as specified on line 46), and executes either the action forward or the action drop. The parameters of the action forward do not have direction annotations, which means they are given by the matched entry when the table invokes forward. At the end of MyPipe is the apply block (line 55), which is the main body of MyPipe. It also invokes tables dmac_table and smac_table, whose definitions in MyPipe are omitted.

```
63 control MyDeparser(packet_out pkt, in hdrs_t hdrs) {
64     Checksum16() ck;
65     apply {
66         pkt.emit(hdrs.ethernet);
67         if (hdrs.ip.isValid()) {
68             ck.clear(); // prepare checksum unit
69             hdrs.ip.hdrChecksum = 16w0;
70             ck.update(p.ip); // compute new checksum according to RFC 791
71             hdrs.ip.hdrChecksum = ck.get();
72         }
73         pkt.emit(p.ip);
74     }
75 }
```

We then have a control declaration MyDeparser, i.e. the deparser block. It takes the headers determined by the match-action pipeline to construct the output packet. In this block, the function pkt.emit inserts a header into the packet.

```
76 VSS(MyParser(), MyPipe(), MyDeparser()) main;
```

The code blocks MyParser, MyPipe, and MyDeparser are *declarations*, they need to be *instantiated* before being invoked. For example, the expression "MyParser()" creates an instance of MyParser, and this instance is used in the instantiation of the main package "VSS(...) main", which represents the switch in the P4 program. So the switch can invoke MyParser as its Parser block.

A parser or control declaration can be instantiated multiple times, so the instantiation mechanism allows the code to be reused modularly to create more complicated

programs. In order to understand instantiation, we can compare to object-oriented programming (OOP). Parser or control declarations are similar to classes in OOP, and instances are similar to objects. In this thesis, we use classes to refer to parser or control declarations, and use objects to refer to instances. We will discuss more details about instantiation in Section 2.2.

In summary, the aspects in which P4 differs from conventional imperative programming languages are: instantiation, copy-in/copy-out mechanism, parsers, tables, extern objects and functions.

We have given a brief introduction for readers unfamiliar with the P4 language to understand the content of this thesis smoothly. For more details about the P4 language, we refer to the P4 Specification [12]. For more applications of P4, we refer to a comprehensive survey paper [22].

1.2 Related Work

In this section, we summarize existing works on P4 program correctness. We will make more detailed comparisons of our results to other works in specific chapters of the thesis from the programming language and verification perspective.

1.2.1 Early P4 Verifiers

Bugs in P4 programs have been of concern since the invention of the language. Three first generation verifiers were built in 2018, including p4v [28], Vera [40], and ASSERT-P4 [33]. These three verifiers are generally similar: they are all automatic and monolithic tools to check properties less complex than full functional correctness for stateful programs.

For example, if a P4 program accesses a field of an invalid header or drops a packet due to a bug mishandling a packet field, these verifiers can identify the bug and generate a counterexample. However, these verifiers are not capable to express or verify properties that involve multiple packets, which are fundamental properties of stateful applications.

p4v is an automatic verifier for $P4_{14}$.[5] It verifies relatively simple properties, including general safety properties (e.g. header validity), architectural properties (e.g. parser roundtripping, which means the composition of deparsing and reparsing is the identity function), and simple application-specific properties that are expressible by Boolean logic and linear integer arithmetic. p4v translates P4 into guarded command language [17]. The translation procedure unrolls P4 parsers into straight-line code and rejects programs that contains unproductive cycles that do not extract any headers. From guarded command language, p4v generates verification conditions and checks them using Z3 [15].

The interesting technique is the treatment of tables that are filled by the control plane. The entries of these tables are not given in the P4 program. Instead, the control plane can modify them to change the packet processing policy, so we cannot determine the action invoked by the table. To address this problem, p4v uses a *control plane interface* to characterize the possibilities of each table, and includes all the possible execution traces for the verification. Vera also uses this technique, where it is called symbolic entries.

Vera is also an automatic verifier for $P4_{14}$ programs. It translates a P4 program into the SEFL language in Symnet [41], which is a symbolic execution system for network systems. It then uses Symnet to examine all possible execution paths of the P4

[5] $P4_{14}$ is the predecessor language to $P4_{16}$.

program. Vera automatically detects common mistakes and hazards, including implicit drops, manipulating dropped packets, invalid header access, out-of-bounds array accesses, and arithmetic overflows. Packets may be recirculated in the switch, i.e. the P4 program may send a packet back to some earlier stages to process the packet again. To reason about packet recirculation, Vera detects loops without progress, and supports specifications in a subset of computation tree logic (CTL).

ASSERT-P4 is a verification tool for $P4_{16}$, which translates the P4 program into a C program together with the specification to analyze using KLEE [6]. However, as we will show in Section 2.3, the semantics of C and the semantics of P4 are different, making this encoding inaccurate and probably causing false reports of bugs.

These three verifier have made their contributions to the correctness of P4 programs, but they do not handle stateful objects well. Although `p4v` supports stateful objects, we did not find any evidence that p4v can relate initial and final states. For example, it cannot express that the final state of a register is obtained by modifying a particular position from the initial state. Vera [40] uses an extremely expensive encoding that is proportional to the size of registers (impractical when the register contains an entire hash table). ASSERT-P4 does not claim to support stateful objects.

1.2.2 Aquila

Aquila [42] supports a more convenient assertion language, multi-pipeline control, more time-efficient verification, and bug localization when the verification claims a bug. But Aquila oversimplifies registers into fields without indexes, so it could not verify programs such as the stateful firewall. Aquila also reduces the risk of bugs in the verifier by translation validation—checking whether its intermediate representation is equivalent to the counterpart generated by Gauntlet [38] (which is a tool for finding

bugs in P4 compilers). But that does not address other software bugs, e.g. the bugs in manipulating assertions, especially when we need a rich and modular assertion language.

1.2.3 Π4

Π4 [20] is a research language and dependent refinement type system designed for a subset of P4. This subset involves headers, parsers, deparsers, but not `extern` objects and persistent state. The type system uses refinement types to represent constraints on values. For example, $\{y : \tau \mid 0 \leq y.\mathrm{a} < N\}$ represents that variable a is in $[0, N)$. A dependent type of the form $(x : \tau_1) \to \tau_2$ is used to define a type for a program segment, which means when starting with a state x that satisfies type τ_1, the program segment is safe and resulting state satisfies τ_2. In particular, x may be referred to in τ_2 to connect the initial and final states, e.g. τ_2 being $\{y : \tau \mid y.\mathrm{a} = x.\mathrm{a}\}$ means a's value stays the same.

Type checking in this system is similar to a verification task. Refinement types are assertions, and dependent types for program segments are function contracts. Π4 also uses an SMT solver to check these types. So Π4 is similar to a verifier. Comparing with previous verifiers, Π4 is compositional: once a program segment satisfies a function contract, one can use the contract to verify the whole program that uses the program segment. For example, this allows a device vendor to specify a fixed-function component such that it can be compositionally verified with user-defined components, and the vendor can modify the fixed-function component without breaking the verification, as long as the function contract of the fixed-function component still holds.

Π4 does not consider stateful objects, e.g. registers, so it cannot be used to reason about stateful behaviour. Also, it does not have scopes to keep variables private in modules.

1.2.4 Petr4

Petr4 [18] is a study of P4's formal semantics, which gives us an important reference. But Petr4's semantics does not have a machine-checkable formalization, and mixes instantiation and execution, which means it "instantiates at runtime" and has to define each control instance as a closure. This makes the semantics less straightforward and makes it much more challenging to prove the program logic and the type system sound. We improve this with a phase distinction between instantiation (Section 2.2) and execution. We will compare our formalization with Petr4 in much more detail in Section 2.4. We also identify and fix some bugs in Petr4 during the formalization in Coq.

Summary Although there are several previous P4 verifiers, none of them address the problem of writing and verifying nontrivial stateful programs, such as programs based on data structures (sliding-window Bloom filter, count-min sketch, etc.).

1.3 Organization

It is assumed that the reader has basic knowledge about formal operational semantics and program logic (e.g. Hoare logic). Readers may refer to textbooks [37, 36] for an introduction to these concepts.

The rest of the thesis is organized as follows. Chapter 2 presents our formalization and mechanization of P4's operational semantics (by the author and Mengying Pan). Chapter 3 shows the design of the Verifiable P4 program logic and its soundness (by the author, with assistance of Shengyi Wang). Chapter 4 describes the verification system based on the program logic and implemented using Coq's Ltac (by the author, with assistance of Mengying Pan and Shengyi Wang). Chapter 5 demonstrates an end-to-end verified stateful firewall that combines the P4 program verification using

the verifier and property proof in Coq (by the author, Shengyi Wang and Lennart Beringer). Chapter 6 concludes the thesis and discusses future work directions.

We described much of this work in a paper published in the 2023 Conference on Interactive Theorem Proving [45] in a short version.

Chapter 2

P4 Semantics

The term "foundationally verified" means that the correctness or desired properties of a program is verified with respect to its formal semantics. So the first step towards foundationally verifying P4 programs is to define the operational semantics of the P4 language. The authority document of P4 semantics is the $P4_{16}$ Specification [12] written by the P4 Committee. But it is a document written in natural language and pseudocode, instead of formal language. In order to clear the jungle, Doenges et al. made an important initiative with Petr4 [18], a pen-and-paper formalization of P4's operational semantics. But Petr4 is not implemented in a proof assistant, so it does not support mechanized proofs. Petr4 semantics also left some unresolved ambiguities and unsupported P4 features.

In this chapter, we describe how we built a mechanized formalization of P4 semantics by inheriting and improving Petr4. We introduce a phase distinction in P4's semantics that separates it into the instantiation phase (Section 2.2) and the execution phase (Section 2.3). This phase distinction mimics the actual deployment of P4 programs: the compiler evaluates the P4 program itself, allocates hardware resources, and produces a hardware program, and the hardware processes the packets without higher-order computation and resource allocation. The instantiation phase

and the execution phase correspond to the compiler and the hardware, respectively. Section 2.4 compares with Petr4's semantics and explains the benefit of this approach.

P4 does not have C-style undefined behavior that permits completely arbitrary results. Instead, it produces unspecified values when reading from uninitialized fields. However, an easily overlooked point in P4's semantics is that reading an uninitialized field twice may yield different values [12, Section 8.23]. We implemented this feature in the execution phase (Section 2.3).

P4 is a language for programming and specifying a variety of hardware and software targets. A P4 *architecture* is a model for a set of compatible targets. The internal language constructs of P4 are independent from architectures, but P4 supports architecture-specific `extern` objects and methods, and the switch's global behavior depends on the switch model. Section 2.5 shows how to write a formal architecture specification and how our operational semantics is linked to an architecture specification. An actual target may reject some programs during compilation due to resource constraints. Our operational semantics does not specify any of these constraints but what the program means when it can compile.

2.1 P4light and the front end

The formalization of P4 semantics is based on an abstract syntax tree (AST). But which AST should we use in order to define formal semantics and verify programs smoothly? According to the experience from CompCert Clight [3] and VST-Floyd [7] on the C language, it is nicer to elaborate, annotate, and transform the AST before defining the formal semantics and verifying programs. So we designed *P4light*, an AST of P4 such that

- each expression node is annotated with its type and implicit type casts are made explicit;

- each name is annotated with a locator (see below) to distinguish the same names in different scopes;
- side-effect expressions do not nest as subexpressions;[1]
- it is still close to P4 source code and each P4 light program is a legal P4 program.

We adapted the Petr4 front end (including the typechecker) to parse, elaborate, and typecheck P4 source program into P4light. Two additional transformation passes are applied to the AST after typechecking in order to make it easier to define the semantics and analyze the program. The first pass extracts side effects from subexpressions such that every every expression with side effects (e.g. function call) must appear directly on the right hand side of an assignment. For example, $\mathsf{a} = \mathsf{f(b)} + \mathsf{c}$ is transformed into $\mathsf{t1} = \mathsf{f(b)}; \mathsf{a} = \mathsf{t1} + \mathsf{c}$. So when analyzing the program, we can treat function calls as a kind of statement and do not have side effects when evaluating expressions.

The second pass adds *locators* to names in a P4 program. Names in a P4 program have multiple layers of scopes and the same identifier may not refer to the same thing. So a locator is added to each appearance of names to distinguish them. The syntax of locators is

$$\textsc{Locator} ::= \mathsf{glob}\ p \mid \mathsf{inst}\ p,$$

where p is a path, for names defined in the global scope and instance scopes (or say class scopes), respectively. Theoretically, one could use identifiers instead of paths in locators to do the same thing, but using paths makes it easier to track the location where each name is defined. With locators, there is no need for the semantics to maintain an environment mapping names to locations. (see Section 2.4)

[1] This point simplifies the development of operational semantics and program logic, and improves the interaction experience of our verifier. But P4 always evaluates expressions from left to right, not like C, whose evaluation order is unspecified. So this transformation does not add restrictions to the semantics.

2.2 Instantiation phase

P4 programs are often compiled and executed on programmable hardware (in other cases, compiled to C code, etc.). Such hardware is similar to FPGAs in the way that each computation unit is only used for one particular computation step once the program is loaded into the hardware. Hardware resources cannot be dynamically allocated during execution; programs cannot have loops, either. These constraints are reflected in the language design of P4. P4 has reusable modules (parser and control declarations), but the language only allows these modules to be *instantiated* with "compile-time known" arguments, so that the compiler can create a copy of the code for each usage and statically allocate hardware resources. For example, Figure 2.1 shows a sliding-window Bloom filter[2] implemented modularly in P4. Each sliding-window Bloom filter instance has 4 panes and each pane has 3 rows. Each row must have a persistent state that survives between packets, which is implemented as a "register" in the Tofino architecture[3] [23], the architecture used in this program. Each of these registers needs to be allocated to a unique location in the hardware. This is similar to hardware description languages, for example, modules in Verilog [34], which can be reused by statically allocating copies on hardware, too. But to my best knowledge, the instantiation phase has not been described in the literature about Verilog's formal semantics, nor in discussions of P4 semantics.

In order to have a clear and human-readable semantics and make it easy to analyze programs' behavior, we separate the semantics of P4 programs into two phases: the first is the *instantiation phase*, which simulates what the compiler should do; the second is the *execution phase*, which simulates the hardware processing of each

[2]In Chapter 5, we will explain what is a sliding-window Bloom filter, give the full version of Figure 2.1 and demonstrate how to verify this P4 program using Coq and our Verifiable P4. Here we only focus on its modular structure.

[3]Tofino is a series of high-performance P4-programmable switch chips developed by Intel. With its performance and flexibility, it is widely used in data centers.

```
package Switch(...); /* Prototype of the main package */

control Row(...) { /* A Bloom filter row */
  Register<...>(...) reg; ...
}

control Pane(...) { /* A simple Bloom filter */
  Row() row_1; Row() row_2; Row() row_3; ...
}

control SBFilter(...) { /* A sliding-window Bloom filter */
  /* Registers for timing */
  Register<...>(...) clear_index;
  Register<...>(...) timer;
  Pane() pane_1; Pane() pane_2; Pane() pane_3; Pane() pane_4; ...
}

Switch(SBFilter()) main; /* SBFilter is instantiated here */
```

Figure 2.1: Example of instantiation

packet. The main benefit of introducing this *phase distinction* is to separate two kinds of computation. The instantiation phase needs to handle higher-order objects, but they are fully determined by the P4 program and independent from the packets and the switch configuration from the control plane. The execution phase processes the packets without higher-order objects and object reallocation.

Similar design choices have also been discussed for "instantiation" in other programming languages, such as the ML language. ML compilers handle modules and functors either by closure passing [29] or by defunctorization [8] (i.e. instantiation at first during compilation). The former is similar to Petr4's approach and the latter is similar to ours. The semantics of ML is usually defined by closure passing. It seems to be because the first a few ML compilers used closure passing and because the instantiation phase seems to be more complicated for ML than P4. The advantage of using closure passing in compilers is that a functor in ML can be compiled just once

into machine code that serves for all of its instances. But program analysis is easier after instantiation in ML [8], which is the same as we observe for P4.

Although P4 compilers create a copy of code for each instance, it is not an ideal approach for the formal semantics, because that means we cannot analyze shared properties for all the instances of the same class. So, instead, we store the class name for each instance as the link to the corresponding code. Meanwhile, the formal semantics should be simple and independent from any hardware architecture, so the objects (i.e. instances) should be still addressed by names unrelated to hardware resource allocation. The control plane names described in the P4 Specification [12, Section 18.3] are perfect for distinguishing objects. The control plane names are fully qualified names (paths) allocated as follows.

1. The control plane names of an object instantiated at the top level are just its name. For example, the control plane name of Switch(SBFilter()) main is main.
2. An object instantiated by nameless instantiation (i.e. directly used as a constructor argument when instantiating another object) gets its control plane name by appending the name of the corresponding formal parameter after the control plane name of the object to which the parameter is passed, separated by a dot. For example, the control plane name of SBFilter() in Switch(SBFilter()) main is main.ig, as ig is the formal parameter's name.
3. An object instantiated inside a parser/control block gets its control plane name by appending its local name after the control plane name of the parser/control block, separated by a dot. For example, if the control plane name of the SBFilter instance is main.ig, then the control plane name of Pane() pane_1 in this instance is main.ig.pane_1.
4. P4 allows using @name annotation to overwrite the local names when generating control plane names as above. This feature is not yet supported in this work.

This allocation guarantees the control plane names are distinct, because the local names are distinct (including parameters and local definitions). These names are determined at compile time and can be used to describe how the switch or the control plane handles these objects (e.g. the control plane may modify registers and table entries). Also, this allocation forms a hierarchy in which all the objects instantiated in an instance (e.g. an SBFilter) have the same prefix, and the suffix of these objects are the same between different instances of the same parser/control block. This benefit will be discussed in more detail in Chapter 3.

Technically, the instantiation phase produces a global environment Γ that does not change during the execution phase. Γ consists of six parts:

- Γ_{func}, for function definitions (including all callable objects, e.g. parsers, control blocks and tables),
- Γ_{typ}, for type definitions,
- Γ_{senum}, for values of serializable enumeration types looked up by member names,
- Γ_{inst}, for class information of instances and references to other instances,
- Γ_{const}, for values of constants which may differ between instances, and,
- Γ_{ext}, for static information of extern objects.

Γ_{func}, Γ_{typ}, and Γ_{senum} are obtained from a single pass through the program, which is simple Γ_{inst} is the main product of the instantiation phase, while Γ_{const} and Γ_{ext} are byproducts during this procedure. Another byproduct is the initial state of extern objects before processing any packets, denoted by s_{init}. Γ_{inst} is a partial map $\textsc{Path} \rightharpoonup \textsc{Ident} \times \textsc{Path}$ that is used to look up object references. For local name bar in an object at path foo, $\Gamma_{\mathsf{inst}}(\mathsf{foo.bar})$ will be used. Entry $p \mapsto (n, q)$ in Γ_{inst} means the object referred by p belongs to the class named n and its actual location is q. Because q is already the actual location, Γ_{inst} always has $q \mapsto (n, q)$.

```
global Γ_inst, Γ_const, Γ_ext, s_init, decl_env := []

procedure instantiate(p, e, decl) :=
    inst_name := decl.name
    class_name := decl.class_name
    p_inst := p · inst_name
    e_0 := e
    for each param in decl.params
        p_param := p_inst · (param.name)
        if param is an instantiation
            instantiate(p_inst, e_0, declaration form of param)
            v := (param.class_name, p_param)
        else // param is not an instantiation
            // param may evaluate to either a value or an object reference
            v := evaluate(e_0, param)
        e := e[param.name ↦ v]
        if v is a value
            Γ_const := Γ_const[p_param ↦ v]
        else
            Γ_inst := Γ_inst[p_param ↦ v]
    body := decl_env[class_name]
    Γ_inst := Γ_inst[p_inst ↦ (class_name, p_inst)]
    if class_name is an extern object class
        (v_inv, v_init) := construct_extern(class_name, e)
        Γ_ext := Γ_ext[p_inst ↦ v_inv]
        s_init := s_init[p_inst ↦ v_init]
    else // class_name is a parser/control block
        for each decl′ in body
            if decl′ is an instantiation then
                instantiate(p_inst, e, decl′)
            update e, Γ_const, Γ_inst (similar to evaluating parameters)

procedure instantiate_prog(prog) :=
    e := []
    for each decl in prog
        if decl is a class then
            decl_env := decl_env(decl.name ↦ decl)
        else if decl is an instantiation then
            instantiate(ε, e, decl)
        e := e[decl.name ↦ (value of the instance)]
```

Figure 2.2: Pseudocode of instantiation

Figure 2.2 shows the pseudocode of the instantiation phase. The operator "·" (centered dot) is used for concatenating paths. All global variables are initialized as empty maps. decl_env stores declarations of classes. These classes must be declared at the top level, so there is no need to consider naming scope, and it is fine to use a global decl_env in the pseudocode. But in the Coq implementation, in order to rule out circular instantiation and guarantee that the instantiation phase terminates, declarations are stored into decl_env as closures: $\mathsf{decl\_env} := \mathsf{decl\_env}[\mathsf{decl.name} \mapsto (\mathsf{decl\_env}, \mathsf{decl})]$. The corresponding decl_env is used when instantiating the body of each declaration, so every recursive call of instantiate decreases on decl_env. The function instantiate takes a declaration decl that instantiates an object and the declaration appears in an object at path p. Local environment e is only used in the instantiation phase. The path for the newly instantiated object will be p_{inst}. The first loop evaluates each parameter of decl. If a parameter is an instantiation expression, it will be converted to a declaration named by the name of the corresponding *formal* parameter. Then this declaration is instantiated recursively under p_{inst}, and it becomes an object of class param.class_name at path p_{param}. If the parameter is not an instantiation, it will be evaluated while the names will be looked up in e_0. The result v will be stored in e and in Γ_{const} or Γ_{inst} depending on whether it is an value or a reference. After evaluating and instantiating the parameters, the body of decl will be instantiated. If it is an extern object, the architecture-specific function construct_extern is called to produce the static value (e.g. the size and width of a register) and the initial value. Otherwise, decl is a parser/control block. In this case, for each decl' that is an instantiation, call instantiate recursively.

The procedure instantiate_prog instantiates the whole program. It inserts class definitions into decl_env and calls instantiate with an empty path for each top level instantiation.

2.2.1 Function lookup

The instantiation phase makes the resolution of program constructs independent from runtime data. Besides the global environment Γ, we only need to know the path of the current control/parser instance, denoted as p. We will show the detailed resolution methods in Section 2.3. Here we show the most important part: function lookup. The judgment of function lookup is of the form

$$\Gamma, p \vdash exp \Downarrow_{\mathsf{lookup}} (p', p_{\mathsf{func}}),$$

where exp is the function expression, and (p', p_{func}) is the result. In the result, p' is either $\star$ or a path: If p' is $\star$, the function exp is executed in the current control/parser instance, such as calling a table or an action defined in the same control block; If p' is a path, the function exp is executed in the instance at p', such as calling a global action/function or another control/parser instance. The difference is that the callee can access class-scope variables of the caller if and only if $p' = \star$. The other part of the lookup result, p_{func}, is the path to look up the function body of exp in Γ_{func}.

Figure 2.3 presents the semantic rules for function lookup. The first two rules handle the case that the function expression is an identifier, and the last three rules handle the case that the function expression is a member expression (expression with a dot). We distinguish the origin of identifiers using locators, which annotate the identifiers in the form $n@loc$. E-LGlob is used for global actions and functions. E-LInst is used for local actions and parser states, which are modeled as functions. The premise $\Gamma_{\mathsf{inst}}(p) = (n_{\mathsf{class}}, p)$ indicates that the current instance p is an instance of class n_{class}. E-LTable is used for calling apply method of tables. We use $\mathsf{kind}(n_1@(\mathsf{inst}\ p'))$ to denote the determination of whether the expression is a table, using the type annotation generated by the front end (Section 2.1). Tables can access control block's local variables, so the first lookup result is $\star$. In the last two rules, E-LMemGlob

$$\frac{}{\Gamma, p \vdash n@(\mathsf{glob}\ p') \Downarrow_{\mathsf{lookup}} (\varepsilon, p')}\ \text{E-LGlob}$$

$$\frac{\Gamma_{\mathsf{inst}}(p) = (n_{\mathsf{class}}, p)}{\Gamma, p \vdash n@(\mathsf{inst}\ p') \Downarrow_{\mathsf{lookup}} (\star, n_{\mathsf{class}}.p')}\ \text{E-LInst}$$

$$\frac{\mathsf{kind}(n_1@(\mathsf{inst}\ p')) = \mathsf{table} \qquad \Gamma_{\mathsf{inst}}(p) = (n_{\mathsf{class}}, p)}{\Gamma, p \vdash n_1@(\mathsf{inst}\ p').n_2 \Downarrow_{\mathsf{lookup}} (\star, n_{\mathsf{class}}.p'.n_2)}\ \text{E-LTable}$$

$$\frac{\mathsf{kind}(n_1@(\mathsf{glob}\ p_2)) \neq \mathsf{table} \qquad \Gamma_{\mathsf{inst}}(p_2) = (n_{\mathsf{class}}, p_3)}{\Gamma, p_1 \vdash n_1@(\mathsf{glob}\ p_2).n_2 \Downarrow_{\mathsf{lookup}} (p_3, n_{\mathsf{class}}.n_2)}\ \text{E-LMemGlob}$$

$$\frac{\mathsf{kind}(n_1@(\mathsf{inst}\ p_2)) \neq \mathsf{table} \qquad \Gamma_{\mathsf{inst}}(p_1.p_2) = (n_{\mathsf{class}}, p_3)}{\Gamma, p_1 \vdash n_1@(\mathsf{inst}\ p_2).n_2 \Downarrow_{\mathsf{lookup}} (p_3, n_{\mathsf{class}}.n_2)}\ \text{E-LMemInst}$$

Figure 2.3: Semantics rules for function lookup

and E-LMemInst, the method name n_2 is prepended with the class name n_{class}, which determined by the locator annotated to n_1.

The function lookup is fully deterministic and implemented as a function in the mechanized semantics. This function is used in the same way in the program logic (Section 3.2), making it an important benefit of the instantiation phase.

2.2.2 Abstract methods

An important feature that has not been formalized previously (including Petr4's implementation) is abstract methods. Formalizing abstract methods is necessary for verifying stateful programs on Tofino. An abstract method is a short P4 program segment provided to an extern object to customize its behavior. The extern object's internal logic may call these abstract methods. The syntax of abstract methods is

```
Register<bit<32>, bit<16>>(32w65536, 0) reg;

RegisterAction<bit<32>, bit<16>, bit<32>>(reg) regact = {
    void apply(inout bit<32> value, out bit<32> rv) {
        rv = value;
        if (value == N − 1) {
            value = 0;
        }
        else {
            value = value + 1;
        }
    }
};
...
regact.execute(0);
```

Figure 2.4: An example of abstract method

similar to object-oriented languages. The extern object classes defined by the architecture may declare abstract methods and the abstract methods must be implemented for each instance during instantiation.

Abstract methods are necessary in P4 for at least one reason: on Tofino, each register is allocated in a pipeline stage, so it can be only accessed once per packet (unless recirculating, which is costly). So any read-then-write operation (such as incrementing a register cell) must be done in a single operation. The abstract method provides an interface that allows the programmer to specify how the value to write is computed from the original value. Figure 2.4 shows an example that uses abstract method to read and then update a register in the Tofino architecture. reg is a register and regact defines an action that modifies reg's value. apply is an abstract method, whose prototype is defined in the extern type RegisterAction and whose implementation for regact is as shown in Figure 2.4. Later, the P4 program may use, e.g., regact.execute(0) to update the 0-th cell's value. Inside the execute method, the abstract method apply will be invoked and the old value of the 0-th cell will be passed in as value while value after the function call will be written into the 0-th cell.

The P4 specification only allows abstract methods to "use the supplied arguments or refer to values that are in the top-level scope" [12]. Since it is independent from any local variable, the semantics of an abstract method can be represented by a relation of initial extern state, list of input arguments, final extern state, list of output arguments, and signal. This relation is determined during instantiation from the semantics of normal functions and statements and stored in Γ_{ext} as a constant data of the extern object.

The Tofino architecture allows more general forms of abstract methods. For example, abstract methods with @synchronous annotation may access local variables. This feature is nonstandard P4, in that it has not been accepted by the P4 Language Consortium. The formalization of more general abstract methods will require investigation in more applications.

2.3 Execution phase

The instantiation phase generates the static global environment Γ. Then the execution phase executes the program according to Γ. Because P4 programs do not have loops and recursions, it is not necessary to consider nonterminating programs. So the execution phase is defined as a big-step operational semantics. The basic expressions and statements are treated similarly as in Petr4. The most important difference is the treatment of uninitialized bits.

2.3.1 Uninitialized bits

Unlike C, where reading an uninitialized variable may cause undefined behavior, the P4 Specification states such read yields an *unspecified value*, including reading an uninitialized variable and reading a field of an invalid header [12, Section 8.23]. Also,

```
bit<8> x, y, z;
y = x;
z = x; /* z may differ from y */
```

reading twice may yield different values. The following program gives an example. So it is not enough to characterize P4's behavior by assigning an arbitrary value when a variable is uninitialized. Further, P4 supports bit access, e.g. x[5:2] = y, so each bit is virtually a field and has its own initialized-or-not status. Therefore in the operational semantics, we consider that each bit in storage can be either 0, 1, or uninitialized, represented by $0, 1, \perp$. A value with such three-valued bits is called a *storable value*. When a variable is declared without initialization, a storable value filled with $\perp$ is stored, unless the P4 Specification specifies something different in particular. For example, the validity bit of a uninitialized header is 0 (invalid), not $\perp$. On the other hand, the result of any expression evaluation, including single-variable expressions, is a normal value without uninitialized bits. So storable values are first converted to normal values nondeterministically when used in an expression, including being used as operands of arithmetic operations or the right-hand side of an assignment and being passed as arguments. When storing an evaluation result, it is converted to a storable value without uninitialized bits. For example,

```
bit<8> x, y, z, w;
y = x; /* The storable value is converted to normal value before writing into y */
z = y;
w = y; /* w and z must be the same */
```

2.3.2 Nondeterministic semantics

Because storable values need to be converted to normal values nondeterministically, P4's semantics is nondeterministic. Consider an abstract semantic judgment $a \Downarrow b$,

where a is the input and b is the output. For example, a may be a program statement and the initial state while b may be the final state of executing the statement. In a nondeterministic semantics, $a \Downarrow b$ means a possible result of executing a is b, and $a \Downarrow b_1, a \Downarrow b_2, a \Downarrow b_3$ may be all valid judgments. If $a \Downarrow b$ does not hold for any b, a cannot be executed at all, instead of undefined behavior. Correctness of compiling source program a into target program a' is that any behavior of a' is a valid behavior of a: $\forall b, a' \Downarrow b \implies a \Downarrow b$.[4] When proving program a satisfies some property P, the formalization is $\forall b.\ a \Downarrow b \implies P(b)$, which reads as every possible result b satisfies P.

2.3.3 Semantic rules

The syntax of values and l-values is as follows:

Value := BasicValue	(e.g. signed/unsigned integers and Booleans)
| List(Ident × Value)	(structs)
| Bool × List(Ident × Value)	(headers; Boolean is the validity bit)
| ···	(other values not used in this thesis)
Lvalue := Path	(local variable)
| Lvalue × Ident	(field of a struct, header, or union)
| Lvalue × Int × Int	(bit-slice)
| ···	(other l-values not used in this thesis)

Values include basic values, structs, and headers, but unlike previous work [18], we do not use closures. L-values are assignable variables or fields, including local variables, fields, and bit-slices. There are some more kinds of values and l-values, but they are

[4] The target program does not need to exhibit every possibility of the source program. It only needs to be within the possible execution of the source program.

not interesting enough to be covered in the thesis. The program state, usually denoted by s, consists of two parts: a stack frame for local variables within a control/parser block, and an extern state for extern objects.

$$\mathsf{StackFrame} := \mathsf{Path} \rightharpoonup \mathsf{Value}$$

$$\mathsf{ExternState} := \mathsf{Path} \rightharpoonup \mathsf{ExternObject}$$

$$\mathsf{State} := \mathsf{StackFrame} \times \mathsf{ExternState}$$

extern objects are as defined by the architecture. A table whose entries can be configured by the control plane is also considered as an extern object, because they are accessible from the outside of the P4 program.

Let Γ be the global static environment generated in Section 2.2. As P4 program statements are mostly inside classes (i.e. parsers and control blocks) and the instantiation phase does not duplicate the program for each instance, we need to know in which instance the current statement is executed in order to correctly interpret names. So we use a path p to indicate the path of the object that the program is currently in (p is an empty path if not in any object). Names are handled using p and locators (generated in Section 2.1). Local variables are always defined in classes, therefore a name referring to a local variable always has a locator of the form $\mathsf{inst}\ p'$: they are looked up in the stack frame using p'. A new empty stack frame will be used when calling a method in a different instance, since stack frames are not shared between different instances, and the old stack frame will be reused when returning from the call, so scope of local variables is resolved. A name referring to an instance may have a locator of the form $\mathsf{glob}\ p'$ or $\mathsf{inst}\ p'$. In the first case, it will be looked up in Γ_{inst} using p', and in the second case, it will be looked up in Γ_{inst} using $p \cdot p'$, because p' is the relative path from the current path p.

We adopt the following notations in the semantic rules. We use italic font for variables, such as *exp* for an expression and *stmt* for a statement, and use sans serif font for constants, such as normal and return for signals. We use $[v]_{\mathsf{sv}}$ to denote converting a normal value v to a storable value, and use $[v]_{\mathsf{v}}$ to denote nondeterministically converting a storable value to a normal value. Overline is used to indicate a list of items, such as $\overline{v}$.

The big-step semantic judgments are written as six auxiliary judgments

$$s \vdash lv \Downarrow_{\mathsf{read}} v \qquad \text{(l-value read, 6 rules)}$$

$$s \vdash lv := v \Downarrow_{\mathsf{write}} s' \qquad \text{(l-value write, 9 rules)}$$

$$\Gamma, p, s \vdash exp \Downarrow v \qquad \text{(expression, 18 rules)}$$

$$\Gamma, p, s \vdash exp \Downarrow (lv, sig) \qquad \text{(l-expression, 5 rules)}$$

$$\Gamma, p, s \vdash \overline{(d, exp)} \Downarrow \overline{(v, lv)} \qquad \text{(argument list, 5 rules)}$$

$$\Gamma, p \vdash exp \Downarrow_{\mathsf{lookup}} (p', p_{\mathsf{func}}) \qquad \text{(function lookup, 5 rules)}$$

and three main judgments

$$\Gamma, p, s \vdash stmt \Downarrow (s', sig) \qquad \text{(statement, 16 rules)}$$

$$\Gamma, p, s \vdash exp \Downarrow (s', sig) \qquad \text{(call-expression, 2 rules)}$$

$$\Gamma, p, s \vdash (f, \overline{v_{\mathsf{in}}}) \Downarrow (s', \overline{v_{\mathsf{out}}}, sig) \qquad \text{(function, 3 rules)}$$

For example, the judgment $\Gamma, p, s \vdash e \Downarrow v$ reads as "in global environment Γ, with object path p, in state s, the P4 expression e evaluates to value v." Judgment $\Gamma, p, s \vdash stmt \Downarrow (s', sig)$ reads as "for Γ and p, from state s, the execution of the P4 statement *stmt* results in state s' and signal *sig*." Signal is used to mark control flow in the conventional way to handle return and exit statements versus normal control flow. Call-expressions are treated as a separated judgment for convenience. They are

$$\frac{}{\Gamma, p, s \vdash x@(\mathsf{inst}\ p') \Downarrow (p', \mathsf{normal})}\ \text{E-LVar}$$

$$\frac{\Gamma, p, s \vdash exp \Downarrow (lv, sig)}{\Gamma, p, s \vdash exp.n \Downarrow ((lv, n), sig)}\ \text{E-LMember} \qquad \frac{s_{\mathsf{local}}(p) = v}{(s_{\mathsf{local}}, s_{\mathsf{extern}}) \vdash p \Downarrow_{\mathsf{read}} v}\ \text{E-RVar}$$

$$\frac{s \vdash lv \Downarrow_{\mathsf{read}} [\ldots, (n, v'), \ldots]}{s \vdash (lv, n) \Downarrow_{\mathsf{read}} v'}\ \text{E-RStruct}$$

$$\frac{s \vdash lv \Downarrow_{\mathsf{read}} (\mathsf{valid} := \_, [\ldots, (n, v'), \ldots])}{s \vdash (lv, n) \Downarrow_{\mathsf{read}} v'}\ \text{E-RHeader}$$

$$\frac{}{(s_{\mathsf{local}}, s_{\mathsf{extern}}) \vdash p := v \Downarrow_{\mathsf{write}} (s_{\mathsf{local}}[p \mapsto v], s_{\mathsf{extern}})}\ \text{E-WVar}$$

$$\frac{s \vdash lv \Downarrow_{\mathsf{read}} [\ldots, (n, \_), \ldots] \qquad s \vdash lv := [\ldots, (n, v), \ldots] \Downarrow_{\mathsf{write}} s'}{s \vdash (lv, n) := v \Downarrow_{\mathsf{write}} s'}\ \text{E-WStruct}$$

$$\frac{\begin{array}{c} s \vdash lv \Downarrow_{\mathsf{read}} (\mathsf{valid} := 1, [\ldots, (n, \_), \ldots]) \\ s \vdash lv := (\mathsf{valid} := 1, [\ldots, (n, v), \ldots]) \Downarrow_{\mathsf{write}} s' \end{array}}{s \vdash (lv, n) := v \Downarrow_{\mathsf{write}} s'}\ \text{E-WHeader1}$$

$$\frac{s \vdash lv \Downarrow_{\mathsf{read}} (\mathsf{valid} := b, [\ldots, (n, \_), \ldots]) \qquad b \in \{0, \bot\}}{s \vdash (lv, n) := v \Downarrow_{\mathsf{write}} s}\ \text{E-WHeader0}$$

Figure 2.5: L-value evaluation, read and write rules of P4light

used not only in statements with function calls but also in tables. Tables are treated as functions that first evaluate keys and match them with a table entry, followed by constructing and executing a call-expression from the matched table entry.

Figure 2.5 shows selected rules for evaluating, reading and writing l-values. Rules E-LVar and E-LMember evaluate l-expressions based on locators. Rules E-RVar and E-WVar are the base cases in which the l-value is just a path of a local variable, so it is directly read from or write to s_{local}. E-RStruct (E-RHeader, resp.) reads a field from a struct (header, resp.). E-WStruct writes to a field of a struct. E-

$$\frac{\Gamma, p, s \vdash exp \Downarrow v}{\Gamma, p, s \vdash (\mathsf{in}, exp) \Downarrow ([v]_{\mathsf{sv}}, \_)}\ \text{E-ArgIn} \qquad \frac{\Gamma, p, s \vdash exp \Downarrow (lv, \mathsf{normal})}{\Gamma, p, s \vdash (\mathsf{out}, exp) \Downarrow (\_, lv)}\ \text{E-ArgOut}$$

$$\frac{\Gamma, p, s \vdash exp \Downarrow (lv, \mathsf{normal}) \qquad s \vdash lv \Downarrow_{\mathsf{read}} v}{\Gamma, p, s \vdash (\mathsf{out}, exp) \Downarrow ([[v]_{\mathsf{v}}]_{\mathsf{sv}}, lv)}\ \text{E-ArgInOut}$$

Figure 2.6: Semantics rules for argument evaluation

WHeader1 and E-WHeader0 define the semantics of writing to a header field. The write operation only takes effect when the validity bit of the header is 1 (i.e. the header is valid), in accordance with the P4 Specification [12, Section 8.23].

It is worth noticing that the rule E-RHeader ignores the header's validity bit. If the validity bit is 1, it reads the field as expected. If the validity bit is 0 or $\perp$, it still works because we designed the semantics to maintain an invariant that all the bits of the header's fields are $\perp$ if the validity bit is 0 or $\perp$. So reading from an invalid header according to the rule E-RHeader will yield $\perp$s, precisely following the P4 Specification.

Figure 2.6 displays the rules for argument evaluation. Each argument is evaluated to a pair of a storable value and an l-value. Either part of the pair might be empty, denoted by "_". Rules are presented for a single argument, and its lifting to multiple arguments is conventional. Handling of nonnormal signals in l-value evaluation is omitted. The rule E-ArgIn evaluates an in parameter to the storable value corresponding to v, and the l-value part is empty. Similarly, the rule E-ArgOut evaluates an out parameter, and the result is only an l-value. The rule E-ArgInOut evaluates an inout parameter. The semantic rules for argument evaluation is essentially the same as Petr4's copy-in and copy-out rules, except handling nondeterministic bits. But we prefer calling them "argument evaluation", because copy-in and copy-out happen during argument passing, not argument evaluation. (see E-CallFunc rules and E-Internal)

$$\frac{\Gamma, p, s \vdash stmt_1 \Downarrow (s', \mathsf{normal}) \qquad \Gamma, p, s' \vdash stmt_2 \Downarrow (s'', sig)}{\Gamma, p, s \vdash stmt_1; stmt_2 \Downarrow (s'', sig)}\ \text{E-Seq}$$

$$\frac{\Gamma, p, s \vdash stmt_1 \Downarrow (s', sig) \qquad sig \neq \mathsf{normal}}{\Gamma, p, s \vdash stmt_1; stmt_2 \Downarrow (s', sig)}\ \text{E-Seq2}$$

$$\frac{\Gamma, p, s \vdash exp \Downarrow \mathsf{true} \qquad \Gamma, p, s \vdash stmt_1 \Downarrow (s', sig)}{\Gamma, p, s \vdash \mathsf{if}\ (exp)\ stmt_1\ \mathsf{else}\ stmt_2 \Downarrow (s', sig)}\ \text{E-IfT}$$

$$\frac{\Gamma, p, s \vdash exp \Downarrow \mathsf{false} \qquad \Gamma, p, s \vdash stmt_2 \Downarrow (s', sig)}{\Gamma, p, s \vdash \mathsf{if}\ (exp)\ stmt_1\ \mathsf{else}\ stmt_2 \Downarrow (s', sig)}\ \text{E-IfF}$$

$$\frac{\Gamma, p, s \vdash exp \Downarrow v}{\Gamma, p, s \vdash \mathsf{return}\ exp \Downarrow (s, \mathsf{return}\ v)}\ \text{E-Return}$$

$$\frac{\begin{array}{c}\mathsf{kind}(exp_2) = \mathsf{expr} \qquad \Gamma, p, s \vdash exp_1 \Downarrow (lv, \mathsf{normal}) \\ \Gamma, p, s \vdash exp_2 \Downarrow v \qquad s \vdash lv := [v]_{\mathsf{sv}} \Downarrow_{\mathsf{write}} s'\end{array}}{\Gamma, p, s \vdash exp_1 := exp_2 \Downarrow (s', \mathsf{normal})}\ \text{E-Assign}$$

$$\frac{\begin{array}{c}\Gamma, p, s \vdash exp_1 \Downarrow (lv, \mathsf{normal}) \\ \Gamma, p, s \vdash exp_2(\overline{exp_3}) \Downarrow (s', \mathsf{return}\ v) \qquad s' \vdash lv := [v]_{\mathsf{sv}} \Downarrow s''\end{array}}{\Gamma, p, s \vdash exp_1 := exp_2(\overline{exp_3}) \Downarrow (s'', \mathsf{normal})}\ \text{E-AssignCall}$$

Figure 2.7: Semantics rules for statements

The selected semantics rules for statements are shown in Figure 2.7. These rules are mostly standard. In the rule E-Seq, the execution continues to $stmt_2$ if the signal from $stmt_1$ is normal. In the rule E-Seq2, $stmt_2$ is skipped as $stmt_1$ has changed the normal control flow: e.g., the signal will be "return v" if $stmt_1$ executes a return statement. The rules E-IfT and E-IfF handle if-statements. The rule E-Return handles return statements. The rule E-Assign handles simple assignment statements without function calls. The rule E-AssignCall handles statements with function calls.

Figure 2.8 shows the rules for function call expressions. E-CallBuiltin handles built-in function calls, e.g. h.setValid(), where h is a header. Normal function calls are

$$\frac{\begin{array}{c}\mathsf{kind}(exp_1.n) = \mathsf{builtin} \\ \Gamma, p, s \vdash exp_1 \Downarrow (lv, \mathsf{normal}) \qquad \overline{d} = \mathsf{dirs}(exp_1.n) \\ \Gamma, p, s \vdash \overline{(d, exp_2)} \Downarrow \overline{(v, lv')} \qquad \Gamma, p, s \vdash (lv, n, \overline{v}) \Downarrow_{\mathsf{builtin}} (s', sig)\end{array}}{\Gamma, p, s \vdash exp_1.n(\overline{exp_2}) \Downarrow (s', sig)} \text{ E-CallBuiltin}$$

$$\frac{\begin{array}{c}\mathsf{kind}(exp_1) = \mathsf{func} \qquad \Gamma, p \vdash exp_1 \Downarrow_{\mathsf{lookup}} (\star, p_{\mathsf{func}}) \\ f = \Gamma_{\mathsf{func}}(p_{\mathsf{func}}) \qquad \overline{d} = \mathsf{dirs}(exp_1) \qquad \Gamma, p, s \vdash \overline{(d, exp_2)} \Downarrow \overline{(v, lv)} \\ \Gamma, p, s \vdash (f, \overline{v}) \Downarrow (s', \overline{v'}, sig) \qquad s' \vdash \overline{lv := v'} \Downarrow_{\mathsf{write}} s''\end{array}}{\Gamma, p, s \vdash exp_1(\overline{exp_2}) \Downarrow (s'', sig)} \text{ E-CallFunc1}$$

$$\frac{\begin{array}{c}\mathsf{kind}(exp_1) = \mathsf{func} \qquad \Gamma, p \vdash exp_1 \Downarrow_{\mathsf{lookup}} (p', p_{\mathsf{func}}) \\ f = \Gamma_{\mathsf{func}}(p_{\mathsf{func}}) \qquad \overline{d} = \mathsf{dirs}(exp_1) \qquad \Gamma, p, s \vdash \overline{(d, exp_2)} \Downarrow \overline{(v, lv)} \\ \Gamma, p', ([], s_{\mathsf{extern}}) \vdash (f, \overline{v}) \Downarrow ((\_, s'_{\mathsf{extern}}), \overline{v'}, sig) \\ (s_{\mathsf{local}}, s'_{\mathsf{extern}}) \vdash \overline{lv := v'} \Downarrow_{\mathsf{write}} s''\end{array}}{\Gamma, p, (s_{\mathsf{local}}, s_{\mathsf{extern}}) \vdash exp_1(\overline{exp_2}) \Downarrow (s'', sig)} \text{ E-CallFunc2}$$

Figure 2.8: Semantics rules for function call expressions

handled by E-CallFunc1 and E-CallFunc2. These two rules are applied based on the function lookup result in Section 2.2.1, which are $\Gamma, p \vdash exp_1 \Downarrow_{\mathsf{lookup}} (\star, p_{\mathsf{func}})$ and $\Gamma, p \vdash exp_1 \Downarrow_{\mathsf{lookup}} (p', p_{\mathsf{func}})$, respectively. In E-CallFunc1, the function body is executed in the same path p and the same state s. In E-CallFunc2, the function body is executed in path p' and local variables are cleared in the state, and the original local variables, stored in s_{local}, are unchanged until copying the out parameters as $(s_{\mathsf{local}}, s'_{\mathsf{extern}}) \vdash \overline{lv := v'} \Downarrow_{\mathsf{write}} s''$. In both E-CallFunc1 and E-CallFunc2, direction information of the arguments, $\overline{d}$, is extracted from the type annotation of the function expression exp_1. We assume the argument evaluation result, $\overline{(v, lv)}$, is automatically split into two lists, $\overline{v}$ and $\overline{lv}$, such that $\overline{v}$ contains in and inout parameters, and $\overline{lv}$ contains out and inout parameters.

Figure 2.9 shows the rules for functions. E-Internal is for normal P4 functions. At the beginning, the in parameters are "copied in" using l-value write. Because the parameter names are annotated with locators, there no concern of scope. At the end,

$$\frac{\begin{array}{c} s \vdash \overline{p_{\mathsf{in}} := v_{\mathsf{in}}} \Downarrow_{\mathsf{write}} s' \\ \Gamma, p, s' \vdash \mathsf{stmt} \Downarrow (s'', \mathsf{return}\ v) \qquad s'' \vdash \overline{p_{\mathsf{out}}} \Downarrow_{\mathsf{read}} \overline{v_{\mathsf{out}}} \end{array}}{\Gamma, p, s \vdash (\mathsf{internal}\ (p_{\mathsf{in}}, p_{\mathsf{out}}, \mathsf{stmt}), v_{\mathsf{in}}) \Downarrow (s', v_{\mathsf{out}}, \mathsf{return}\ v)}\ \text{E-INTERNAL}$$

$$\frac{\begin{array}{c} \Gamma, p, s \vdash \overline{key} \Downarrow \overline{v} \qquad \mathcal{C}(p \cdot n) = \overline{entry} \qquad (\overline{v}, \overline{kind}, \overline{entry}) \Downarrow_{\mathsf{match}} n'(\overline{exp_2}) \\ \overline{action} = [\ldots, n'(\overline{exp_1}), \ldots] \qquad \Gamma, p, s \vdash n'(\overline{exp_1}, \overline{exp_2}) \Downarrow (s', \mathsf{return\ null}) \end{array}}{\Gamma, p, s \vdash (\mathsf{table}\ (n, \overline{key}, \overline{kind}, \overline{action}), []) \Downarrow (s', [], \mathsf{null})}\ \text{E-TABLE}$$

Figure 2.9: Semantics rules for functions

the out parameters are "copied out" as l-value read. E-TABLE is the more interesting rule, which is for tables. The function body of a table essentially has four fields:

- n, the local name of the table;
- $\overline{key}$, the keys, which is a list of expressions;
- $\overline{kind}$, a list of match kinds of the same length as $\overline{key}$;
- $\overline{action}$, a list actions pending to be called based on table matching result.

Table evaluation first evaluates $\overline{key}$ into values, $\overline{v}$. The condition $\mathcal{C}(p \cdot n) = \overline{entry}$ means retrieving table entries from control plane information, denoted by $\mathcal{C}$, using the table's fully qualified name $p \cdot n$. Then, $(\overline{v}, \overline{kind}, \overline{entry}) \Downarrow_{\mathsf{match}} n'(\overline{exp_2})$ indicates the procedure of finding the matching entry $n'(\overline{exp_2})$ of $\overline{v}$ in $\overline{entry}$. This procedure is primarily determined by the architecture, because the architecture may define custom match kinds. Then, the rule finds the corresponding action $n'(\overline{exp_1})$ in $\overline{action}$. The arguments $\overline{exp_1}$ and $\overline{exp_2}$ are combined together in order to execute the action.

In this presentation of the E-TABLE rule, certain technical details have been omitted. First, a table has its return value that enables the retrieval of the matched action after the table.apply(). The expression of E-TABLE ignores this complexity and returns null. Second, instead of having entries determined by the control plane, there is another kind of table whose entries are hardcoded in the P4 program. These tables do not need the step $\mathcal{C}(p \cdot n) = \overline{entry}$.

2.4 Comparison with Petr4

This section compares our semantics with Petr4's [18] semantics and explains our design choices. Petr4 was the first formal semantics for P4. It gave two representations of the P4 semantics: one is called Core P4, for a subset of P4 with formal semantics and pen-and-paper soundness proof; the other is the interpreter, which is an OCaml program and supports almost all P4 features such as parsers and extern functions and objects.

Figure 2.10 shows the semantic rules handling objects in Petr4's Core P4 semantics. It is adapted for better presentation: first, it is simplified by removing support for the features that are not related to the comparison; second, Petr4's Core P4 semantics does not include extern objects, so we extracted the semantic rules for extern objects from Petr4's interpreter and added to Figure 2.10.

Petr4's Core P4 semantics uses closures intensively to handle function-like objects with local scopes in P4, so it is in a more standard way of defining semantics. But since P4's scopes are statically allocated, we find that defining the semantics based on the phase distinction is more convenient for program reasoning, as we will demonstrate in Example 2.1.

In Petr4 semantics, statements and declarations are treated similarly without phase distinction, so here we consider both kinds as statements. The form of semantic judgment is

$$\langle C, \Delta, \sigma, \epsilon, stmt\rangle \Downarrow \langle \Delta', \sigma', \epsilon', sig\rangle,$$

where σ is a global storage that maps locations to values, ϵ is a local environment that maps currently visible names to locations. Locations are allocated dynamically. The judgment ℓ fresh is used to obtain a previously not used location ℓ.

The states of extern objects are persistent between packets, unless the switch specification resets them outside of the P4 program. So in order to make sure extern

E-CtrlDecl

$$\frac{\ell \text{ fresh} \qquad val = \mathsf{cclos}(\epsilon, \mathsf{ctrl}(\overline{d\ x : \tau})(\overline{x_c : \tau_c})\ \{\overline{decl}\ \ stmt\})}{\langle C, \Delta, \sigma, \epsilon, \mathsf{ctrl}\ X(\overline{d\ x : \tau})(\overline{x_c : \tau_c})\ \{\overline{decl}\ \ stmt\}\rangle \Downarrow \langle \Delta, \sigma[\ell \mapsto val], \epsilon[X \mapsto \ell], \mathsf{normal}\rangle}$$

E-CtrlInst

$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, X\rangle \Downarrow \langle \sigma_1, \mathsf{cclos}(\epsilon_{cc}, \mathsf{ctrl}(\overline{d\ x : \tau})(\overline{x_c : \tau_c})\ \{\overline{decl}\ \ stmt\})\rangle \\ \epsilon(\mathsf{path}) = p \qquad \langle C, \Delta, \sigma_1, \epsilon, \overline{exp}\rangle \Downarrow \langle \sigma_2, \overline{val_c}\rangle \\ \overline{\ell_c}, \ell \text{ fresh} \qquad val = \mathsf{clos}(\epsilon_{cc}[\mathsf{path} \mapsto p.x][\overline{x_c \mapsto \ell_c}], \overline{d\ x : \tau}, \{\}, \{\overline{decl}\ \ stmt\})\end{array}}{\langle C, \Delta, \sigma, \epsilon, X(\overline{exp})\ x\rangle \Downarrow \langle \Delta, \sigma_2[\overline{\ell_c \mapsto val_c}][\ell \mapsto val], \epsilon[x \mapsto \ell], \mathsf{normal}\rangle}$$

E-ExtnInst1

$$\frac{\begin{array}{c}\langle C, \Delta, \sigma, \epsilon, X\rangle \Downarrow \langle \sigma_1, \mathsf{extern}\ Y\rangle \qquad \epsilon(\mathsf{path}) = p \\ (p.x \mapsto \_) \notin \sigma_1 \qquad \langle C, \Delta, \sigma_1, \epsilon, \overline{exp}\rangle \Downarrow \langle \sigma_2, \overline{val_c}\rangle \qquad \langle C, Y, \overline{val_c}\rangle \Downarrow_{\mathsf{extern}} val\end{array}}{\langle C, \Delta, \sigma, \epsilon, X(\overline{exp})\ x\rangle \Downarrow \langle \Delta, \sigma_2[p.x \mapsto val], \epsilon[x \mapsto p.x], \mathsf{normal}\rangle}$$

E-ExtnInst2

$$\frac{\langle C, \Delta, \sigma, \epsilon, X\rangle \Downarrow \langle \sigma_1, \mathsf{extern}\ Y\rangle \qquad \epsilon(\mathsf{path}) = p \qquad (p.x \mapsto \_) \in \sigma_1}{\langle C, \Delta, \sigma, \epsilon, X(\overline{exp})\ x\rangle \Downarrow \langle \Delta, \sigma_1, \epsilon[x \mapsto p.x], \mathsf{normal}\rangle}$$

Figure 2.10: Petr4's Core P4 semantic rules for objects (simplified for demonstration)

objects are always allocated to the same locations, their control plane names are used as their locations, which are computed rather than arbitrarily allocated. These names, if computed correctly, are exactly the paths that we use in our semantics (see Section 2.2). Together with internal locations allocated during execution, they make up locations in this version of Petr4 semantics. In order to maintain these names without introducing a new context in the semantic judgment, a special field called path is added to the environment ϵ (as shown in Figure 2.10).

Rule E-CtrlDecl means, when evaluating a control declaration, the semantics allocates a fresh location ℓ and stores the declaration into ℓ as a closure with the current local environment ϵ. Rule E-CtrlInst evaluates a control instantiation. It first retrieves the closure from the environment and the storage, followed by find-

ing p, which is the *path* of the current environment, and evaluating the constructor arguments $\overline{exp}$.

Fresh locations $\overline{\ell_c}$ are allocated for the constructor arguments, and fresh location ℓ is for the control instance. Then the closure *val* representing the control instance is created by binding the parameter names to locations $\overline{\ell_c}$ and path to $p.x$, which is the path of the control instance, respectively. Finally, the constructor arguments are stored in fresh locations $\overline{\ell_c}$ and *val* is stored in ℓ. In our semantics, these two rules are handled by the instantiation phase without dynamic allocation or closure passing.

For extern object instantiation, Petr4 semantics needs to consider whether the object has already been initialized. If it has not been initialized, the rule ExtnInst1 is applied. In ExtnInst1, $\langle C, \Delta, \sigma, \epsilon, X\rangle \Downarrow \langle \sigma_1, \mathsf{extern}\ Y\rangle$ means evaluating the name X gives an extern object class Y. It finds the path p of the current environment and tests $p.x$ is uninitialized. Then $\langle C, Y, \overline{val_c}\rangle \Downarrow_{\mathsf{extern}} val$ means according to the architecture, initializing an extern object of class Y with parameters $\overline{val_c}$ gives *val*. In this rule, the location for the new instance is given by $p.x$ instead of allocating a fresh one. Otherwise, the rule ExtnInst2 will be applied. In ExtnInst2, $p.x$ is already in σ so already initialized, therefore the value from previous packet processing should be used without reinitialization. In our semantics, extern objects are allocated in the instantiation phase, so there is no need to consider whether an object is initialized or not.

Another comparison between Petr4 and our semantics is on location manipulation, as shown in Figure 2.11. The two E-VarAssign rules handle exactly the same assignment statement whose left hand formula is just a variable.[5] For assignment $x := exp$, Petr4 semantics needs to first looks up in the environment ϵ to find ℓ before writing into it. In contrast, our semantics is facilitated with the locator, and

[5]They are special cases of the complete assignment rules. For example, E-VarAssign in this thesis is derived from E-Assign, E-LVar, and E-WVar. These simple rules are more accessible for comparison.

E-VarAssign (Petr4)

$$\frac{\epsilon(x) = \ell \qquad \langle C, \Delta, \sigma, \epsilon, exp\rangle \Downarrow \langle val\rangle}{\langle C, \Delta, \sigma, \epsilon, x := exp\rangle \Downarrow \langle \Delta, \sigma[\ell \mapsto val], \epsilon, \mathsf{normal}\rangle}$$

E-VarAssign (This thesis)

$$\frac{\Gamma, p, (s_{\mathsf{local}}, s_{\mathsf{extern}}) \vdash exp \Downarrow val}{\Gamma, p, (s_{\mathsf{local}}, s_{\mathsf{extern}}) \vdash x@(\mathsf{inst}\ p') := exp \Downarrow ((s_{\mathsf{local}}[p' \mapsto val], s_{\mathsf{extern}}), \mathsf{normal})}$$

Figure 2.11: Comparison with Petr4 on location manipulation

the assignment statement becomes $x@(\mathsf{inst}\ p') := exp$.[6] So the result is directly written in p'. The same look-up procedure also occurs when evaluating variables in the expression exp. This significantly reduces the work of designing and proving the program logic. The following example illustrates how our design facilitates reasoning about program execution.

Example 2.1. Figure 2.12 shows an example P4 program written in a simplified version of the V1Model architecture, and whose ingress block gets input a and outputs x.

In our semantics, the program in Example 2.1 first goes through the instantiation phase. We get the objects

$$\mathsf{main.ig} \mapsto (\mathsf{ctrl\ MyIngress}, \mathsf{main.ig})$$
$$\mathsf{main.ig.c1} \mapsto (\mathsf{ctrl\ C}, \mathsf{main.ig.c1})$$
$$\mathsf{main.ig.c2} \mapsto (\mathsf{ctrl\ C}, \mathsf{main.ig.c2})$$
$$\mathsf{main.ig.c1.reg} \mapsto (\mathsf{register} : [\mathsf{8w0}])$$
$$\mathsf{main.ig.c2.reg} \mapsto (\mathsf{register} : [\mathsf{8w0}])$$

Then the switch starts processing packets. In this procedure, the object structure does not change—only the values in registers may change and be kept between packets.

[6] Variables can only be local, so their locators must be $\mathsf{inst}\ p$.

```
control C(out bit<8> x) {
    register<bit<8>>(1) reg;
    apply {
        bit<8> y;
        reg.read(y, 0);
        y = y + 1;
        reg.write(0, y);
        x = y;
    }
}

control MyIngress(in bit<8> a, out bit<8> x) {
    C() c1;
    C() c2;
    apply {
        if(a == 0) {
            c1.apply(x);
        }
        c2.apply(x);
    }
}

Main(MyIngress()) main;
```

Figure 2.12: P4 program in Example 2.1

In contrast, Petr4's semantics uses a special field cnt with initial value 0 in σ to count the location to allocate. It first loads control definitions into the storage

$$\epsilon = [\mathsf{C} \mapsto 0; \mathsf{MyIngress} \mapsto 1]$$

$$\sigma = [\mathsf{cnt} \mapsto 2; 0 \mapsto (\text{body of } \mathsf{C}); 1 \mapsto (\text{body of } \mathsf{MyIngress})].$$

Petr4's semantics does not support nameless instantiation MyIngress(), but we can follow the implementation of the interpreter to see that, after the instantiation of

main, the program environment and storage will be

$$\epsilon = [\mathsf{C} \mapsto 0; \mathsf{MyIngress} \mapsto 1; \mathsf{main} \mapsto 3]$$
$$\epsilon_{\mathsf{ig}} = [\mathsf{C} \mapsto 0; \mathsf{path} \mapsto \mathsf{main.ig}]$$
$$\sigma = [\mathsf{cnt} \mapsto 4; 0, 1 \mapsto \ldots; 2 \mapsto (\epsilon_{\mathsf{ig}}, \text{body of } \mathsf{MyIngress}); 3 \mapsto \mathsf{package}[\mathsf{ig} \mapsto 2]].$$

Then the MyIngress control block is executed with actual packet data. It knows that the closure stored at location 2 should be executed, by looking up $\epsilon(\mathsf{main}) = 3$, $\sigma(3)(\mathsf{ig}) = 2$. As it is inside MyIngress, ϵ_{ig} is used. It starts with instantiating two instances of C, namely c1 and c2. The rule E-CtrlInst is used and there is

$$\epsilon_{\mathsf{ig}} = [\mathsf{C} \mapsto 0; \mathsf{path} \mapsto \mathsf{main.ig}; \mathsf{c1} \mapsto 4; \mathsf{c2} \mapsto 5]$$
$$\epsilon_{\mathsf{c1}} = [\mathsf{path} \mapsto \mathsf{main.ig.c1}]$$
$$\epsilon_{\mathsf{c2}} = [\mathsf{path} \mapsto \mathsf{main.ig.c2}]$$
$$\sigma = [\mathsf{cnt} \mapsto 6; 0, 1, 2, 3 \mapsto \ldots; 4 \mapsto (\epsilon_{\mathsf{c1}}, \text{body of } \mathsf{C}); 5 \mapsto (\epsilon_{\mathsf{c2}}, \text{body of } \mathsf{C})].$$

Then the program's behavior depends on the value of a. If $\mathsf{a} = 0$, then c1 is called and a fresh location is allocated for x in C. (x is not copied in because it is an out parameter.

$$\epsilon_{\mathsf{c1}} = [\mathsf{path} \mapsto \mathsf{main.ig.c1}; \mathsf{x} \mapsto 6]$$
$$\sigma = [\mathsf{cnt} \mapsto 7; 0, 1, 2, 3, 4, 5 \mapsto \ldots; 6 \mapsto \_].$$

Then reg let instantiated and initialized from rule E-ExtnInstInit.

$$\epsilon_{\mathsf{c1}} = [\mathsf{path} \mapsto \mathsf{main.ig.c1}; \mathsf{x} \mapsto 6; \mathsf{reg} \mapsto \mathsf{main.ig.c1.reg}]$$
$$\sigma = [\mathsf{cnt} \mapsto 7; 0, 1, 2, 3, 4, 5, 6 \mapsto \ldots; \mathsf{main.ig.c1.reg} \mapsto [0]].$$

Then, the rest of C is executed:

$$\epsilon_{c1} = [\mathsf{path} \mapsto \mathsf{main.ig.c1}; \mathsf{x} \mapsto 6; \mathsf{reg} \mapsto \mathsf{main.ig.c1.reg}; \mathsf{y} \mapsto 7]$$
$$\sigma = [\mathsf{cnt} \mapsto 8; 0, 1, 2, 3, 4, 5 \mapsto \ldots; 6 \mapsto 1; 7 \mapsto 1; \mathsf{main.ig.c1.reg} \mapsto [1]].$$

Then x will be copied out and c2 will be executed. In c2, x and y will be allocated at 8 and 9 respectively. But this allocation depends on the actual data. The case discussed above is a equals 0. If a is not equal to 0, c1 will not be executed and c2 will start with $\mathsf{cnt} \mapsto 6$. x and y will be allocated at 6 and 7 instead.

From this example, we can see that our semantics is much easier to manipulate. This advantage will be more apparent in the program logic (Chapter 3) and the verification (Chapter 5). It is also important for proving correctness of a P4 compiler, which is beyond the scope of this thesis. First, instantiation and execution (pure data processing) are mixed and executed intermittently in Petr4's semantics. In the design of the P4 language, the restrictions are designed to force instantiation to be done statically and the cost of implementing the program on actual hardware is determined only by instantiation. So it is better to point out this idea in its formal semantics.

The second benefit is that locations are statically determined and easy to be separated in our semantics. Petr4's semantics allocates fresh locations during execution. In practice, a counter is used to track the next location to allocate. However, that means we need to reason about this counter, even as the behavior of the program is totally irrelevant to the counter. And the locations are, in other words, dynamically allocated pointers, so we need to consider that the locations are nonoverlapping and constants are never modified when defining the semantics of the program logic judgments and proving soundness of the logic.

Program analyzers also need to be careful to avoid name capturing. The usage of locators in our semantics simplifies the variable disambiguation procedure. Context

does not change within a control/parser object, so it does not need to be maintained in the type system, soundness proof, and the analyzer.

Function call interface Finally, there is another difference between Petr4 and our semantics that is noteworthy but perpendicular with the instantiation-versus-closure discussion. Function parameters are passed as a list of values in our semantics rather than a named mapping as in Petr4, because the names of formal parameters are internal for the callee, so the caller should not refer to these names for copying-in and copying-out. In particular, control/parser instances may be parameterized with other control/parser instances as constructor parameters. These host instances can call these parameter instances or pass them to other instances as constructor parameters. In either case, only the type signatures of the callees are relevant, not the names. In modular reasoning, we want to reason about the caller with only the behavior of the callee but not the implementation, so we want to remove the names of formal parameters in the semantics of function call.

2.5 Architecture specification

Architecture-specific components are not part of P4's core semantics, but they are very important in order to formalize programs' behavior. The formal specification of a P4 architecture consists of three parts: the extern objects and methods, table match kinds, and the switch model. The P4 semantics is parameterized by such an architecture specification.

The specification of extern objects and methods consists of an initialization function for each class of objects and an execution relation for each method. The initialization function is used in the instantiation phase. Its input is a 5-tuple

$(\Gamma_{\mathsf{ext}}, e_{\mathsf{extern}}, \overline{type}, p, \overline{(p_{\mathsf{param}} \mid v_{\mathsf{param}})})$, where Γ_{ext} and e_{extern} are the static environment and the dynamic state for extern objects in the ongoing instantiation phase, respectively. The list of type parameters is denoted as $\overline{type}$. The path p indicates where the extern object is instantiated. The last element, $\overline{(p_{\mathsf{param}} \mid v_{\mathsf{param}})}$, is a mixed list of paths and values, which is the constructor parameters The paths in the list represent parameters that are objects. The output of the initialization function is a pair $(\Gamma'_{\mathsf{ext}}, e'_{\mathsf{extern}})$.

The execution relation is used in the execution phase. It is a relation on an 8-tuple

$$(\Gamma_{\mathsf{ext}}, e_{\mathsf{extern}}, p, \overline{type}, \overline{v_{\mathsf{param}}}, e'_{\mathsf{extern}}, \overline{v'_{\mathsf{param}}}, sig),$$

where Γ_{ext} is the static extern environment, p is the path of the extern object, e_{extern} is extern state before the call, $\overline{type}$ and $\overline{v_{\mathsf{param}}}$ are type and in parameters, e'_{extern} is the extern state after the call, $\overline{v'_{\mathsf{param}}}$ is out parameters, sig is the signal.

An architecture may define custom match kinds, so their interpretation needs to be specified by the architecture. Technically, it defines a function that takes a list of values of keys with their match kinds and a list of table entries and returns the matched table entry.

The switch model is the behavior of the whole switch. P4 programmable hardware consists of P4 programmable components and fixed-function components, and a P4 program defines the programmable components. But still it is necessary to define other components and the connection between them. So in an architecture specification, there is a inductive relation that defines how the architecture execute a packet. This relation invokes the P4 semantics for the P4 programmable components. Currently, it only processes one packet at a time without queueing and concurrency.

2.6 Implementation

This section briefly describes the implementation of the semantics in this chapter.

P4light and front end The P4light syntax and the front end consist of four parts: P4light syntax, parser (including lexer) and typechecker (including elaborator), two transformation passes (as described in Section 2.1), and pretty printer. The P4light syntax is written in Coq and extracted to OCaml to connect with OCaml code. The parser and typechecker are written in OCaml. The two transformation passes are written in Coq and extracted to OCaml, so it is possible to prove their correctness in the future. The pretty printer is an OCaml program that prints the P4light AST as Coq source code, in order to handle P4 programs in Coq.

Instantiation The instantiation phase is implemented as a computable function in Coq. One may use Coq's built-in Compute mechanism to generate the global environment.

Execution Normal values and storable values are implemented as a data type parameterized by different bit representations (bool and option bool, respectively), This makes some proofs applicable for both representations. The execution phase judgments are defined as (mutually) inductive relations.

Artifact The artifact from this chapter is distributed via a GitHub repository [43].

Chapter 3

Program Logic

Program logics, especially Hoare logic and separation logic, are widely used for reasoning about imperative programs in languages such as C and Java.[1] In this chapter, we introduce our "Verifiable P4" program logic for semi-modular verification of P4 control blocks. The basis of Verifiable P4 logic is Hoare logic with function calls [25]. It uses modification sets, as in Dafny [26] and ACSL [14], to track heap-like shared state between functions. We propose hierarchical extern predicates that provide a better encapsulation of state representation and modification sets in P4 control blocks, utilizing the hierarchical structure of the state as designed in Chapter 2. The syntax and semantics of function specifications and Hoare triples are based on hierarchical extern predicates, and then we derive program logic rules from the operational semantics. All the program logic rules are proven sound with respect to the operational semantics in the Coq proof assistant.

Verifiable P4 logic requires each function to have a function specification in the same way as many other program logics. Because P4 does not support recursion, the functions can be proved in the order from the lowest level to the highest level. So an

[1] Program logics are not only the standard method for proving functional correctness of imperative programs, but also the logical basis to justify static analysis algorithms.

PATH p, MOD m M
PRE (ARG $\vec{P}$, MEM $\vec{Q}$, EXT $\vec{R}$)
POST (RET v, ARG $\vec{P'}$, MEM $\vec{Q'}$, EXT $\vec{R'}$)

Figure 3.1: Simple form of function specification

environment of function specifications is not necessary in the program logic judgments. and step-indexing [2] is not needed in the semantics of function specifications.

A simple form of function specification is shown in Figure 3.1. We postpone detailed explanation until Section 3.2, and only provide an overview to motivate hierarchical predicate. In Figure 3.1, p is the path where the function is executed (the fully qualified name of the object), which is the same as in the semantics. MOD m M indicates the sets of local variables and extern objects modified in the function. Describing modified local variables is necessary, because P4 programs have local variables visible in different functions (actions) in the same class (e.g. a control block, see the example in Section 3.2). The rest of the function specification is a precondition and a postcondition. RET v describes the return value. The three remaining parts ARG, MEM, and EXT are descriptions of in and out arguments, local variables, and extern objects, respectively.

The extern objects are crucial for stateful applications, and their state is shared around the whole program. In a function specification, the state of extern objects is described by EXT $\vec{R}$, where $\vec{R}$ is a list of *hierarchical predicates* for extern objects as defined in Section 3.1. This design allows us to meet two requirements at the same time: (1) to avoid the difficulty of automating proofs in separation logic, and use modification set instead to manipulate the predicates, and (2) to still encapsulate the representation and modification of extern objects inside a control block, which makes them invisible from outside in the specification, just as they are invisible from outside in the program. In Section 3.2, we describe the form of function specifications and program logic rules of our P4 program logic in detail.

Verifiable P4 program logic does not handle packet parsers in P4. Parsers have a different control flow and they are stateless, so it is presumably more suitable for a different approach of verification. Leapfrog [19] is an automatic verifier for P4 parser equivalence, and it may be extended to more general automatic verification and verification with soundness proof in the future. So we only focus on control blocks in the program logic for P4.

3.1 Hierarchical extern predicates

Modularity is just as important in verification as in programming. It saves effort and enables reusable components. To achieve modularity, we follow the approach used by most modular verification frameworks, which is using function specifications, each of which consists of a precondition and a postcondition, to describe a function, in particular, a control block. As the extern objects in the control block are persistent, their states need to be maintained in function specifications. However, these extern objects are invisible from outside of the control block, so we want to encapsulate which extern objects are used and how they represent the state of the control instance in a predicate, so that the client of the control block (the code that calls the control block) can use it modularly (decoupled from its implementation details) and conveniently (without mentioning each extern object).

Suppose we have a predicate P without its definition and P holds before executing a program block c. After executing c, whether P still holds or not is unclear, because the objects that P depends on may be modified by c. One traditional solution is separation logic, but the complexity and expressive power of separation logic are not needed for P4, a statically instantiated language without pointers. Another traditional solution is using modification sets and dependent sets. That is, we characterize c with its *modification set*, denoted with M, which is the set of paths of the extern

objects that c may modify, and characterize P with its *dependent set*, denoted with D, namely the set of paths of the extern objects that P may depend on. If M and D are disjoint, then P still holds after executing c. But, in this approach, D exposes the names of objects used internally and breaks modularity.

To address these issues, we introduce *hierarchical predicates* for extern objects based on P4's hierarchical state structure and paths. We also refer to them as extern predicates in certain contexts. For paths p and q, let $p \sqsupseteq q$ denote that p is a prefix[2] of q.[3] According to the allocation of control plane names, if the path of a control instance is p, then each extern object allocated inside the control instance has path q such that $q \sqsubseteq p$. When using a predicate P to describe the persistent state in control instance p, we set the dependent set of P as $\{p\}$, and interpret it as P may depend on extern objects with path q such that $q \sqsubseteq p$. In general, predicate P and dependent set D should satisfy that P only depends on the objects with path q such that there exists $p \in D$ such that $q \sqsubseteq p$. In other words, for any states s and s',

$$(\forall pq.\ p \in D \wedge q \sqsubseteq p \implies s(q) = s'(q)) \implies (P(s) \iff P(s')). \tag{3.1}$$

The modification set M is interpreted in the same way: program c may only modify extern objects with path q that there exists $p \in M$ such that $q \sqsubseteq p$. If P does not depend on any objects q modified by c, i.e. there does not exist p, q, r such that

$$p \in D \wedge r \sqsubseteq p \wedge q \in M \wedge r \sqsubseteq q, \tag{3.2}$$

then P still holds after executing c. To simplify (3.2), let

$$p \parallel q \quad := \quad \forall r.\ \neg(r \sqsubseteq p \wedge r \sqsubseteq q), \tag{3.3}$$

[2]Here prefix is considered in terms of lists of identifiers. For example, a.b is a prefix of a.b.c, but not a prefix of a.bb. And p is also a prefix of itself.

[3]$p \sqsubseteq q$, $p \sqsubset q$, and $p \sqsupset q$ are defined accordingly.

which read as paths p and q are disjoint. In other words, $p \parallel q$ means one cannot get the same r by appending to p and q, so p and q must branch somewhere. Thus, $p \parallel q$ has an alternative definition that there exist path r as the common prefix and identifiers a and b such that

$$a \neq b \wedge p \sqsubseteq r.a \wedge q \sqsubseteq r.b. \tag{3.4}$$

This gives a method to efficiently test disjointness. And we define disjointness between sets as

$$D \parallel E \quad := \quad \forall p \in D.\ \forall q \in E.\ p \parallel q,$$

and (3.2) can be rewritten as $D \parallel M$.

When handling hierarchical predicates, it is preferable for the dependent sets to be automatically inferred from the predicates instead of figured out manually. So we define a syntax of hierarchical predicates, denoted by P:

$$\begin{aligned}
P &::= P_{\text{pure}} \mid p \mapsto v \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P \\
&\quad \mid \exists x.P(x) \mid \forall x.P(x) \mid \mathsf{wrap}_D(P) \\
P_{\text{pure}} &::= \text{pure predicate} \\
p &::= \text{path} \\
v &::= \text{object value} \\
D &::= \text{path set.}
\end{aligned}$$

It includes atoms (pure predicates and singletons), normal logical combinators, and a wrapper wrap_D to replace the dependent set with set D that dominates the original dependent set. The logical combinators and quantifiers are interpreted as usual. The wrapper wrap_D does not change the interpretation of the predicate but only the dependent set. Each hierarchical predicate P has its dependent set $\delta(P)$ determined

$$\frac{}{\mathsf{WF}(P_{\text{pure}})} \qquad \frac{}{\mathsf{WF}(p \mapsto v)} \qquad \frac{\mathsf{WF}(P_1) \qquad \mathsf{WF}(P_2)}{\mathsf{WF}(P_1 \wedge P_2)} \qquad \frac{\mathsf{WF}(P_1) \qquad \mathsf{WF}(P_2)}{\mathsf{WF}(P_1 \vee P_2)}$$

$$\frac{\mathsf{WF}(P)}{\mathsf{WF}(\neg P)} \qquad \frac{\mathsf{WF}(P(x)) \qquad \delta(P(x)) \text{ does not depend on } x}{\mathsf{WF}(\exists x.P(x))}$$

$$\frac{\mathsf{WF}(P(x)) \qquad \delta(P(x)) \text{ does not depend on } x}{\mathsf{WF}(\forall x.P(x))} \qquad \frac{\mathsf{WF}(P) \qquad \delta(P) \sqsubseteq D}{\mathsf{WF}(\mathsf{wrap}_D(P))}$$

Figure 3.2: Inference rules for well-formedness

by its syntax:

$$\begin{aligned} \delta(P_{\text{pure}}) &= \varnothing & \delta(p \mapsto v) &= \{p\} \\ \delta(P_1 \wedge P_2) &= \delta(P_1) \cup \delta(P_2) & \delta(P_1 \vee P_2) &= \delta(P_1) \cup \delta(P_2) \\ \delta(\neg P) &= \delta(P) & \delta(\exists x.P(x)) &= \delta(P(x)) \\ \delta(\forall x.P(x)) &= \delta(P(x)) & \delta(\mathsf{wrap}_D(P)) &= D \end{aligned}$$

Predicate P and its syntactic dependent set $\delta(P)$ might not satisfy (3.1). We say P is well-formed if P and $\delta(P)$ satisfy (3.1), i.e.

$$\mathsf{wf}(P) \;:=\; (\forall pq.\; p \in \delta(P) \wedge q \sqsubseteq p \implies s(q) = s'(q)) \implies (P(s) \iff P(s')). \quad (3.5)$$

In order to derive $\mathsf{wf}(P)$ from the syntax of P, we introduce a set of inference rules for syntactic well-formedness $\mathsf{WF}(P)$ as shown in Figure 3.2. The first five rules do not have side conditions. The rules for quantified predicates require that $\delta(P(x))$ does not depend on the quantified variable x. As the dependent set is calculated syntactically, this means x either does not appear in the path p in $p \mapsto v$, or the occurrences in paths are wrapped by wrap_D where D does not depend on x. The rule for wrap_D requires that the original dependent set $\delta(P)$ is covered by the new

dependent set D, denoted by $\delta(P) \sqsubseteq D$. It is defined as

$$E \sqsubseteq D \quad := \quad \forall p \in E.\ \exists q \in D.\ p \sqsubseteq q.$$

Theorem 3.1 (Soundness)**.** *For any predicate P, if $\mathsf{WF}(P)$ is derivable from the inference rules in Figure 3.2, then we have $\mathsf{wf}(P)$ as defined in (3.5).*

Proof. The proof is by induction on the inference tree. Most of the rules are straightforward, except the last rule. Assume we have $\delta(P) \sqsubseteq D$ and $\mathsf{WF}(P)$. The condition $\delta(P) \sqsubseteq D$ means

$$(\forall pq.\ p \in \delta(P) \wedge q \sqsubseteq p \implies s(q) = s'(q)) \implies (P(s) \iff P(s')).$$

The condition $\mathsf{WF}(P)$ implies $\mathsf{wf}(P)$ by induction. We need to prove $\mathsf{wf}(\mathsf{wrap}_D(P))$, which is

$$(\forall pq.\ p \in D \wedge q \sqsubseteq p \implies s(q) = s'(q)) \implies (P(s) \iff P(s')),$$

as wrap_D does not change the interpretation. So it is enough to show

$$(\forall pq.\ p \in D \wedge q \sqsubseteq p \implies s(q) = s'(q)) \implies (\forall pq.\ p \in \delta(P) \wedge q \sqsubseteq p \implies s(q) = s'(q))$$

We only need to show, for every path q,

$$(\exists p.\ p \in \delta(P) \wedge q \sqsubseteq p) \implies (\exists r.\ r \in D \wedge q \sqsubseteq r) \tag{3.6}$$

Let p be as in the first existential quantifier, so $p \in \delta(P)$ and $q \sqsubseteq p$. By the definition of $\delta(P) \sqsubseteq D$, there exists a path $r \in D$ such that $p \sqsubseteq r$. By the definition of $\sqsubseteq$, it is transitive, so $q \sqsubseteq r$. So we have (3.6). □

Recall the program in Figure 2.1. In Section 5.3, we are going to relate the P4 program with a functional program that serves as its functional model, in order to prove properties about the program. So we use hierarchical predicates to relate P4 state with model values. The predicate $\mathsf{row\_repr}(p, r)$ describes that a Row at p represents a row model value r:

$$\mathsf{row\_repr}(p, r) \quad := \quad \mathsf{wrap}_{\{p\}}(p.\mathsf{reg} \mapsto r).$$

Each Pane has three rows, so its representation predicate pane_repr is the conjunction of row_repr on each P4 instance $\mathsf{row\_}i$ and its corresponding model value $a[i]$:

$$\mathsf{pane\_repr}(p, a) \quad := \quad \mathsf{wrap}_{\{p\}} \left(\bigwedge_{i=1,2,3} \mathsf{row\_repr}(p.\mathsf{row\_}i, a[i]) \right).$$

Similarly, a filter (the name SBFilter is reserved for later use) consists of four panes and auxiliary registers clear_index and timer, and the predicate filter_repr is defined as

$$\begin{aligned} \mathsf{filter\_repr}(p, f) \quad := \quad \mathsf{wrap}_{\{p\}} \Bigg(& p.\mathsf{clear\_index} \mapsto \cdots \wedge p.\mathsf{timer} \mapsto \ldots \\ & \wedge \bigwedge_{i=1,2,3,4} \mathsf{pane\_repr}(p.\mathsf{pane\_}i, f.a[i]) \Bigg). \end{aligned}$$

It is easy to check well-formedness of these predicates. The dependent set of all these predicates are $\{p\}$ for their respective p's. So the client can handle them without knowing their internal implementation.

A commonly used technique to prove correctness of programs is layer refinement. That is, a low-level functional model (like above) is related to a high-level functional model by a simulation relation $f \mathrel{\mathcal{R}} f'$, where f is a value from the low-level model and f' is a value from the high-level model. They we prove operations preserve the simulation relation, then we only need to reason about the high-level model instead

of the low-level model. When we want to describe the P4 state of a SBFilter with a high-level functional model, we can define predicate using the quantifier in the syntax:

$$\mathsf{SBFilter\_repr}(p, f') \quad := \quad \exists f.\ \mathsf{filter\_repr}(p, f) \ \wedge \ f \ \mathcal{R} \ f'.$$

So the client can use the high-level functional model directly.

3.2 Program logic

Based on hierarchical predicates, we give our Verifiable P4 program logic that is useful and proved sound with respect to the operational semantics. As mentioned above, it is a Hoare logic with dependent set and modification set. Given a P4 program, the global environment Γ is directly calculated, so we treat Γ as a known constant. The form of function specification is as shown in Figure 3.3. First, there are three layers of quantified variables: WITH $\vec{x}$, WITH $\vec{y}$, and EX $\vec{z}$. All the three layers of quantified variables can be dependently typed in Coq's type system, so they can include propositions. The outer layer WITH $\vec{x}$ provides universally quantified variables $\vec{x}$ whose scope is the whole specification. PATH p indicates "when executing in an instance at path p". There is a common pattern, for a P4 class Foo,

$$\mathsf{WITH}\ p\ (\_ : \Gamma(p) = \mathsf{Foo}), \quad \mathsf{PATH}\ p, \ldots$$

This pattern gives a specification for every instance p of class Foo, where $\Gamma(p)$ is an abuse of notation for "getting the class name of instance p in Γ". Although it is a nice form of specification, we have not yet established a method to assist its proof in our verifier (Chapter 4), so it is currently only used in specifications of extern methods, whose correctness is directly proved using the semantics without using the verifier. For internal functions, we use a pragmatic approach that is setting p to be

Full form of function specification:

WITH $\vec{x}$, PATH p, MOD m M
  WITH $\vec{y}$,
    PRE (ARG $\vec{P}$, MEM $\vec{Q}$, EXT $\vec{R}$)
    POST (EX $\vec{z}$, RET v, ARG $\vec{P}'$, MEM $\vec{Q}'$, EXT $\vec{R}'$)

Quantifier-free form of function specification:

PATH p, MOD m M
  PRE (ARG $\vec{P}$, MEM $\vec{Q}$, EXT $\vec{R}$)
  POST (RET v, ARG $\vec{P}'$, MEM $\vec{Q}'$, EXT $\vec{R}'$)

Figure 3.3: Function specification

the path of each instance, respectively, and running the proof script multiple times (see Chapter 5).

WITH $\vec{y}$ and EX $\vec{z}$ are standard quantifiers in function specifications, as used in VST [7], VeriFast, and Dafny. WITH $\vec{y}$ means variables $\vec{y}$ are shared between the precondition and the postcondition. They can be any value in a particular function call, so if the precondition is satisfied with certain $\vec{y}$, the postcondition is satisfied with the same $\vec{y}$. EX $\vec{z}$ are existentially quantified for the postcondition.

In the C language, we do not need to mention local variables in function specifications, since each function call creates a new stack frame. But P4 has local variables visible in different functions (actions) in the same class, for example,

```
control Ingress {
    bit<8> x;
    action incr() { x = x + 1; }
    apply { x = 0; incr(); }
}
```

The variable x is visible in both the apply block and the incr action. As defined in our semantics, a P4 function call only creates a new stack frame if it calls to a

different instance (which often means a different control block, but not always). So P4 functions that are called in the same control block need to describe local variables that the functions use in their preconditions and postconditions, specify which variables they modify.

For clarity, we explain the rest of the function specification in the quantifier-free form as shown in the second half of Figure 3.3. The MOD-clause MOD m M specifies the *variable modification set* m and the *object modification set* M. m is either a set of variables (indicated with paths) or "$*$", which means any variable can be modified. M is the modification set as described in Section 3.1. These two sets are placed outside of quantified variables $\vec{y}$ and $\vec{z}$, so that their validity can be proved and applied automatically without supplying $\vec{y}$ and $\vec{z}$, which depends on symbolic program data and sometimes requires human effort to figure out.

The precondition of the form ARG $\vec{P}$, MEM $\vec{Q}$, EXT $\vec{R}$, where $\vec{P}$ is a list of values corresponding to in parameters (including inout parameters) of the function, $\vec{Q}$ is a list of pairs of paths and values that describes local variables, $\vec{R}$ is a list of well-formed hierarchical predicates. The postcondition is similar, but with an additional return value v.

Bitwise abstract interpretation All values used in preconditions and postconditions, including v and values in $\vec{P}$ and $\vec{Q}$, are *bitwise abstract values*. The same applies for assertions in Hoare triples (see below). Recall that, in Section 2.3, we mentioned uninitialized variables in P4 need to be treated explicitly, so variables are stored using storable values whose bits can take values within 0, 1, and $\perp$ (uninitialized).

During verification, we have not only uninitialized bits but also *unknown* bits, for example, when uninitialized bits are nondeterministically converted to 0/1 during parameter copy-in/copy-out. Although these bits are "initialized", it is counterintuitive

Abstract value \ Storable value	0	1	$\bot$
0	T	F	F
1	F	T	F
$\bot$	T	T	T

Table 3.1: Truth table of bitwise abstract interpretation

for programmers to keep track of their "initialization" and reason about nondeterministically converted values, so our experience shows most P4 programs do not rely on such quirky behavior. If the program logic directly reflects initialized and uninitialized bits, the verification using the program logic becomes inconvenient. For example, consider that a struct value with two single-bit fields $\{|\mathsf{x} := 0; \mathsf{y} := \bot|\}$ is passed into a function as an argument. According to the semantics of argument evaluation, this value is determinized to $\exists b.\ \{|\mathsf{x} := 0; \mathsf{y} := b|\}$. Although the programmer is happy to treat y as uninitialized, it need to be handled explicitly in such a program logic. It is inconvenient: (1) it requires more often manipulation of quantifiers; (2) all the assertions need to identify whether each field is initialized or not.

In order to resolve the inconvenience and simplify verification, we apply abstract interpretation [13] on a simple abstract domain, namely bitwise abstract values. The construction of bitwise abstract values coincides with storable values, i.e. each bit takes values within 0, 1, and $\bot$, but $\bot$ is interpreted as "don't care", so it can correspond with any bit value, as shown in Table 3.1. This is also used to encode a struct with some of the fields unspecified. This abstract interpretation is usually unnoticeable by the verifier user.

Hoare triple In Verifiable P4 program logic, a Hoare triple is of the form $\{X\}stmt\{Y, Z\}$, where Y is the normal postcondition and Z is the return postcondition. This method of manipulating control flow signals in Hoare logic is the same as ret_assert in VST [1, Chapter 24], so we skip detailed discussion, and only consider the

Simplified rule for assignment:

$$\frac{\Gamma, p, \vec{P} \vdash exp \Downarrow v}{\Gamma, p \vdash \{\mathsf{MEM}\ \vec{P}, \mathsf{EXT}\ \vec{Q}\} x@(\mathsf{inst}\ p') := exp \{\mathsf{MEM}\ \vec{P}[p' \mapsto v], \mathsf{EXT}\ \vec{Q}\}}$$

Simplified function call rule (when $p = q$):

$$\frac{\begin{array}{c}\Gamma, p \vdash f() \Downarrow (q, func) \qquad \Gamma, func \text{ satisfies } \mathsf{PATH}\ q, \mathsf{MOD}\ m\ M, \mathsf{PRE}\ X_f, \mathsf{POST}\ Y_f \\ p = q \qquad X \vdash X_f \qquad (X, Y_f, m, M) \Downarrow_{\mathsf{merge}} Y\end{array}}{\Gamma, p \vdash \{X\} f(); \{Y\}}$$

Figure 3.4: Sample program logic rules

form of $\{X\}stmt\{Y\}$. The assertions X and Y are of the form "MEM $\vec{P}$, EXT $\vec{Q}$", where the meaning of MEM and EXT is the same as in function specifications.

3.2.1 Program logic rules

Figure 3.4 shows a few simplified program logic rules, which we use to explain the interesting aspects of the program logic.

The first rule is for assignment, which corresponds to the semantic rule in Figure 2.12. When the precondition is (MEM $\vec{P}$, EXT $\vec{Q}$), we want to find the post-condition of executing $x@(\mathsf{inst}\ p') := exp$, where $\mathsf{inst}\ p'$ is the locator. The premise $\Gamma, p, \vec{P} \vdash exp \Downarrow v$ means the expression exp is evaluated to v if the variables are as described in $\vec{P}$. (We omit the conversions between normal values and storable values that use three-valued bits.) Then the post condition is simply obtained by setting $p' \mapsto v$ in $\vec{P}$. Although we do not present the soundness proof here, one can see that if is almost straightforward because the modification of s_{local} in the semantics is directly reflected in the logic rule. In comparison, using the Petr4 semantic rule in Figure 2.12 will be much more complicated to design a program logic and prove its soundness, because it is necessary to track location allocation and prove their distinctness.

The second rule in Figure 3.4 is for function calls. For simplicity, we present a special case that function f has no parameters or return value, its specification is given in quantifier-free form, and $p = q$ (explained below).

The most interesting part is the last premise, $(X, Y_f, m, M) \Downarrow_{\mathsf{merge}} Y$. It merges the assertion at the call site with the assertion from the function, using the disjointness we established in Section 3.1. It is defined as

$$((\mathsf{MEM}\ \vec{P}_1, \mathsf{EXT}\ \vec{Q}_1), (\mathsf{MEM}\ \vec{P}_2, \mathsf{EXT}\ \vec{Q}_2), m, M) \Downarrow_{\mathsf{merge}}$$
$$((\mathsf{MEM}\ \mathsf{filter}(\vec{P}_1, m) + \vec{P}_2, \mathsf{EXT}\ \mathsf{filter}(\vec{Q}_1, M) + \vec{Q}_2),$$

where $\mathsf{filter}(\vec{P}_1, m) + \vec{P}_2$ means to remove local variables in $\vec{P}_1$ that appear in m since they are modified (remove everything in $\vec{P}_1$ if m is $*$) and put together with $\vec{P}_2$, and $\mathsf{filter}(\vec{Q}_1, M) + \vec{Q}_2$ means to remove every hierarchical predicates Q in $\vec{Q}_1$ that $\delta(Q) \parallel M$ (the dependent set of Q is disjoint from M) and put together with $\vec{Q}_2$.

The meaning of the rest premises in the second rule is as follows. The rule first looks up f in Γ and finds out f is to execute function body *func* in path q. (We ignore the detailed function resolution procedure here.) The second premise states that *func* satisfies its function specification. We ignore the ARG part in X_f and Y_f, since there are no parameters. The third premise is $p = q$, which means the function body *func* is executed in the same stack frame where f is called, as defined in the semantics. In the general case, if $p \neq q$, the function body *func* is executed in a new stack frame, then the local variable assertions in Y_f and the modified local variables in m do not affect the postcondition Y. Then the rule needs the precondition X to imply function precondition X_f.

The complete program logic rules are provided as source code in Coq. We proved all these rules are sound with respect to the operational semantics in Coq. The result that we proved can be expressed as the following theorem.

Theorem 3.2 (Soundness)**.** *The semantics of the judgment* $\Gamma, p \vdash \{X\} stmt \{Y\}$ *is for every states* s *and* s'*, and signal sig, if we have* $\Gamma, p, s \vdash stmt \Downarrow (s', sig)$*, and have* $X(s)$*, then sig* $=$ `return` *and* $Y(s')$*. All Verifiable P4 program logic rules are proved in Coq with respect to the operational semantics and semantics of Hoare triples.*

Chapter 4

Tactic-Based Verifier

The previous chapter presented a sound program logic for P4 programs. But verifying a program, i.e., proving each function specification, by directly applying those rules is still a tough and laborious process, which involves picking appropriate rules, instantiating variables, simplifying terms, and resolving entailments between assertions. In this section, we describe our "Verifiable P4" verifier based on Coq's Ltac tactic programming that helps users verify P4 control blocks. The general design of the tactic-based verifier is similar to VST-Floyd [7], which is a verifier for C programs. It automatically applies programming logic rules according to the P4 program syntax. It also includes automatic simplification and resolution of assertions for both efficiency and readability of intermediate steps. Our verifier gives a practical way to construct foundational proofs showing that P4 programs implement functional models; but the amount of work to verify a program depends on how much the natural interpretation of the program differs from the mathematical language used in its specification.

We first give a simple example as a walkthrough in Section 4.1. Then we introduce the core tactics provided by the verification system and their mechanism in Section 4.2. After that, we show the automated resolution of MOD-clauses in Section 4.3.

Proof assistant Here we give some additional preliminaries for the readers who are not familiar with proof assistants. The typical workflow of interactive theorem proving in a proof assistant like Coq is as follows. The user inputs the theorem/lemma statement they want to prove, which becomes the initial proof goal in the proof assistant. The proof assistant displays the current proof goal(s), and the user inputs a command, which is called a tactic, to either resolve the proof goal or reduce it into one or more easier goals, based on their knowledge about the proof. This procedure iterates until there are no remaining proof goals, at which point the lemma is proved. The advantage of interactive theorem proving is it can prove complicated mathematical properties that are highly unlikely to become automatically provable in the near future.

A tactic is a program used to construct a proof. Tactics can vary from performing a simple proof step to automatically searching for a proof using advanced algorithms. In practice, users can summarize common patterns in proofs and create new reusable tactics. Program verification proofs often contain many common patterns, making them an ideal target for building a tactic system. For example, VST-Floyd is a tactic system for C program verification using separation logic. Coq provides a tactic language, Ltac [16], for building custom tactics.

4.1 Walkthrough

In this section, we give a simple example of program verification using our Verifiable P4 verifier. A real, large-scale application of proving a sliding-window Bloom filter will be presented in Chapter 5.

Consider a P4 control block in the V1Model architecture as shown in Figure 4.1. The first two lines list the function prototypes of read and write. The program to verify is in the Increment control block, which maintains a register array reg with a

```
// Function prototypes
void read(out T result, in bit<32> index);
void write(in bit<32> index, in T value);

// Prorgam to verify
control Increment(out bit<8> x) {
    register<bit<8>>(1) reg;
    apply {
ℓ1      reg.read(x, 0);
ℓ2      x = x + 1;
ℓ3      reg.write(0, x);
    }
}
```

Figure 4.1: An example P4 control block in V1Model

```
Definition p : path := ["incr"].
Definition Increment_spec : func_spec :=
  WITH,
    PATH p
    MOD * [p]
    WITH (x : Z),
      PRE
        (ARG [], MEM [], EXT [p.reg ↦ (ObjRegister [P4Bit 8 x])])
      POST
        (RET Null, ARG [P4Bit 8 (x + 1)], MEM [],
            EXT [p.reg ↦ (ObjRegister [P4Bit 8 (x + 1)])])
```

Figure 4.2: Function specification of Increment

single cell of type bit<8> (8-bit unsigned integer), referred as reg[0]. Line ℓ_1 loads the value of reg[0] to x. Line ℓ_2 increases x by 1 (modulo 2^8). Line ℓ_3 writes x back to reg[0]. Overall, this function increases the value in reg by 1 and passes the new value out as out parameter x.

Figure 4.2 shows the function specification written in Coq (simplified for presentation). The value in the register reg before calling apply is represented by a mathematical integer $(x : \mathsf{Z})$ in the WITH-clause. The state to track in the function specification is simple: the precondition has neither in arguments (ARG) nor local

variables (MEM), and the EXT part contains a singleton extern predicate saying the register at path p.reg is an 8-bit unsigned integer, which is the 8-bit representation of x (x does not need to be between 0 and 255); the postcondition contains return value Null (as the control block does not have a return value), out argument $x + 1$ (ARG), no assertion on local variables (MEM), and a singleton extern predicate saying the new value of the register p.reg is $x + 1$.

The proof of Increment_spec using the verifier is a step-by-step forward symbolic execution, as in VST-Floyd [7]. Roughly speaking, the user first applies the start_function tactic to reduce from Increment_spec to a Hoare triple across the function body, of the form

$$\{X_0\}\ell_1; \ell_2; \ell_3\{Y\}.$$

Then the user uses the step[1] and step_call tactics to infer the postcondition of each statement. It would be ideal if the inferred postcondition is the strongest postcondition: it is indeed the strongest for assignment statements, but it is not the strongest for function calls. Because function calls are reasoned using function specifications, which contain quantifiers, the strongest postcondition of a function call also includes quantifiers. Those quantifiers are cumbersome in entailment resolution and human interaction, so an instance of quantifier variables is inferred to generate a postcondition, which is not the strongest but enough to prove the program in most cases. Let X_1 be the postcondition of executing ℓ_1 from X_0. Then the tactic applies program logic rules to reduce the goal to

$$\{X_1\}\ell_2; \ell_3\{Y\}.$$

[1]These tactics are called forward and forward_call in VST. We call them step instead, since "forward" is a commonly used term in networking.

```
Proof.
  start_function.
  step. (* initialization *)
  step_call register_read_body.
  { entailer. } (* function precondition *)
  { simpl. lia. } (* resolve arithmetic conditions *)
  { simpl. lia. } (* resolve arithmetic conditions *)
  step.
  step_call register_write_body.
  { entailer. } (* function precondition *)
  { simpl. lia. } (* resolve arithmetic conditions *)
  { simpl. lia. } (* resolve arithmetic conditions *)
  step. (* empty statement *)
  entailer.
Qed.
```

Figure 4.3: Proof script of Increment

The next two steps reduce the proof goal to $\{X_2\}\ell_3\{Y\}$ and then to $X_3 \implies Y$ in a similar way. Finally, the user uses the entailer tactic to resolve the entailment $X_3 \implies Y$.

Figure 4.3 shows a more concrete version of the Coq proof script to prove Increment_spec. In addition to the aforementioned start_function, step, step_call, and entailer tactics, Figure 4.3 uses step two more times to handle an implicit initialization statement at the beginning and an empty statement at the end, respectively. Also, after applying step_call with a function body proof register_read_body or register_write_body, the user uses entailer to prove the function precondition is satisfied. The variables in the WITH-clauses in register_read_body and register_write_body, i.e. quantified variables, are also inferred by entailer. Some arithmetic proof goals are also needed to prove after function calls, which are used to ensure the index to access the register is in bounds. They can be proved easily by Coq's built-in automated tactics simpl and lia, which are not in the interest of this thesis.

The step_call tactic requires the user to provide the proof of the callee function, such as register_read_body and register_write_body. In this example, since

register_read_body and register_write_body are proofs of extern methods, we have proved them by hand directly with respect to the operational semantics in our library. These methods are a fixed set for each architecture (such as V1Model and Tofino), so we can provide as a library. In the case of P4 functions (control blocks, tables, and actions), the user should first specify and prove the callee function in the same way as the example. P4 language does not allow recursive functions, so the user can prove the functions in order from the lowest layer to the highest. The step_call tactic requires the user to provide the proof instead of automatically searching in a database. This is because functions, especially control blocks, may have separate specifications for different cases, and automatic search may not find the proof for the desired specification.

4.2 Tactics

In this section, we give detailed explanation of the structure of the tactic system, functionality of tactics, and their implementation. Besides start_function, step, and entailer mentioned in the previous section, there are two more groups of tactics: predicate manipulating tactics that perform necessary transformations during the step-by-step proof, and table tactics that handles P4 tables.

start_function tactic (for normal functions) The start_function for normal functions is used at the beginning of each normal function proof, including actions and control blocks. It reduces a function specification to a Hoare triple. It first resolves the MOD-clause as Section 4.3 will show. Then it simulates the copy-in procedure by converting the in-argument assertions in the precondition into local variable assertions. It also infers the postcondition of Hoare triple to prove from the function postcondition by generating the weakest precondition of the copy-out procedure.

step tactics There are a three different step tactics used depending on the next program statement:

- step: Execute the next program statement, including built-in function calls but not other function calls.
- step_if/step_if Y: This tactic handles if-statements. Y is the postcondition of the if-statement, which can be omitted if there are no other statements after the if-statement. step_if makes one subgoal for the then-clause and one for the else-clause.
- step_call lem $\vec{v}$: This tactic processes a call to a function whose specification is proved by the lemma lem. $\vec{v}$ is an optional list of Coq terms to instantiate the second WITH-clause in the function specification proved in lem. If some of the WITH-variables are not instantiated (including that $\vec{v}$ is not provided, $\vec{v}$ is shorter than the list of variables, and some terms in $\vec{v}$ are filled with "_"), the uninstantiated variables are filled with unification variables, which means leaving them as values to be determined. These values are often inferable from the values in the current assertion and function arguments, and automatically filled by the following entailer tactic. So the user only needs to provide them manually in some special cases.

All the step tactics include simplifying expressions with best effort.

Predicate manipulating tactics During the proof of a function body, the user sometimes needs to transform the precondition in the Hoare triple before proceeding to the next step. For example, suppose the precondition contains a encapsulated hierarchical predicate, e.g. $\mathsf{wrap}_p(P \wedge Q \wedge R)$, and the next step is a function call that needs to match the contents inside wrap. Then she wants unpack the predicate into three single predicates P, Q and R using the normalize_EXT tactic below before calling step_call. The list of predicate manipulating tactics is as follows.

- normalize_EXT: Flatten the hierarchical predicates by breaking wrap and conjunctions, and then move the members to the top level of the EXT part as much as possible.
- Intros: Move an existentially quantified variable from the precondition to the context.
- Intros_prop: Move a pure proposition (P_{pure}) from the precondition to the context.
- P4assert P: This tactic is used when a pure proposition P is needed for manual transformation (e.g. rewriting with an equality). After applying P4assert P, the user first proves the precondition implies P, then P will be available in the context for the remaining proof.

entailer tactic The entailer tactic is used to resolve or simplify assertion entailments, mostly in the form of $\mathsf{MEM}\ \vec{Q}, \mathsf{EXT}\ \vec{R} \implies \mathsf{MEM}\ \vec{Q'}, \mathsf{EXT}\ \vec{R'}$ or $\mathsf{ARG}\ \vec{P}, \mathsf{MEM}\ \vec{Q}, \mathsf{EXT}\ \vec{R} \implies \mathsf{ARG}\ \vec{P'}, \mathsf{MEM}\ \vec{Q'}, \mathsf{EXT}\ \vec{R'}$. These goals appear at the end of a function, a block, or at a function call (where we need to prove that the current precondition satisfies the function precondition). This tactic is safe in most cases: safe means it does not turn a provable goal into any unprovable goals. The exception is the case where a pure proposition needs to be proved from the precondition first, before reducing the entailment.

The reduction of entailment in this tactic is done by matching the corresponding parts in the precondition and the postcondition. Function arguments are matched by their positions. Local variables are matched by their names. Both function arguments and local variables are reduced to the order relation (inclusion) in the abstract domain, and resolve automatically if possible. Singleton extern predicates, which are of the form $p \mapsto v$, i.e. the value of the extern object p has value v, are matched with any singleton with the same p. Other extern predicates, if they are not directly resolved,

are not be able to be matched. All the resolution steps include instantiation of unification variables, which are often introduced by postconditions with existential variables and by function calls for which the user has not (manually) provided an instantiation for the WITH-clause.

There are some other tactics for resolving entailments. The Exists tactic instantiates existential quantifiers in the postcondition. Predicate manipulating tactics mentioned above are also applicable on entailments.

Table tactics There are two tactics for tables:

- start_function (for tables): This tactic is used at the beginning of proofs of tables, which are considered as a kind of function. It only supports the case that table entries are known. It evaluates table key expressions and matches them against the entries. It reduces the proof goal into each entry's action call.
- table_action lem: After splitting a table into the cases for each entry, table_action lem is used on each goal to apply the function body proof lem of the action called by the table in this case and reduce the proof goal into two entailments, before and after the function call, respectively.

Summary With these five groups of tactics handling P4 execution, users can easily verify P4 programs without paying further attention to P4 features. They only need to focus on normal Coq proof specific to their theory and algorithm, such as arithmetic, data structures, and packet forwarding protocols.

4.3 Automated proof of MOD-clauses

In a function specification, we need to have a MOD-clause of the form MOD m M to indicate modified local variables and extern objects, so that the function specification

can compose with the assertion at the call site. When the start_function tactic reduces a function specification to a Hoare triple, it first proves the MOD-clause. P4 is a simple language and the phase distinction between instantiation and execution has eliminated higher-order runtime behavior, so the MOD-clause should be proved automatically.

To prove a function only modifies m and M, we reason on the syntactic structure of the function body.[2] The breakdown proof is done by applying proven-sound deduction rules similar to program logic rules, e.g.

$$\frac{\Gamma, p \vdash \mathit{stmt}_1 \ \mathsf{MOD}\ m\ M \qquad \Gamma, p \vdash \mathit{stmt}_2 \ \mathsf{MOD}\ m\ M}{\Gamma, p \vdash \mathit{stmt}_1; \mathit{stmt}_2 \ \mathsf{MOD}\ m\ M} \ .$$

Simple statements without function calls are also proved by applying such rules. Function call statements can be proved either by stepping into the function or by using its function specification. Function specifications designed in Chapter 3 has the MOD-clauses outside of quantified variables, except the top-level quantified variables ($\vec{x}$), which are expected to be inferable. So instantiating other quantified variables is not required, making it easy to automatically apply the MOD-clauses in function specifications.

The proof search is based on Coq's built-in eauto search engine, so we do not have to build and optimize a new one. The deduction rules are designed such that each step has at most one applicable rule, so the proof search does not backtrack and is reasonably efficient. The eauto search engine also allows the user to add proved function specifications into the rules, so the caller does not need to step in to the

[2]It is possible that a variable is first modified and then modified back. In this case, these syntax-directed proof rules are incomplete. However, it is uncommon in practical programming, and if it occurs, it is no harm to add them to the MOD-clause and maintain their value in the precondition and the postcondition.

function. In the large-scale example presented in Chapter 5, the proof search takes at most about 3 seconds for a function.

Discussion Another approach to prove MOD-clauses automatically is *proof by reflection*, which is the method of proving a property by writing a testing program for the property and proving its correctness. So the proof is simply by evaluating the program. We did not use proof by reflection for MOD-clauses because proof by reflection is usually considered more heavyweight. It requires more effort to implement and modify. We refer the reader to a fantastic book [10] for more information about proof by reflection.

Chapter 5

Application Demonstration: Sliding-Window Bloom Filter

In this section, we present an application case of our verification framework to demonstrate its utility. Using this example, we show our Verifiable P4 is capable of modularly verifying P4 programs at a larger scale, and proving stateful properties of processing a sequence of packets (called flow properties).

Consider a switch that connects an internal network with an untrusted external network. Adversaries from the external network may perform distributed denial-of-service (DDoS) attacks on internal hosts. Although the switch is capable of handling the huge traffic, the internal hosts cannot, because they are based on CPUs. One policy for protecting internal hosts is to allow incoming traffic recently requested by internal hosts and block most unsolicited incoming traffic. This is a kind of stateful firewall, which can be implemented based on a sliding-window Bloom filter (SBF). SBF is an approximate data structure for maintaining a set whose elements expire after a fixed amount of time. We take a P4 implementation of such a stateful firewall on the Tofino architecture with a single pipeline as a case study on verifying stateful P4 programs. It is acceptable that the stateful firewall allows a small portion of

unsolicited traffic that internal hosts can handle, but it is necessary to prove that the stateful firewall does not block any benign traffic. We formalize this property as a formula on the history of packets and give an end-to-end verification.

5.1 Sliding-window Bloom filter

Programmable switches have limited resources. Therefore *sketching data structures* are used to implement features with tiny space and time (usually in terms of hardware pipeline stages). The trade-off is tolerating errors with a low probability.

A famous example of sketching data structures is the Bloom filter [4], which approximately supports querying existence of elements in a set. It supports two operations: (1) inserting an element, and (2) querying an element and reporting "likely in the set" (positive) or "definitely not in the set" (negative).

Beyond the Bloom filter, *sliding-window Bloom filter* (SBF) is the task of maintaining elements in a set, and *automatically removing each element approximately T time units after its most recent insertion.* For fixed time interval T and tolerance parameter δ, the data structure provides two operations: (1) inserting an element with a timestamp (timestamps of operations must be nondecreasing), and (2) querying an element with a timestamp and reporting "likely inserted within time $(\mathbf{1} + \delta)T$" (positive) or "definitely not inserted within time T" (negative). A classic algorithm for sliding-window Bloom filter is using multiple Bloom filters periodically, which we will show below. Although there is a more efficient algorithm [32], that algorithm cannot be implemented on the Tofino architecture (nor on other hardware P4 architectures).

We choose the sliding-window Bloom filter on the Tofino architecture generated by the CatQL code generator [35] as the verification target. In the following sections, we will prove that the P4 program always returns a positive result if the element is in the set. This is called *no-false-negative property.* On the other hand, it is also desired

that the probability of returning a positive result when the element is not in the set (low-false-positive-rate property). This property requires a probabilistic model of elements and hash functions used in the Bloom filter, so we leave it as future work. But the no-false-negative proof includes useful steps for the low-false-positive-rate property.

A sliding-window Bloom filter (SBF) consists of k Bloom filters, called *panes*. Each pane has r *rows*, each associated with a hash function h_i. The hash function array $(h_1, h_2, \dots)$ is the same for all the panes.[1] Each row is a hash table of S bit-value slots initially filled with 0s.[2] Inserting x into a Bloom filter is to set the $h_i(x)$-th slot to 1 for each row i. Querying x in a Bloom filter is to take conjunction over the $h_i(x)$-th slot of all rows i. If x is already inserted, then querying x must be 1 as the $h_i(x)$-th slot is set to 1 for all i. So if the result is 0, it is "definitely not in the set". One can expect the false-positive rate is low: If γ is the density of 1s in each row, an ideal analysis of hash functions gives the probability of getting 1 on query of a not-in-the-set x is γ^r.

An SBF maintains time by dividing it into pieces of the same size, called *time panes*, as shown in Figure 5.1. Each time pane is associated with an SBF pane, which is allocated circularly. In this example, there are $k = 4$ SBF panes. At any moment, there are $k-1$ *working panes* (denoted by solid boxes) and 1 *clearing pane* (denoted by a dashed box). If the present time is in pane3 as shown in the first row of Figure 5.1, pane1 through pane3 are working panes, and new elements will be inserted into the latest working pane, pane3. When querying, all working panes are queried and the overall result is their disjunction. So a query is guaranteed no-false-negative, if the element has been inserted between the beginning of the earliest working pane and

[1] One may choose to use different hash functions for each pane, for example, in order to prevent some exploitation of the hash functions. These are not considered in this thesis.

[2] The implementation of panes (Bloom filters) is slightly different from the standard implementation, which uses a single hash table for all hash functions. This difference is because the Tofino architecture only allows one update per register (the underlying implementation of hash table) per packet. This difference does not affect the essence of the Bloom filter.

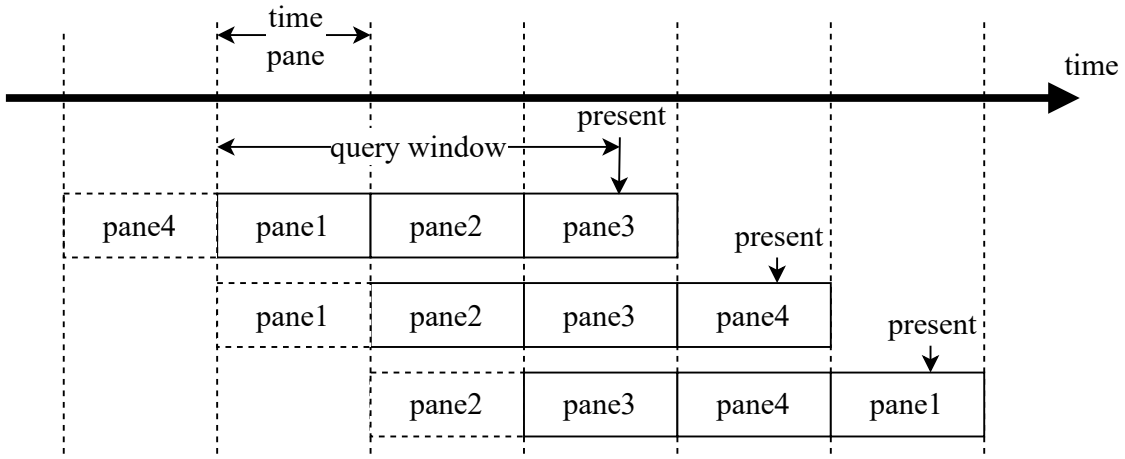

Figure 5.1: Panes in a sliding window Bloom filter

the present time, which is the query window shown in the figure. When the present time moves to the next pane, as shown in the second row, pane4 is finished clearing and reused for new elements. Pane1 is outdated and going to be cleared, and the left endpoint of the query window becomes the beginning of pane2.

To clear a pane means to set it to all zeros. This cannot be done instantaneously on Tofino (or similar architectures). Instead, with each packet processed, one more bit of the clearing pane will be set to zero. This is called the *clearing duty*. So we require an assumption in our verification result: there are always *at least s packets in each time pane* to serve the clearing duty, where s is the number of slots in each row. We can guarantee this assumption using the Tofino switch's *packet generator*, a component just before the P4 pipeline that can be configured to insert extra packets at the desired rate.

Which elements are maintained by this data structure? As shown in Figure 5.1, the length of the query window varies between $(k-2)$ and $(k-1)$ time panes. So the aforementioned time T is the lower bound of query window size, i.e. $(k-2)$ time panes. If an element is inserted within time T, it is guaranteed no-false-negative.

Rest of the chapter Next, we will show some key points in the P4 program of SBF, which provides further illustration of P4 programming, and explains the tricks used to implement the algorithm efficiently that makes the verification challenging. Although the P4 program is clumsy, the functional model (Section 5.3.1) and the axiomatic interface (Section 5.2.1) to characterize SBF are much easier to read and reason about.

5.1.1 P4 implementation

The P4 program of SBF that we verify is as follows. This program is generated by the CatQL code generator. This program is the extended version of Figure 2.1. The program has been significantly simplified for presentation,[3] but it unavoidably remains very complicated, as it was crafted by CatQL to maximize the efficient utilization of hardware resources. P4 language features, such as registers, actions and tables, also add complexity to the program.

This program demonstrates the complexity of P4 programs. Nonetheless, it follows the natural modular structure of SBF, which contains three layers from bottom to top: row, pane, and SBF. Each control block corresponds to a layer. So we can verify it modularly.

```
1  control Row(in bit<8> api, in bit<18> index, out bit<8> res) {
2      Register<bit<8>, bit<18>>(262144, 0) reg;
```

The program starts with the P4 implementation of a Bloom filter row. It has two in parameters: api takes value from NOOP (no operation), INSERT, QUERY, and CLEAR; it represents the operation to take on the row; index represents the index of the slot to modify or query, which is a hash value of the key. Row has one out

[3]The full version is available in the appendix.

```
U execute(in I index) {
    U rv;
    T value = reg.read(index);
    apply(value, rv);
    reg.write(index, value);
    return rv;
}
```

Figure 5.2: Behavior of execute in RegisterAction

parameter, namely res, which is used to pass out the query result. Line 2 defines reg, which is a Tofino register with $2^{18} = 262144$ slots, initialized with zeros.

The P4 implementation is a bit more verbose than in conventional imperative programming languages, because the P4 implementation is closely related to hardware.

```
3   RegisterAction<bit<8>, bit<18>, bit<8>>(reg) regact_insert = {
4       void apply(inout bit<8> value, out bit<8> rv) {
5           value = 1;
6           rv = 1;
7       }
8   };
9   action act_insert() {
10      res = regact_insert.execute(index);
11  }
12  action act_query() { ... } /* Similar to regact_insert and act_insert */
13  action act_clear() { ... } /* Similar to regact_insert and act_insert */
```

Line 3 defines a register action regact_insert on register reg, which is used to update reg when api is INSERT. It contains an abstract method (c.f. Section 2.2.2) apply, which is called by the register action itself when its execute method is called on line 10. The behavior of execute is illustrated in Figure 5.2. In particular, regact_insert.execute(index) sets the index-th slot to 1 and returns 1. Line 9 wraps the register action as an action. The register actions are defined and wrapped as actions similarly for QUERY and INSERT.

```
14  table tbl_row {
15      key = { api : ternary; }
```

```
16          actions = {
17              NoAction(); act_insert(); act_query(); act_clear();
18          }
19          const entries = {
20              NOOP : NoAction();
21              INSERT : act_insert();
22              QUERY : act_query();
23              CLEAR : act_clear();
24          }
25      }
26      apply {
27          tbl_row.apply();
28      }
29  }
```

Then we define a match-action table tbl_row, which corresponds to a hardware match-action table. This table is responsible for calling the appropriate action based on the value of api. The control block's apply block, which is its entrance, is just calling tbl_row.

```
30  control Pane(inout pane_md_t pane_md) {
31      Row() row_1; Row() row_2; Row() row_3;
32      apply {
33          row_1.apply(pane_md.api, pane_md.index_1, pane_md.res_1);
34          row_2.apply(pane_md.api, pane_md.index_2, pane_md.res_2);
35          row_3.apply(pane_md.api, pane_md.index_3, pane_md.res_3);
36      }
37  }
```

The implementation of a pane is simple. It just dispatches the operation to each of its rows, and returns the result from each row as-is.

```
38  control SBFilter(in bit<64> key, in bit<8> api, in bit<48> tstamp,
39                   out bit<8> res) {
40      sbf_md_t sbf_md; /* temporary variables */
41      Register<bit<18>, bit<1>>(1, 0) reg_clear_index;
42      Register<pair<bit<16>, bit<16>>, bit<1>>(1, {0, 0}) reg_timer;
43      Pane() pane_1; Pane() pane_2; Pane() pane_3; Pane() pane_4;
```

The control block for an SBF has three in parameters, key, api, and tstamp, which represent the key, the operation, and the timestamp, respectively. The operation can take value from INSERT, QUERY and CLEAR. All the operations involve serving the clearing duty, and the CLEAR operation does not do anything else. The out parameter res returns the query result. Line 40 defines a struct to use as temporary variables. Lines 41, 42 & 43 define the stateful objects that an SBF maintains. Among them, reg_clear_index is an 18-bit integer that increments by one in each step and wraps back to 0 after reaching $2^{18} - 1$. It determines the position to be cleared in each row of the clearing pane. reg_timer maintains the time information according to the timestamp of each packet generated by the switch, in nanoseconds, to control which panes are being used and when to enter a new time pane. pane_1, ..., pane_4 are 4 instances of Pane to implement the panes of the SBF.

```
68      apply {
69          act_hash_index_1(); /* caculate hash functions */
70          act_hash_index_2();
71          act_hash_index_3();
72          tbl_clear_index.apply(); /* increment clear index */
73          tbl_timer.apply(); /* update timer */
74          tbl_set_panes.apply(); /* generate operation for each pane */
75          pane_1.apply(sbf_md.pane_1); /* invoke each pane */
76          pane_2.apply(sbf_md.pane_2);
77          pane_3.apply(sbf_md.pane_3);
78          pane_4.apply(sbf_md.pane_4);
79          tbl_merge_panes.apply(); /* merge query results of panes */
80      }
81  }
```

We temporarily skip some lines of code and show the apply block of SBFilter first, which is the main body of SBFilter. It first calculates the three hash functions used by the SBF, followed by calling tbl_clear_index to read and increment reg_clear_index. Lines 73 & 74 update reg_timer and use it to generate an operation for each pane, respectively. This part involves some tricks to efficiently maintain the timer, which

will be explained below. Then it invokes each pane to process their operation, followed by merging the query result using a match-action table tbl_merge_panes.

```
44    RegisterAction<...>(reg_timer) regact_timer_signal_0 = {
45        void apply(inout pair<bit<16>, bit<16>> val, out bit<16> rv) {
46            if (val.lo != 0) { /* increase hi by 1 when setting lo back to 0 */
47                if (val.hi == 4*PT-1) /* test if wrapping back to 0 */
48                { val.lo = 0; val.hi = 0; }
49                else
50                { val.lo = 0; val.hi = (val.hi + 1); }
51            }
52            else
53            { val.lo = val.lo; val.hi = val.hi; }
54            rv = val.hi;
55        }
56    };
57    RegisterAction<...>(reg_timer) regact_timer_signal_1 = { ... }
58    table tbl_timer { ... } /* select the regact based on the 21st bit of tstamp */
59    table tbl_set_panes { /* assign operations to panes based on timer */
60        const entries = {
61            (INSERT, 0*PT .. 1*PT-1) : set_api(CLEAR, NOOP, NOOP, INSERT);
62            (INSERT, 1*PT .. 2*PT-1) : set_api(INSERT, CLEAR, NOOP, NOOP);
63            (INSERT, 2*PT .. 3*PT-1) : set_api(NOOP, INSERT, CLEAR, NOOP);
64            (INSERT, 3*PT .. 4*PT-1) : set_api(NOOP, NOOP, INSERT, CLEAR);
65            ...
66        }
67    }
```

Lines 44–67 briefly show the tricks of maintaining the timer. In order to measure the time elapsed efficiently, the program counts the number of rounds that the 21^{st} bit (the bit representing 2^{21}) of the timestamp switches from 0 to 1 and back to 0. We assume that the packet flow is dense enough that the timestamps are continuous up to the 21^{st} bit. This ensures the timestamp does not leap without being noticed by the timer. This assumption is easily satisfied by a practical flow, as it only requires one packet in every 2^{21} ns $\approx$ 2 ms. And it can be further ensured using the packet generator. Such a round is called a "tick-tock", so each tick-tock is $2 \cdot 2^{21}$ ns.

Let PT (pane time) be the number of tick-tocks in each time pane. The SBF panes are used circularly, so its period is $k \cdot \mathsf{PT}$ tick-tocks. The register reg_timer keeps track

of the position in this period using two integers, lo and hi, which are the 21st bit of the previous timestamp, and the number of tick-tocks modulo $k \cdot \mathsf{PT}$. Line 44 shows the register action used to update the timer when the 21st bit of the current timestamp is 0. Another register action regact_timer_signal_1 is used when the 21st bit is 1. These two actions are selected by table tbl_timer. Then table tbl_set_panes uses the value of hi to generate operations of panes. This manipulation of timestamps reduces the usage of arithmetic operators and uses more tables. This is a common way to efficiently utilize resources in P4 programming. But it increases the complexity of verification.

5.2 Verification organization

To provide navigation for the remaining of this chapter, we begin by presenting an overview of the the whole verification process in Figure 5.3. We first compiled the P4 source program into P4light AST using our front end (Section 2.1). We then generated the global environment using the instantiation phase program. The rest of the verification process is separated in two parts shown in gray regions: verification of the sliding-window Bloom filter (SBF) and verification of the firewall (FW) as a client of the SBF. We will illustrate them in detail in Sections 5.3 & 5.4, respectively. The verification of the SBF concerns the implementation of the SBF, in particular, the SBF instance and its subordinate objects. The verification of the firewall concerns the client code that calls the SBF, the model of other programmable blocks (parsers and deparsers), and the Tofino switch model.

We design a narrow interface between these two parts, which only consists of the interface of the abstract functional model of SBF (but not the implementation) and two proof blocks: (1) the Verifiable P4 proof of that the SBF P4 program satisfies the abstract function specification, and (2) the proof of that the abstract SBF model satisfies the axiomatic interface of SBF. The statements of the latter are shown in

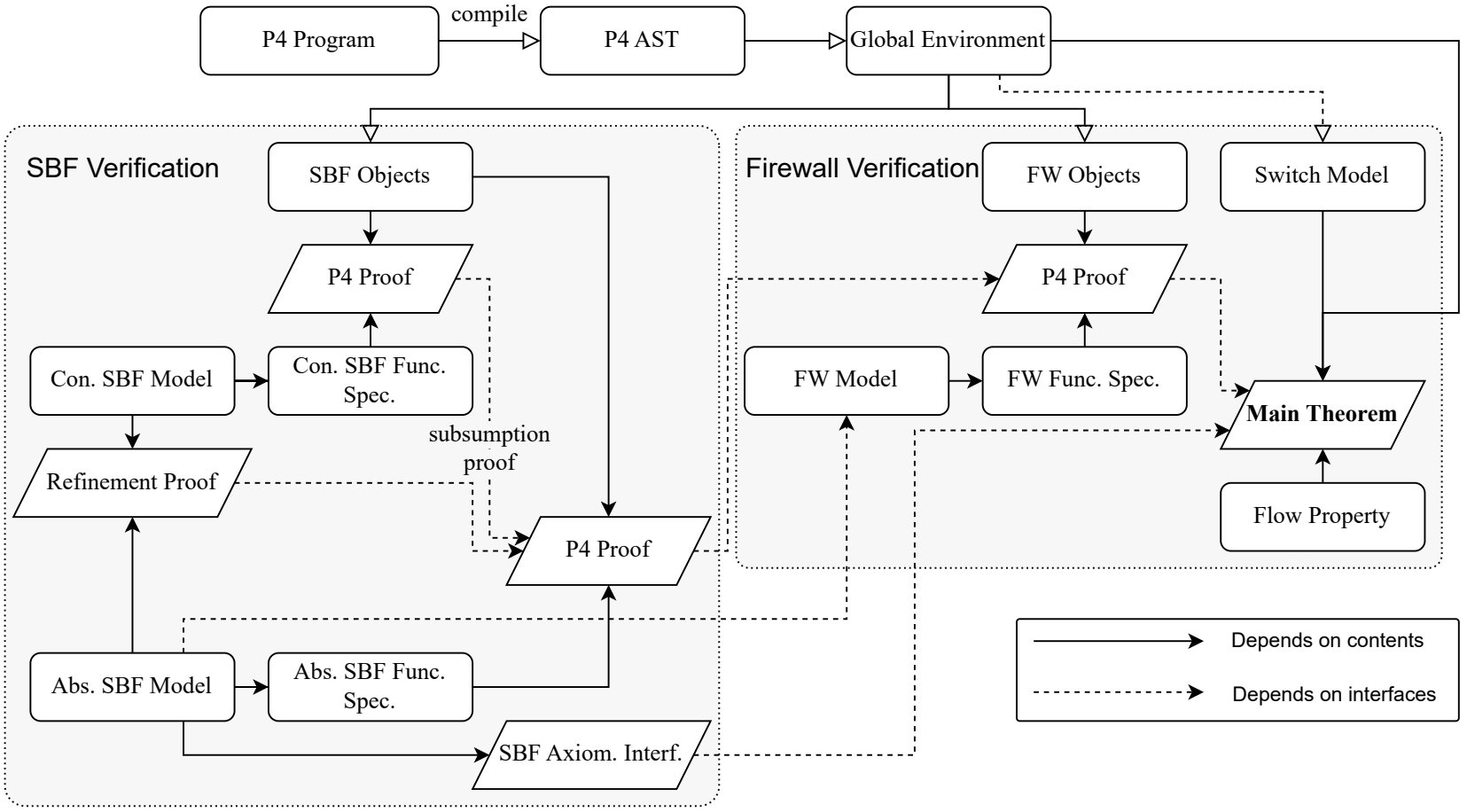

Figure 5.3: Overview. Rounded rectangles represent blocks of definitions; parallelograms represent blocks of proofs. A solid line indicates that a block depends on the *implementation* of another block; a dashed line indicates that a block depends only on the *interface* of another block. A proof block usually establishes the relationship between two definition block and therefore depends on these two blocks. The proof body is irrelevant to its usage, so proof blocks are always depended with dashed lines.

Section 5.2.1. This narrow interface allows modular verification: a data structure and its client are verified separately; the data structure proofs can be used for a different client, and the client proofs can be plugged with a different data structure implementation that satisfies the interface.

One verification methodology used for SBF is layer refinement, which is similar to the technique in CertiKOS [21] and verified FEC [11]. We implement two layers of functional model, called "concrete functional model" and "abstract functional model." The concrete functional model is defined closely related to the P4 program, so it is convenient to prove that the P4 program implements the functional model. However, the shortcoming of the concrete functional model is that it is too much involved with the tricks used in the P4 program, making it more difficult to prove desired properties about the functional model. So we define the abstract functional model of SBF in a more elegant way. We prove the refinement relation between these two functional models, and then prove the abstract function specification through subsumption of specifications. We then prove the axioms in the axiomatic interface, with the benefit of the abstract functional model.

The next part is the verification of the firewall. We first prove P4 program of the firewall as a client of the SBF. For the behavior of the switch, we focus on the part related to the firewall build a switch model that assumes the behavior of other components (parser and deparser). With the switch model, we prove that the stateful firewall guarantees "no-false-negative," in other words, the response packets to internal hosts are always allowed by the stateful firewall.

5.2.1 Axiomatic interface of SBF

Figure 5.4 shows the axioms to characterize SBF. There are three operations, SBFinsert, SBFquery, and SBFclear, defined in the abstract SBF model. SBFinsert

If state f is OK until (at least) deadline t, and if you insert IP address h at time t and then look up h at time t' no more than T seconds later, it will be present.
QueryInsertSame : ∀ f t t' h, ok_until f t → t ≤ t' ≤ t+T →
SBFquery (SBFinsert f (t, h)) (t', h) = Some true.

If you could find IP address h' in the state, and you insert (perhaps different) IP address h, then h' is still in there.
QueryInsertOther : ∀ f t t' h h', SBFquery f (t', h') = Some true →
ok_until f t → t ≤ t' → SBFquery (SBFinsert f (t, h)) (t', h') = Some true.

Doing a clear-step won't affect any query results.
QueryClear : ∀ f t t' h, ok_until f t → t ≤ t' →
SBFquery f (t', h) = SBFquery (SBFclear f t) (t', h).

If state f is OK until deadline t, you can extend its deadline by up to 100 microseconds (T/C) by [insert].
OkInsert : ∀ f t t' h, ok_until f t → t ≤ t' ≤ t+T/C →
ok_until (SBFinsert f (t,h)) t'.

If state f is OK until deadline t, you can extend its deadline by up to 100 microseconds (T/C) by [clear].
OkClear : ∀ f t t', ok_until f t → t ≤ t' ≤ t+T/C → ok_until (SBFclear f t) t'.

The initial state is OK until its preset deadline.
OkEmpty : ∀ t, ok_until (SBFempty t) t.

Figure 5.4: Axiomatic interface of SBF

inserts a record to the SBF with a timestamp, and serves the clear duty of one slot. SBFclear only serves the clear duty of one slot. SBFquery queries the SBF with a timestamp without modification. In order to characterize the requirement of serving the clear duty, we add a predicate ok_until f t to test whether f is in good shape without clearing before t. The meaning of the axioms is explained in the figure. These axioms are enough to prove the "no-false-negative" property. (see Section 5.4)

5.3 Verification of sliding-window Bloom filter

This section shows the verification process of the SBF module using Coq and Verifiable P4.

5.3.1 Concrete functional model

The first step of the verification is defining a functional model of the P4 program in the Coq proof assistant. Because we are going to define a more abstract functional model as an intermediate step of the proof, we name this as the *concrete* functional model, as it is closer to the original program.

```
1  Parameter (num_slots num_rows num_panes pane_tick_tocks : Z).
2  Let cycle_tick_tocks := pane_tick_tocks * num_panes.
3
4  (* a.[i] and a.[i := x] are get and set for lists, respectively. *)
5  (* map2 applies a binary function on each pair of
6     corresponding elements of two lists and results in a single list. *)
7  (* fold_andb is logical "and" over a list of Booleans. *)
8
9  Definition row := list bool.
10
11 Definition row_insert (r : row) (i : Z) : row := r.[i := true].
12
13 Definition pane := list row.
14
15 Definition pane_insert (f : pane) (is : list Z) : pane :=
16   map2 row_insert f is.
```

The code above is the first part of the concrete functional model. At the beginning, the dimension and time constants of the SBF are defined as parameters in the functional model. These constants are hardcoded in the P4 program, as the P4 language is not flexible enough to make all of them parameters. With these constants parametrized, all the proofs about the functional model itself are for any parameters.

And we expect to be able to replay the same Verifiable P4 proof scripts, which connects the program with the concrete functional model, for programs with different values of the constants.

The functional model is layered in the same way as the P4 program. Types row and pane represent the persistent states of rows and panes, respectively. The function row_insert defines the operation of inserting an element into a row, with the element is represented by its hash value for the row. The function pane_insert defines the operation of inserting an element into a pane, with the element is represented by a list of hash values. The QUERY and CLEAR operations are defined similarly in the functional model.

```
18 Record filter := mk_filter {
19   fil_panes : list pane;
20   fil_clear_index : Z;
21   fil_timer : Z * bool;
22 }.
23
24 Definition update_timer (t : Z * bool) (tick : bool) : Z * bool :=
25   if tick
26   then (fst t, true)
27   else if snd t
28     then if (fst t =? cycle_tick_tocks - 1)
29       then (0, false)
30       else (fst t + 1, false)
31     else t.
32
33 Definition get_clear_pane (t : Z * bool) : Z :=
34   fst t / pane_tick_tocks.
35
36 Definition get_insert_pane (cp : Z) : Z :=
37   if (cp =? 0) then num_panes - 1 else cp - 1.
38
39 Definition filter_insert (f : filter) (tick : bool) (h : list Z) : filter :=
40   ... (* unpack f, calling update_timer, get_clear_pane, get_insert_pane *)
41   let new_panes := panes.[cp := pane_clear panes[cp] (Zrepeat clear_index num_rows)]
42                         .[ip := pane_insert panes[ip] h] in
43   mk_filter new_panes new_clear_index new_timer.
```

Function specification of Row.apply() for the INSERT operation:

```
PATH p
MOD ∗ [p]
WITH (r : row) (i : Z) (_ : 0 ≤ i < num_slots),
  PRE (ARG [P4Bit 8 INSERT; P4Bit index_w i], MEM [ ], EXT [row_repr p r])
  POST (RET Null, ARG [P4Bit 8 1], MEM [ ], EXT [row_repr p (row_insert r i)])
```

Function specification of action act_insert() in Row:

```
PATH p
MOD [["rw"]] [p]
WITH (r : row) (i : Z) (_ : 0 ≤ i < num_slots),
  PRE (ARG [ ], MEM [(["index"], P4Bit index_w i)], EXT [row_repr p r])
  POST (RET Null, ARG [ ], MEM [(["rw"], P4Bit 8 1)],
        EXT [row_repr p (row_insert r i)])
```

Figure 5.5: Function Specifications with concrete functional model

The code above is the SBF layer of the functional model. Type filter represents the persistent state of an SBF. Because the dimension and time constants are parametrized, we use division and modulo operations to define the manipulation of the timer. The function filter_insert defines the operation of inserting an element into an SBF, with the element is represented by a list of hash values. Variables cp and ip represent the indices of the panes to clear and to insert, respectively. The QUERY and CLEAR operations are defined similarly.

5.3.2 Function specifications

The function specifications link the functional model with the P4 program state. For example, Figure 5.5 shows the specification of the control block Row's apply function when the operation is INSERT, and the specification of the action act_insert() are as shown in Figure 5.5. The hierarchical predicate row_repr, as defined in Section 3.1, is used to link the functional model value with the state of extern objects in P4. Because each layer supports different operations, we write a function specification for each operation and prove them separately, instead of organizing different operations

into a single function specification. The specification for other operations and other layers are defined similarly.

5.3.3 Verification of abstract methods

Abstract methods are an important feature in P4, whose semantics we defined in Section 2.2.2. For example, when P4 is compiled to hardware in architectures such as Tofino, there is a significant limitation in the way that the P4 program accesses its state: only one access (to any particular piece of state) per packet. To work around that limitation, P4 allows that single access to be a read-modify-write expressed as a "register action," in which the custom modification is written by the P4 programmer as an abstract method.

In this section, we show how to verify P4 extern objects with abstract methods, by demonstrating Tofino's register actions. Register actions are widely used in stateful programs in Tofino, such as the SBF (see Section 5.1.1). A register action contains an abstract method, apply, that specifies how the register is updated. We have built a systematic way to modularly verify register actions with respect to functional models, and built automatic tactics to facilitate the verification procedure. We hope this approach is general enough for all extern objects with abstract methods.

An abstract method is a code block that does not have access to any variables beyond the parameters passed into the the abstract method. So, in a naive approach, an abstract method can be modeled as a pure function from values to values. For example, when the abstract method has one in parameter and one out parameter, we can model its behavior as a function $f : \mathsf{Val} \to \mathsf{Val}$. Its function specification can be written as below.

```
PATH p, MOD * [ ]
WITH (v : Val),
  PRE (ARG [v], MEM [ ], EXT [ ])
  POST (RET Null, ARG [f v], MEM [ ], EXT [ ])
```

Function specification of abstract method apply in RegisterAction:

PATH p, MOD $*$ [ ]
WITH $(v : A)$ $(H_v : P\ v)$,
  PRE (ARG [$r\ v$], MEM [ ], EXT [ ])
  POST (RET Null, ARG [$r\ (f\ v)$, $g\ v$], MEM [ ], EXT [ ]).

Function specification of method execute in RegisterAction:

PATH p, MOD $*$ [p_{reg}]
WITH (reg : list Val) $(i : \mathsf{Z})$ $(v : A)$ $(H_v : P\ v)$ $(H'_v : \mathsf{reg}[i] = r\ v)$,
  PRE (ARG [P4Bit w i], MEM [ ], EXT [$p_{\mathsf{reg}} \mapsto$ (ObjRegister reg)])
  POST (RET $(g\ v)$, ARG [ ], MEM [ ],
    EXT [$p_{\mathsf{reg}} \mapsto$ (ObjRegister reg[$i := r\ (f\ v)$])]).

Figure 5.6: Function specifications in RegisterAction

However, this form is not convenient to apply in actual verification tasks. Because Val contains all kinds of values, including signed and unsigned integers, structs, headers, etc., the model function f needs to be defined for all the cases, and the function specification needs to be proved for all the cases. But in fact, these parameters are not only typed, but sometimes also have more restrictive constraints from the program design. In order to characterize and utilize those constraints, we allow the parameters to be represented by a custom type A instead of Val. We use a predicate $P : A \to \mathsf{Prop}$ to characterize valid inputs, and use $r : A \to \mathsf{Val}$ to describe the representation of A as a value. This approach gives more convenience and flexibility for treating abstract methods.

For example, consider the abstract method in the register action mentioned in Section 2.2.2. Figure 5.6 first shows the general form of function specification of abstract method apply, where functions r, f and predicate P are as above, and function g relates the input with the out parameter rv for the return value of execute. Our library provides a general specification of execute parametrized by the apply specification. Once the user has proved the abstract method satisfies a specification in this form, the specification of function execute provided in our library can be specialized for the instance. The specialized form is given in the second part of Figure 5.6, where the

parameters A, r, P, f, g are from the apply method of the instance. The specialization procedure is supported by the fully automatic tactic build_execute_body, so the user does not have to handle the details.

Abstract methods are often simple. For abstract methods that do not have constraints P on their parameters and do not have branches in the function body, we have implemented an automatic tactic to infer and prove its function specification. For example, a simple abstract method used in SBF is

```
void apply(inout bit<32> val, out bit<32> rv) {
    rv = val;
    val = (val + 32w1);
}
```

By automatic inference, we get

$$A = Z \qquad P(a) = \mathsf{true} \qquad r(a) = \mathsf{P4Bit}\ 32\ a$$

$$f(a) = a + 1 \qquad g(a) = \mathsf{P4Bit}\ 32\ a.$$

5.3.4 Abstract functional model

The concrete functional model is coarse and close to the P4 implementation, therefore it is not convenient for reasoning about its own properties, such as the axioms shown in Figure 5.4.

```
Record SBFilter_core : Type := {
    t_switch : Z;    t_last : Z;    n_clear : Z;    p̄ : list (list H)
}.

Definition SBFilter = option SBFilter_core.
```

From a mathematician's perspective, the sliding-window Bloom filter should be rather defined as above. The elements in each pane should not be represented by hash tables. Instead, as a mathematical model, we should use the set of elements

to represent a pane. And an SBF should not be represented by physical panes used periodically, but by a list of working panes sorted from the oldest to the newest (denoted by $\overline{p}$). For the clearing pane, we only need to count the number of slots that have been cleared (denoted by n_{clear}). To keep track of time, we use a timestamp of unlimited integer instead of 64-bit integer, and only keep track of two things: the time when we will switch to the next pane (t_{switch}), which is also the upper bound of the current query window, and the last timestamp we have seen (t_{last}). Also, the SBF might fall into a corrupted state if the clear duty is not properly completed, or if the packets are not dense enough for the timer to work. So we define SBF as an option type, and use None to represent the corrupted state. In summary, we define the representation of an SBF as SBFilter as above, where H denotes the type of elements.

```
Definition SBFrefresh (f : SBFilter) (t : Z) : SBFilter :=
    match f with
    | Some (t_switch, t_last, n_clear, p) ⇒
        assert(t ∈ [t_last, t_last + t_tick]);
        if (t > t_switch) then
            assert(n_clear ≤ num_slots);
            let p' := p[1..(num_panes − 1)] ++ [[]] in
            Some ((t_switch + t_pane), t_last, 0, p')
        else
            Some f
    | None ⇒ None
    end.

Definition SBFinsert (f : SBFilter) (t, h) : SBFilter :=
    match SBFrefresh f t with
    | Some (t_switch, t_last, n_clear, p) ⇒
        p' := p.[num_panes − 2 := p.[num_panes − 2] ++ [h]]
        Some (t_switch, t, n_clear + 1, p')
    | None ⇒ None
    end.
```

Then we can define function SBFinsert that inserts an element to the SBF, which is used in the SBF axioms in Figure 5.4. We first define function SBFrefresh that refreshes the SBF up to time t, which includes two parts. The first part is checking

the time interval between the last and the current timestamps such that tracking time by tick is accurate. The second part is, when $t > t_{\mathsf{switch}}$, removing the first pane (i.e. the oldest pane) from $\overline{p}$ and inserting an empty pane (corresponding to the fully-cleared pane) at the end, to form $\overline{p}'$. In SBFrefresh, we use assert to denote checking a property and returning None if it fails.

In function SBFinsert, after using SBFrefresh to refresh the SBF up to date, we insert element h into the latest pane, update t_{last} with t, and increment n_{clear}. The other operations SBFquery and SBFclear are defined similarly using SBFrefresh. The abstract operation SBFquery only returns the query result without modifying the SBF in any ways including serving the clear duty.

5.3.5 Refinement proof

In order to prove that the concrete functional model refines the abstract functional model, we first define a simulation relation R between them. For a concrete SBF f and an abstract SBF f', we say $f \mathcal{R} f'$ if either f' is the corrupted state None, or the following four conditions are satisfied:

1. The working panes of the concrete SBF simulate the working panes in $\overline{p}'$ in the abstract SBF. The correspondence between concrete panes and abstract panes is determined by i_{c}, which is the index of the clearing pane. i_{c} is determined by the timer record in the concrete SBF. The panes in $\overline{p}'$ are compared with $[p_{i_{\mathsf{c}}+1}, \ldots, p_{\mathsf{num\_panes}-1}, p_0, \ldots, p_{i_{\mathsf{c}}-1}]$, respectively, where $p_0, \ldots, p_{\mathsf{num\_panes}-1}$ are panes in the concrete SBF in physical order. The simulation relation between a concrete pane and an abstract pane is that the hash tables in the concrete pane is the same as the result of inserting the elements in the corresponding abstract pane.

2. The clearing pane in the concrete SBF is properly cleared with respect to n_{clear} in the abstract SBF. That includes two cases: If $n_{\mathsf{clear}} \geq \mathsf{num\_slots}$, i.e. the clearing pane is fully cleared, then all slots of the concrete clearing pane are 0; If $n_{\mathsf{clear}} < \mathsf{num\_slots}$, then the n_{clear} slots before the next slot to clear in the concrete are 0, counting from the end if reaching the beginning of the hash table.
3. The timer of the concrete SBF simulates the timer of the abstract SBF, which contains the following two conditions:
   (a) The time to switch panes in the abstract SBF, t_{switch}, is aligned with a tick-tock in the concrete SBF. This is because the concrete SBF keeps track of time by examining a particular bit. Each time the bit flips is called a tick, and a cycle of flipping twice is called a tick-tock. The concrete SBF always switches panes at the end of a tick-tock, and the abstract SBF must align with it. Mathematically, that is $(2t_{\mathsf{tick}})$ divides t_{switch}.
   (b) The time interval between the last timestamp t_{last} and the time to switch panes t_{switch} matches with the concrete SBF. The concrete SBF tracks time by counting the number of ticks with modulus, and switches panes when the number reaches certain points. Let n be the number of ticks from now till pane switching in the concrete SBF. For example, $n = 1$ if the current tick is the last one before pane switching. If t_{last} is at the beginning of the current tick, we have $t_{\mathsf{switch}} - t_{\mathsf{last}} = n \cdot t_{\mathsf{tick}}$. Since t_{last} can be anywhere in the current tick, the condition needs to be satisfied is $t_{\mathsf{switch}} - t_{\mathsf{last}} \in \big((n-1) \cdot t_{\mathsf{tick}}, n \cdot t_{\mathsf{tick}}\big]$.
4. The numbers used in the concrete SBF are in their designated range, respectively: $\mathsf{fil\_clear\_index}$ must fall in $[0, \mathsf{num\_slots})$. The first element of $\mathsf{fil\_timer}$ must fall in $[0, \mathsf{cycle\_tick\_tocks})$. The second element of $\mathsf{fil\_timer}$ must be 0 or 1.

```
PATH p
MOD ∗ [p]
WITH (h : H) (t : Z) (f : SBFilter),
  PRE (ARG [h; P4Bit 8 INSERT; P4Bit 48 t; P4Bit 8 1], MEM [ ],
       EXT [SBFilter_repr p f])
  POST (RET Null, ARG [P4Bit 8 1], MEM [ ],
        EXT [SBFfilter_repr p (filter_insert f (t, h))])
```

Figure 5.7: Function Specifications relating the abstract functional model

This simulation relation allows us to prove that the three operations (insert, query, and clear) preserve the simulation relation. For example, the preservation of the insert operation is as follows.

Lemma 5.1. *For any concrete SBF f, abstract SBF f', timestamp t, and element h, if $f \ \mathcal{R} \ f'$, then inserting (t, h) preserves the simulation relation, i.e.*

$$\big(\mathsf{filter\_insert}\ f\ (\lfloor t/t_{\mathsf{tick}} \rfloor \bmod 2)\ (\overline{hash}(h))\big) \ \ \mathcal{R} \ \ \big(\mathsf{SBFinsert}\ f'\ (t, h)\big).$$

The lemmas for the other two operations, query and clear, are defined similarly. We proved these three lemmas in the Coq proof assistant, using normal interactive proof techniques and an automatic solver reasoning about arrays [44].

We utilize this abstraction in the function specifications of SBFilter to encapsulate SBF's implementation details. As mentioned in Section 3.1, we define hierarchical predicate

$$\mathsf{SBFilter\_repr}(p, f') \ \ := \ \ \exists f.\ \mathsf{filter\_repr}(p, f) \ \wedge \ f \ \mathcal{R} \ f'.$$

to relate the abstract model with P4 state. Then we can define function specification relating P4 program state to the abstract functional model. For example, Figure 5.7 shows the function specification in the case that the operation is INSERT. This specification hides all the implementation details of the SBF. As long as the implementation satisfies the function specification and the SBF axioms, we can prove the client has

desire property. We prove the abstract function specifications through the concrete specifications and the refinement lemma (Lemma 5.1).

Finally, we define the operator ok_until and prove the axioms of SBF (Figure 5.4). For an abstract SBF f and timestamp t, the predicate ok_until indicates whether f is fresh enough if the next operation is at time t. The predicate ok_until is defined as, f and t satisfies ok_until if the following conditions hold:

1. The adjacent operations are close enough, i.e. $0 \leq t - t_{\mathsf{last}} \leq T/C$.
2. The SBF is well-formed. This includes (1) $t_{\mathsf{last}} < t_{\mathsf{switch}}$; (2) $|\overline{p}| = \mathsf{num\_panes} - 1$; (3) If the time interval between each pair of consecutive operations afterwards is not more than (T/C), including the interval between the next operation and t_{last}, then the clear duty will be fulfilled at or before t_{switch}. We write it as the formula

$$\lfloor t_{\mathsf{switch}} - 1 - t_{\mathsf{switch}}/(T/C) \rfloor + n_{\mathsf{clear}} >= \mathsf{num\_slots}.$$

We proved that the abstract functional model satisfies the SBF axioms in Coq.

5.4 Verification of stateful firewall

In the previous section, we verified the implementation of a sliding-window Bloom filter. In this section, we show how we verify the stateful firewall implemented as a client of the sliding-window Bloom filter, to ensure it satisfies the desired flow property.

Recall that the task of this stateful firewall is to filter and block unsolicited traffic from the external network into the internal network, in order to prevent DDoS attack against internal hosts, whose capacity is far less than the switch. We want to show

the "no-false-negative" property: the stateful firewall allows each packet that is the response of a recent outgoing packet. We express this property as

$$\begin{aligned} &p.\mathsf{dir} = \mathsf{in} \wedge h[i].\mathsf{dir} = \mathsf{out} \wedge h[i].\mathsf{dst} = p.\mathsf{src} \\ &\wedge\ h[i].\mathsf{src} = p.\mathsf{dst} \wedge p.t - h[i].t \leq T \end{aligned} \implies r = \mathsf{forward}, \tag{5.1}$$

where h is the history as a list of packets, p is the current packet, and r is the reaction of the switch for the current packet.

5.4.1 Verifying the P4 program of stateful firewall

The stateful firewall is implemented as a control block, based on the sliding-window Bloom filter, which we have verified in Section 5.3. The control block reads the packet's headers parsed by the parser. It determines whether the packet is outgoing (from internal to external) or incoming (from external to internal), by testing whether the source IP address has the prefix of the internal network. For an outgoing packet, it inserts the pair $(p.\mathsf{dst}, p.\mathsf{src})$ into the SBF. For an incoming packet, it queries where the pair $(p.\mathsf{src}, p.\mathsf{dst})$ is in the SBF, and it forwards the packet only if the SBF reports "positive". Otherwise, it drops the packet.

Similar to Section 5.3, we build a functional model of this control block, based on the abstract functional model of SBF. Then we prove the control block satisfies the functional model using our Verifiable P4 framework. This program is much simpler than the SBF, and so is the proof.

5.4.2 Switch model

The stateful firewall runs on a switch of the Tofino architecture, which also takes part in the program's behavior. So we need a model of the switch's behavior to reason about the program. In Section 2.5, we discussed architecture specification.

In this application, instead of adopting our full model of Tofino, we adopt a single-pipeline idealized model for a that axiomatizes the behavior of other programmable components, so we can focus on the SBF and stateful firewall. The other programmable components, the parser and the deparser, have very different nature than the SBF and stateful firewall. The main complexity of the SBF and stateful firewall is the persistent state, while the parser and the deparser are stateless and their complexity is about packet structure. We leave verification of parsers and deparsers as a separate future problem. For example, Leapfrog [19] is a verifier for P4 parsers that we could perhaps integrate with Verifiable P4.

In this idealized switch model, we only consider the IP header of each packet, and treat the rest as payload. In an IP header, we only care about the destination and source addresses. So we represent a packet as $((dst, src), payload)$. The input packet also comes with a timestamp t. The result is represented as either None to indicate the packet is dropped or Some $((dst, src), payload)$ to indicate the forwarded packet. Let Γ be the global environment of the P4 program, p be the path of the control instance, and *func* be the function body of the control's apply block. The behavior of the switch is shown in Figure 5.8. The judgment $\Downarrow_{\mathsf{packet}}$ describes the processing of a single packet. The judgment $\Downarrow_{\mathsf{flow}}$ describes the processing of a flow, i.e. a sequence of packets. For simplicity, we use encode to denote encoding the timestamp and the header into a list of P4 values as arguments. And decode denotes decoding the drop flag and the header from the arguments.

5.4.3 Flow property proof

Given the execution relation of a flow packets, the flow property that we want to prove is described as the following theorem.

Theorem 5.1. *Let the initial state s_{extern} be empty. For any historical packets h, current packet p, assume we have the following conditions:*

$$\frac{\Gamma, p, ([], s_{\mathsf{extern}}) \vdash (func, \mathsf{encode}(t, (dst, src))) \Downarrow ((s'_{\mathsf{local}}, s'_{\mathsf{extern}}), \overline{v_{\mathsf{out}}}, \mathsf{return\ Null}) \quad \mathsf{decode}(\overline{v_{\mathsf{out}}}).\mathsf{drop} = \mathsf{true}}{(s_{\mathsf{extern}}, (t, (dst, src), payload)) \Downarrow_{\mathsf{packet}} (s'_{\mathsf{extern}}, \mathsf{None})}$$

$$\frac{\Gamma, p, ([], s_{\mathsf{extern}}) \vdash (func, \mathsf{encode}(t, (dst, src))) \Downarrow ((s'_{\mathsf{local}}, s'_{\mathsf{extern}}), \overline{v_{\mathsf{out}}}, \mathsf{return\ Null}) \quad \mathsf{decode}(\overline{v_{\mathsf{out}}}).\mathsf{drop} = \mathsf{false}}{(s_{\mathsf{extern}}, (t, (dst, src), payload)) \Downarrow_{\mathsf{packet}} (s'_{\mathsf{extern}}, \mathsf{Some}\ (\mathsf{decode}(\overline{v_{\mathsf{out}}}).\mathsf{hdr}, payload))}$$

$$\frac{}{(s_{\mathsf{extern}}, []) \Downarrow_{\mathsf{flow}} (s_{\mathsf{extern}}, [])}$$

$$\frac{(s_{\mathsf{extern}}, p_0) \Downarrow_{\mathsf{packet}} (s'_{\mathsf{extern}}, r_0) \qquad (s'_{\mathsf{extern}}, \overline{p}) \Downarrow_{\mathsf{flow}} (s''_{\mathsf{extern}}, \overline{r})}{(s_{\mathsf{extern}}, p_0 :: \overline{p}) \Downarrow_{\mathsf{flow}} (s''_{\mathsf{extern}}, r_0 :: \overline{r})}$$

Figure 5.8: Switch model

1. *Let r be the flow processing result of the current packet p, i.e.*

$$(s_{\mathsf{extern}}, h + +[p]) \Downarrow_{\mathsf{flow}} (s'_{\mathsf{extern}}, r' + +[r]).$$

2. *The flow is dense. That means for every i such that $0 \leq i < |h|$,*

$$0 \leq h[i+1].t - h[i].t \leq T/C,$$

where we consider $h[|h|] = p$.

Then we have (5.1).

We proved this theorem in Coq, based the proof of the SBF program, the proof of SBF axioms, firewall program proof, and the switch model. The core of this part of proof is a mathematical proof separate from the P4 program. Lennart Beringer, who was expert in Coq but not very knowledgeable about P4 or Verifiable P4 at the time, completed this mathematical proof in two days. [45]

Chapter 6

Conclusion

The foundation and correctness of programs have long been areas of concern. For the P4 programming language, we built Verifiable P4, which contains the following components. We improved the previous Petr4 operational semantics by introducing the instantiation phase separate from the execution phase. We mechanized the operational semantics in the Coq proof assistant, guided by the $P4_{16}$ Specification (Chapter 2). We defined hierarchical predicates for extern objects, and built a Hoare logic for P4, which we proved sound with respect to the operational semantics (Chapter 3). We built a tactic system using Coq's Ltac language to facilitate the verification process (Chapter 4).

To evaluate the utility of Verifiable P4, we applied Verifiable P4 on a sophisticated stateful program. The program that we verified is a stateful firewall purely implemented on the data plane. It maintains its internal state using a sliding-window Bloom filter, which is a sketching data structure. Using Verifiable P4, we proved its functional property about processing a flow of packets. The proof is fully checked by Coq and modularly reusable.

In conclusion, we have built a mechanized operational semantics and verification system for foundationally and modularly verifying stateful P4 programs. On one

hand, we hope this work introduces a new verification methodology to the P4 community. One the other hand, we anticipate this work will bring more applications for the software verification community.

6.1 Future work

There are several directions for further research in this area.

First, it is beneficial to support more P4 features and more automatic verification. Currently, Verifiable P4 only weakly supports features such as parsers and tables with control plane entries. So verification involving those features requires significant manual manipulation. In order to enhance automation, the approaches to explore include migrating the techniques from other P4 verifiers, and looking for new methods for stateful programs.

Second, we plan to achieve modular verification of instantiation. A single class in P4 may be instantiated to multiple instances. Also, the same class can be instantiated differently in different P4 programs. In the semantics and the program logic, we designed the instantiation phase to characterize this behavior, and allowed the path of the instance to be a variable in a function specification. But in the tactic-based verifier, we still need to support such variables in the verification procedure, so that each class only needs to be verified once and applied to all the instances.

Third, we are interested in modeling and verifying the behavior of the whole switch, not only its P4 match-action pipeline. Since P4 programs only define the programmable components, the rest of the switch is defined by the switch model. In a real switch, such as Tofino [23], the full switch model is rather complicated, with features such as packet recirculation and resubmission, traffic manager, packet generator, and digest to the controller. We plan to build a language to define and verify the behavior of the switch.

Fourth, it is worth considering programming tools on a level higher than P4. Lucid [39] and CatQL [35] are high-level tools that generate P4 programs. We plan to define operational semantics of the languages in these tools, verify the code generators or secure them by compilation validation, and build verification tools for these high-level languages.

Fifth, we would like to explore more applications of our verification system. On one hand, that will produce more secured data plane programs. On the other hand, that will further evaluate the verification system and lead to directions of future improvements.

Appendix A

Programs Omitted in the Main Text

A.1 P4 implementation of sliding-window Bloom filter and stateful firewall

This is the complete version of the program that was excerpted in Section 5.1.1.

```
1  #define NOOP 0
2  #define CLEAR 1
3  #define INSERT 2
4  #define QUERY 3
5  #define INSQUERY 4
6  #define UPDATE 5
7  #define UPDQUERY 6
8  #define DONTCARE 0
9  #define QDEFAULT 0
10
11 #include <core.p4>
12 #include <tna.p4>
13 #include "common/headers.p4"
14 #include "common/util.p4"
15
16
```

```
17 typedef bit<8> api_t;
18
19 typedef bit<16> window_t;
20
21 typedef bit<4> pred_t;
22
23 typedef bit<18> bf2_index_t;
24
25 typedef bit<8> bf2_value_t;
26
27 typedef bit<64> bf2_key_t;
28
29 struct bf2_win_md_t {
30     api_t api;
31     bf2_index_t index_1;
32     bf2_index_t index_2;
33     bf2_index_t index_3;
34     bf2_value_t rw_1;
35     bf2_value_t rw_2;
36     bf2_value_t rw_3;
37 }
38
39 struct bf2_ds_md_t {
40     window_t clear_window;
41     bf2_index_t clear_index_1;
42     bf2_index_t hash_index_1;
43     bf2_index_t hash_index_2;
44     bf2_index_t hash_index_3;
45     bf2_win_md_t win_1;
46     bf2_win_md_t win_2;
47     bf2_win_md_t win_3;
48     bf2_win_md_t win_4;
49 }
50
51 struct metadata_t {
52     bf2_key_t bf2_key;
53     api_t bf2_api;
54     bit<8> solicited;
55 }
56
57 struct window_pair_t {
58     window_t lo;
59     window_t hi;
60 }
61
```

```
62  parser EtherIPTCPUDPParser(packet_in pkt, out header_t hdr) {
63      state start {
64          transition parse_ethernet;
65      }
66      state parse_ethernet {
67          pkt.extract(hdr.ethernet);
68          transition select(hdr.ethernet.ether_type) {
69              ETHERTYPE_IPV4 : parse_ipv4;
70              _ : reject;
71          }
72      }
73      state parse_ipv4 {
74          pkt.extract(hdr.ipv4);
75          transition select(hdr.ipv4.protocol) {
76              IP_PROTOCOLS_TCP : parse_tcp;
77              IP_PROTOCOLS_UDP : parse_udp;
78              _ : accept;
79          }
80      }
81      state parse_tcp {
82          pkt.extract(hdr.tcp);
83          transition accept;
84      }
85      state parse_udp {
86          pkt.extract(hdr.udp);
87          transition accept;
88      }
89  }
90
91  parser SwitchIngressParser(packet_in pkt,
92                             out header_t hdr,
93                             out metadata_t ig_md,
94                             out ingress_intrinsic_metadata_t ig_intr_md) {
95      TofinoIngressParser() tofino_parser;
96      EtherIPTCPUDPParser() layer4_parser;
97      state start {
98          tofino_parser.apply(pkt, ig_intr_md);
99          layer4_parser.apply(pkt, hdr);
100         transition accept;
101     }
102 }
103
104 control Bf2BloomFilterRow(in api_t api,
105                           in bf2_index_t index,
106                           out bf2_value_t rw) {
```

```
107     Register<bf2_value_t, bf2_index_t>(32w262144, 8w0) reg_row;
108     RegisterAction<bf2_value_t, bf2_index_t, bf2_value_t>(reg_row) regact_insert = {
109         void apply(inout bf2_value_t value, out bf2_value_t rv) {
110             value = 8w1;
111             rv = 8w1;
112         }
113     };
114     action act_insert() {
115         rw = regact_insert.execute(index);
116     }
117     RegisterAction<bf2_value_t, bf2_index_t, bf2_value_t>(reg_row) regact_query = {
118         void apply(inout bf2_value_t value, out bf2_value_t rv) {
119             rv = value;
120         }
121     };
122     action act_query() {
123         rw = regact_query.execute(index);
124     }
125     RegisterAction<bf2_value_t, bf2_index_t, bf2_value_t>(reg_row) regact_clear = {
126         void apply(inout bf2_value_t value, out bf2_value_t rv) {
127             value = 8w0;
128             rv = 8w0;
129         }
130     };
131     action act_clear() {
132         rw = regact_clear.execute(index);
133     }
134     table tbl_bloom {
135         key = {
136             api : ternary;
137         }
138         actions = {
139             act_insert();
140             act_query();
141             act_clear();
142             .NoAction();
143         }
144         const entries = {
145             INSERT : act_insert();
146             QUERY : act_query();
147             CLEAR : act_clear();
148             _ : .NoAction();
149         }
150         default_action = .NoAction();
151         size = 4;
```

```
152     }
153     apply {
154         tbl_bloom.apply();
155     }
156 }
157
158 control Bf2BloomFilterWin(inout bf2_win_md_t win_md) {
159     Bf2BloomFilterRow() row_1;
160     Bf2BloomFilterRow() row_2;
161     Bf2BloomFilterRow() row_3;
162     apply {
163         row_1.apply(win_md.api, win_md.index_1, win_md.rw_1);
164         row_2.apply(win_md.api, win_md.index_2, win_md.rw_2);
165         row_3.apply(win_md.api, win_md.index_3, win_md.rw_3);
166     }
167 }
168
169 control Bf2BloomFilter(in bf2_key_t ds_key,
170                        in api_t api,
171                        in bit<48> ingress_mac_tstamp,
172                        inout bf2_value_t query_res) {
173     bf2_ds_md_t ds_md;
174     CRCPolynomial<bit<32>>(32w79764919, true, false, false, 32w0, 32
        w4294967295) poly_idx_1;
175     Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly_idx_1) hash_idx_1;
176     action act_hash_index_1() {
177         ds_md.hash_index_1 = hash_idx_1.get(ds_key)[17:0];
178     }
179     table tbl_hash_index_1 {
180         actions = {
181             act_hash_index_1();
182         }
183         default_action = act_hash_index_1();
184         size = 1;
185     }
186     CRCPolynomial<bit<32>>(32w517762881, true, false, false, 32w0, 32
        w4294967295) poly_idx_2;
187     Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly_idx_2) hash_idx_2;
188     action act_hash_index_2() {
189         ds_md.hash_index_2 = hash_idx_2.get(ds_key)[17:0];
190     }
191     table tbl_hash_index_2 {
192         actions = {
193             act_hash_index_2();
194         }
```

```
195         default_action = act_hash_index_2();
196         size = 1;
197     }
198     CRCPolynomial<bit<32>>(32w2821953579, true, false, false, 32w0, 32
            w4294967295) poly_idx_3;
199     Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly_idx_3) hash_idx_3;
200     action act_hash_index_3() {
201         ds_md.hash_index_3 = hash_idx_3.get(ds_key)[17:0];
202     }
203     table tbl_hash_index_3 {
204         actions = {
205             act_hash_index_3();
206         }
207         default_action = act_hash_index_3();
208         size = 1;
209     }
210     Register<bit<32>, bit<1>>(32w1, 32w0) reg_clear_index;
211     RegisterAction<bit<32>, bit<1>, bit<32>>(reg_clear_index) regact_clear_index
            = {
212         void apply(inout bit<32> val, out bit<32> rv) {
213             rv = val;
214             val = (val + 32w1);
215         }
216     };
217     action act_clear_index() {
218         ds_md.clear_index_1 = regact_clear_index.execute(1w0)[17:0];
219     }
220     table tbl_clear_index {
221         actions = {
222             act_clear_index();
223         }
224         default_action = act_clear_index();
225         size = 1;
226     }
227     Register<window_pair_t, bit<1>>(32w1, {16w0, 16w0}) reg_clear_window;
228     RegisterAction<window_pair_t, bit<1>, window_t>(reg_clear_window)
            regact_clear_window_signal_0 = {
229         void apply(inout window_pair_t val, out window_t rv) {
230             bool flip = (val.lo != 16w0);
231             bool wrap = (val.hi == 16w28135);
232             if (flip)
233             {
234                 if (wrap)
235                 {
236                     val.lo = 16w0;
```

```
237                     val.hi = 16w0;
238                 }
239                 else
240                 {
241                     val.lo = 16w0;
242                     val.hi = (val.hi + 16w1);
243                 }
244             }
245             else
246             {
247                 val.lo = val.lo;
248                 val.hi = val.hi;
249             }
250             rv = val.hi;
251         }
252     };
253     RegisterAction<window_pair_t, bit<1>, window_t>(reg_clear_window)
          regact_clear_window_signal_1 = {
254         void apply(inout window_pair_t val, out window_t rv) {
255             if ((val.lo != 16w1))
256             {
257                 val.lo = 16w1;
258             }
259             rv = val.hi;
260         }
261     };
262     action act_clear_window_signal_0() {
263         ds_md.clear_window = regact_clear_window_signal_0.execute(1w0);
264     }
265     action act_clear_window_signal_1() {
266         ds_md.clear_window = regact_clear_window_signal_1.execute(1w0);
267     }
268     table tbl_clear_window {
269         key = {
270             ingress_mac_tstamp : ternary;
271         }
272         actions = {
273             act_clear_window_signal_0();
274             act_clear_window_signal_1();
275         }
276         const entries = {
277             48w0 &&& 48w2097152 : act_clear_window_signal_0();
278             _ : act_clear_window_signal_1();
279         }
280         default_action = act_clear_window_signal_1();
```

```
281         size = 2;
282     }
283     action act_set_clear_win_1(bit<8> api_1,
284                                bit<8> api_2,
285                                bit<8> api_3,
286                                bit<8> api_4) {
287         ds_md.win_1.index_1 = ds_md.clear_index_1;
288         ds_md.win_1.index_2 = ds_md.clear_index_1;
289         ds_md.win_1.index_3 = ds_md.clear_index_1;
290         ds_md.win_2.index_1 = ds_md.hash_index_1;
291         ds_md.win_2.index_2 = ds_md.hash_index_2;
292         ds_md.win_2.index_3 = ds_md.hash_index_3;
293         ds_md.win_3.index_1 = ds_md.hash_index_1;
294         ds_md.win_3.index_2 = ds_md.hash_index_2;
295         ds_md.win_3.index_3 = ds_md.hash_index_3;
296         ds_md.win_4.index_1 = ds_md.hash_index_1;
297         ds_md.win_4.index_2 = ds_md.hash_index_2;
298         ds_md.win_4.index_3 = ds_md.hash_index_3;
299         ds_md.win_1.api = api_1;
300         ds_md.win_2.api = api_2;
301         ds_md.win_3.api = api_3;
302         ds_md.win_4.api = api_4;
303     }
304     action act_set_clear_win_2(bit<8> api_1,
305                                bit<8> api_2,
306                                bit<8> api_3,
307                                bit<8> api_4) {
308         ds_md.win_1.index_1 = ds_md.hash_index_1;
309         ds_md.win_1.index_2 = ds_md.hash_index_2;
310         ds_md.win_1.index_3 = ds_md.hash_index_3;
311         ds_md.win_2.index_1 = ds_md.clear_index_1;
312         ds_md.win_2.index_2 = ds_md.clear_index_1;
313         ds_md.win_2.index_3 = ds_md.clear_index_1;
314         ds_md.win_3.index_1 = ds_md.hash_index_1;
315         ds_md.win_3.index_2 = ds_md.hash_index_2;
316         ds_md.win_3.index_3 = ds_md.hash_index_3;
317         ds_md.win_4.index_1 = ds_md.hash_index_1;
318         ds_md.win_4.index_2 = ds_md.hash_index_2;
319         ds_md.win_4.index_3 = ds_md.hash_index_3;
320         ds_md.win_1.api = api_1;
321         ds_md.win_2.api = api_2;
322         ds_md.win_3.api = api_3;
323         ds_md.win_4.api = api_4;
324     }
325     action act_set_clear_win_3(bit<8> api_1,
```

```
326                                 bit<8> api_2,
327                                 bit<8> api_3,
328                                 bit<8> api_4) {
329         ds_md.win_1.index_1 = ds_md.hash_index_1;
330         ds_md.win_1.index_2 = ds_md.hash_index_2;
331         ds_md.win_1.index_3 = ds_md.hash_index_3;
332         ds_md.win_2.index_1 = ds_md.hash_index_1;
333         ds_md.win_2.index_2 = ds_md.hash_index_2;
334         ds_md.win_2.index_3 = ds_md.hash_index_3;
335         ds_md.win_3.index_1 = ds_md.clear_index_1;
336         ds_md.win_3.index_2 = ds_md.clear_index_1;
337         ds_md.win_3.index_3 = ds_md.clear_index_1;
338         ds_md.win_4.index_1 = ds_md.hash_index_1;
339         ds_md.win_4.index_2 = ds_md.hash_index_2;
340         ds_md.win_4.index_3 = ds_md.hash_index_3;
341         ds_md.win_1.api = api_1;
342         ds_md.win_2.api = api_2;
343         ds_md.win_3.api = api_3;
344         ds_md.win_4.api = api_4;
345     }
346     action act_set_clear_win_4(bit<8> api_1,
347                                 bit<8> api_2,
348                                 bit<8> api_3,
349                                 bit<8> api_4) {
350         ds_md.win_1.index_1 = ds_md.hash_index_1;
351         ds_md.win_1.index_2 = ds_md.hash_index_2;
352         ds_md.win_1.index_3 = ds_md.hash_index_3;
353         ds_md.win_2.index_1 = ds_md.hash_index_1;
354         ds_md.win_2.index_2 = ds_md.hash_index_2;
355         ds_md.win_2.index_3 = ds_md.hash_index_3;
356         ds_md.win_3.index_1 = ds_md.hash_index_1;
357         ds_md.win_3.index_2 = ds_md.hash_index_2;
358         ds_md.win_3.index_3 = ds_md.hash_index_3;
359         ds_md.win_4.index_1 = ds_md.clear_index_1;
360         ds_md.win_4.index_2 = ds_md.clear_index_1;
361         ds_md.win_4.index_3 = ds_md.clear_index_1;
362         ds_md.win_1.api = api_1;
363         ds_md.win_2.api = api_2;
364         ds_md.win_3.api = api_3;
365         ds_md.win_4.api = api_4;
366     }
367     table tbl_set_win {
368         key = {
369             api : ternary;
370             ds_md.clear_window : range;
```

```
371         }
372         actions = {
373             act_set_clear_win_1();
374             act_set_clear_win_2();
375             act_set_clear_win_3();
376             act_set_clear_win_4();
377             .NoAction();
378         }
379         const entries = {
380             (INSERT, 16w0 .. 16w7033) : act_set_clear_win_1(CLEAR,
381                                                             NOOP,
382                                                             NOOP,
383                                                             INSERT);
384             (INSERT, 16w7034 .. 16w14067) : act_set_clear_win_2(INSERT,
385                                                                 CLEAR,
386                                                                 NOOP,
387                                                                 NOOP);
388             (INSERT, 16w14068 .. 16w21101) : act_set_clear_win_3(NOOP,
389                                                                  INSERT,
390                                                                  CLEAR,
391                                                                  NOOP);
392             (INSERT, 16w21102 .. 16w28135) : act_set_clear_win_4(NOOP,
393                                                                  NOOP,
394                                                                  INSERT,
395                                                                  CLEAR);
396             (QUERY, 16w0 .. 16w7033) : act_set_clear_win_1(CLEAR,
397                                                            QUERY,
398                                                            QUERY,
399                                                            QUERY);
400             (QUERY, 16w7034 .. 16w14067) : act_set_clear_win_2(QUERY,
401                                                                CLEAR,
402                                                                QUERY,
403                                                                QUERY);
404             (QUERY, 16w14068 .. 16w21101) : act_set_clear_win_3(QUERY,
405                                                                 QUERY,
406                                                                 CLEAR,
407                                                                 QUERY);
408             (QUERY, 16w21102 .. 16w28135) : act_set_clear_win_4(QUERY,
409                                                                 QUERY,
410                                                                 QUERY,
411                                                                 CLEAR);
412             (CLEAR, 16w0 .. 16w7033) : act_set_clear_win_1(CLEAR,
413                                                            NOOP,
414                                                            NOOP,
415                                                            NOOP);
```

```
416             (CLEAR, 16w7034 .. 16w14067) : act_set_clear_win_2(NOOP,
417                                                                CLEAR,
418                                                                NOOP,
419                                                                NOOP);
420             (CLEAR, 16w14068 .. 16w21101) : act_set_clear_win_3(NOOP,
421                                                                 NOOP,
422                                                                 CLEAR,
423                                                                 NOOP);
424             (CLEAR, 16w21102 .. 16w28135) : act_set_clear_win_4(NOOP,
425                                                                 NOOP,
426                                                                 NOOP,
427                                                                 CLEAR);
428             (_, _) : .NoAction();
429         }
430         default_action = .NoAction();
431         size = 13;
432     }
433     Bf2BloomFilterWin() win_1;
434     Bf2BloomFilterWin() win_2;
435     Bf2BloomFilterWin() win_3;
436     Bf2BloomFilterWin() win_4;
437     action act_merge_wins() {
438         query_res = 8w1;
439     }
440     action act_merge_default() {
441         query_res = 8w0;
442     }
443     table tbl_merge_wins {
444         key = {
445             api : ternary;
446             ds_md.win_1.rw_1 : ternary;
447             ds_md.win_1.rw_2 : ternary;
448             ds_md.win_1.rw_3 : ternary;
449             ds_md.win_2.rw_1 : ternary;
450             ds_md.win_2.rw_2 : ternary;
451             ds_md.win_2.rw_3 : ternary;
452             ds_md.win_3.rw_1 : ternary;
453             ds_md.win_3.rw_2 : ternary;
454             ds_md.win_3.rw_3 : ternary;
455             ds_md.win_4.rw_1 : ternary;
456             ds_md.win_4.rw_2 : ternary;
457             ds_md.win_4.rw_3 : ternary;
458         }
459         actions = {
460             act_merge_wins();
```

```
461             act_merge_default();
462             .NoAction();
463         }
464         const entries = {
465             (QUERY, 8w1, 8w1, 8w1, _, _, _, _, _, _, _, _, _) : act_merge_wins();
466             (QUERY, _, _, _, 8w1, 8w1, 8w1, _, _, _, _, _, _) : act_merge_wins();
467             (QUERY, _, _, _, _, _, _, 8w1, 8w1, 8w1, _, _, _) : act_merge_wins();
468             (QUERY, _, _, _, _, _, _, _, _, _, 8w1, 8w1, 8w1) : act_merge_wins();
469             (QUERY, _, _, _, _, _, _, _, _, _, _, _, _) : act_merge_default();
470             (_, _, _, _, _, _, _, _, _, _, _, _, _) : .NoAction();
471         }
472         default_action = .NoAction();
473         size = 6;
474     }
475     apply {
476         tbl_hash_index_1.apply();
477         tbl_hash_index_2.apply();
478         tbl_hash_index_3.apply();
479         tbl_clear_index.apply();
480         tbl_clear_window.apply();
481         tbl_set_win.apply();
482         win_1.apply(ds_md.win_1);
483         win_2.apply(ds_md.win_2);
484         win_3.apply(ds_md.win_3);
485         win_4.apply(ds_md.win_4);
486         tbl_merge_wins.apply();
487     }
488 }
489
490 control SwitchIngress(inout header_t hdr,
491                       inout metadata_t ig_md,
492                       in ingress_intrinsic_metadata_t ig_intr_md,
493                       in ingress_intrinsic_metadata_from_parser_t ig_intr_prsr_md,
494                       inout ingress_intrinsic_metadata_for_deparser_t ig_intr_dprsr_md,
495                       inout ingress_intrinsic_metadata_for_tm_t ig_intr_tm_md)
496                       {
497     action act_for_tbl_1_action_0() {
498         ig_md.solicited = 1;
499     }
500     table tbl_for_stmt_1 {
501         actions = {
502             act_for_tbl_1_action_0();
503         }
504         default_action = act_for_tbl_1_action_0();
505         size = 1;
```

```
506     }
507     action bf2_act_set_insert_key(bit<8> api) {
508         ig_md.bf2_api = api;
509         ig_md.bf2_key = (hdr.ipv4.dst_addr ++ hdr.ipv4.src_addr);
510     }
511     action bf2_act_set_query_key(bit<8> api) {
512         ig_md.bf2_api = api;
513         ig_md.bf2_key = (hdr.ipv4.src_addr ++ hdr.ipv4.dst_addr);
514     }
515     action bf2_act_set_clear_key() {
516         ig_md.bf2_api = CLEAR;
517     }
518     table bf2_tbl_set_key {
519         key = {
520             hdr.ipv4.src_addr : ternary;
521         }
522         actions = {
523             bf2_act_set_insert_key();
524             bf2_act_set_query_key();
525             bf2_act_set_clear_key();
526         }
527         const entries = {
528             2154823680 &&& 4294901760 : bf2_act_set_insert_key(INSERT);
529             _ : bf2_act_set_query_key(QUERY);
530         }
531         default_action = bf2_act_set_clear_key();
532         size = 2;
533     }
534     Bf2BloomFilter() bf2_ds;
535     action act_for_tbl_3_action_0() {
536         ig_intr_dprsr_md.drop_ctl = 1;
537     }
538     action act_for_tbl_3_action_1() {
539         ig_intr_dprsr_md.drop_ctl = 0;
540     }
541     table tbl_for_stmt_3 {
542         key = {
543             ig_md.solicited : ternary;
544         }
545         actions = {
546             act_for_tbl_3_action_0();
547             act_for_tbl_3_action_1();
548         }
549         const entries = {
550             0 : act_for_tbl_3_action_0();
```

```
551                 _ : act_for_tbl_3_action_1();
552             }
553             default_action = act_for_tbl_3_action_1();
554             size = 2;
555         }
556         apply {
557             tbl_for_stmt_1.apply();
558             bf2_tbl_set_key.apply();
559             bf2_ds.apply(ig_md.bf2_key,
560                          ig_md.bf2_api,
561                          ig_intr_md.ingress_mac_tstamp,
562                          ig_md.solicited);
563             tbl_for_stmt_3.apply();
564         }
565 }
566
567 control SwitchIngressDeparser(packet_out pkt,
568                               inout header_t hdr,
569                               in metadata_t ig_md,
570                               in ingress_intrinsic_metadata_for_deparser_t
                                      ig_intr_dprsr_md)
571                               {
572     apply {
573         pkt.emit(hdr);
574     }
575 }
576
577 parser SwitchEgressParser(packet_in pkt,
578                           out header_t hdr,
579                           out metadata_t eg_md,
580                           out egress_intrinsic_metadata_t eg_intr_md) {
581     TofinoEgressParser() tofino_parser;
582     EtherIPTCPUDPParser() layer4_parser;
583     state start {
584         tofino_parser.apply(pkt, eg_intr_md);
585         layer4_parser.apply(pkt, hdr);
586         transition accept;
587     }
588 }
589
590 control SwitchEgress(inout header_t hdr,
591                      inout metadata_t eg_md,
592                      in egress_intrinsic_metadata_t eg_intr_md,
593                      in egress_intrinsic_metadata_from_parser_t eg_intr_from_prsr,
```

```
594                         inout egress_intrinsic_metadata_for_deparser_t
                                eg_intr_md_for_dprsr,
595                         inout egress_intrinsic_metadata_for_output_port_t
                                eg_intr_md_for_oport)
596                         {
597     apply { }
598 }
599
600 control SwitchEgressDeparser(packet_out pkt,
601                              inout header_t hdr,
602                              in metadata_t eg_md,
603                              in egress_intrinsic_metadata_for_deparser_t
                                    eg_intr_dprsr_md)
604                              {
605     apply {
606         pkt.emit(hdr);
607     }
608 }
609
610 Pipeline(SwitchIngressParser(), SwitchIngress(), SwitchIngressDeparser(),
        SwitchEgressParser(), SwitchEgress(), SwitchEgressDeparser()) pipe;
611
612 Switch(pipe) main;
```

A.2 Concrete functional model

This is the complete version of the program that was excerpted in Section 5.3.1.

```
1  #define NOOP 0
2  #define CLEAR 1
3  #define INSERT 2
4  #define QUERY 3
5  #define INSQUERY 4
6  #define UPDATE 5
7  #define UPDQUERY 6
8  #define DONTCARE 0
9  #define QDEFAULT 0
10
11 #include <core.p4>
12 #include <tna.p4>
13 #include "common/headers.p4"
```

```
14 #include "common/util.p4"
15
16
17 typedef bit<8> api_t;
18
19 typedef bit<16> window_t;
20
21 typedef bit<4> pred_t;
22
23 typedef bit<18> bf2_index_t;
24
25 typedef bit<8> bf2_value_t;
26
27 typedef bit<64> bf2_key_t;
28
29 struct bf2_win_md_t {
30     api_t api;
31     bf2_index_t index_1;
32     bf2_index_t index_2;
33     bf2_index_t index_3;
34     bf2_value_t rw_1;
35     bf2_value_t rw_2;
36     bf2_value_t rw_3;
37 }
38
39 struct bf2_ds_md_t {
40     window_t clear_window;
41     bf2_index_t clear_index_1;
42     bf2_index_t hash_index_1;
43     bf2_index_t hash_index_2;
44     bf2_index_t hash_index_3;
45     bf2_win_md_t win_1;
46     bf2_win_md_t win_2;
47     bf2_win_md_t win_3;
48     bf2_win_md_t win_4;
49 }
50
51 struct metadata_t {
52     bf2_key_t bf2_key;
53     api_t bf2_api;
54     bit<8> solicited;
55 }
56
57 struct window_pair_t {
58     window_t lo;
```

```
59      window_t hi;
60  }
61
62  parser EtherIPTCPUDPParser(packet_in pkt, out header_t hdr) {
63      state start {
64          transition parse_ethernet;
65      }
66      state parse_ethernet {
67          pkt.extract(hdr.ethernet);
68          transition select(hdr.ethernet.ether_type) {
69              ETHERTYPE_IPV4 : parse_ipv4;
70              _ : reject;
71          }
72      }
73      state parse_ipv4 {
74          pkt.extract(hdr.ipv4);
75          transition select(hdr.ipv4.protocol) {
76              IP_PROTOCOLS_TCP : parse_tcp;
77              IP_PROTOCOLS_UDP : parse_udp;
78              _ : accept;
79          }
80      }
81      state parse_tcp {
82          pkt.extract(hdr.tcp);
83          transition accept;
84      }
85      state parse_udp {
86          pkt.extract(hdr.udp);
87          transition accept;
88      }
89  }
90
91  parser SwitchIngressParser(packet_in pkt,
92                             out header_t hdr,
93                             out metadata_t ig_md,
94                             out ingress_intrinsic_metadata_t ig_intr_md) {
95      TofinoIngressParser() tofino_parser;
96      EtherIPTCPUDPParser() layer4_parser;
97      state start {
98          tofino_parser.apply(pkt, ig_intr_md);
99          layer4_parser.apply(pkt, hdr);
100         transition accept;
101     }
102 }
103
```

```
104 control Bf2BloomFilterRow(in api_t api,
105                           in bf2_index_t index,
106                           out bf2_value_t rw) {
107     Register<bf2_value_t, bf2_index_t>(32w262144, 8w0) reg_row;
108     RegisterAction<bf2_value_t, bf2_index_t, bf2_value_t>(reg_row) regact_insert = {
109         void apply(inout bf2_value_t value, out bf2_value_t rv) {
110             value = 8w1;
111             rv = 8w1;
112         }
113     };
114     action act_insert() {
115         rw = regact_insert.execute(index);
116     }
117     RegisterAction<bf2_value_t, bf2_index_t, bf2_value_t>(reg_row) regact_query = {
118         void apply(inout bf2_value_t value, out bf2_value_t rv) {
119             rv = value;
120         }
121     };
122     action act_query() {
123         rw = regact_query.execute(index);
124     }
125     RegisterAction<bf2_value_t, bf2_index_t, bf2_value_t>(reg_row) regact_clear = {
126         void apply(inout bf2_value_t value, out bf2_value_t rv) {
127             value = 8w0;
128             rv = 8w0;
129         }
130     };
131     action act_clear() {
132         rw = regact_clear.execute(index);
133     }
134     table tbl_bloom {
135         key = {
136             api : ternary;
137         }
138         actions = {
139             act_insert();
140             act_query();
141             act_clear();
142             .NoAction();
143         }
144         const entries = {
145             INSERT : act_insert();
146             QUERY : act_query();
147             CLEAR : act_clear();
148             _ : .NoAction();
```

```
149             }
150             default_action = .NoAction();
151             size = 4;
152         }
153         apply {
154             tbl_bloom.apply();
155         }
156 }
157
158 control Bf2BloomFilterWin(inout bf2_win_md_t win_md) {
159     Bf2BloomFilterRow() row_1;
160     Bf2BloomFilterRow() row_2;
161     Bf2BloomFilterRow() row_3;
162     apply {
163         row_1.apply(win_md.api, win_md.index_1, win_md.rw_1);
164         row_2.apply(win_md.api, win_md.index_2, win_md.rw_2);
165         row_3.apply(win_md.api, win_md.index_3, win_md.rw_3);
166     }
167 }
168
169 control Bf2BloomFilter(in bf2_key_t ds_key,
170                        in api_t api,
171                        in bit<48> ingress_mac_tstamp,
172                        inout bf2_value_t query_res) {
173     bf2_ds_md_t ds_md;
174     CRCPolynomial<bit<32>>(32w79764919, true, false, false, 32w0, 32w4294967295
            ) poly_idx_1;
175     Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly_idx_1) hash_idx_1;
176     action act_hash_index_1() {
177         ds_md.hash_index_1 = hash_idx_1.get(ds_key)[17:0];
178     }
179     table tbl_hash_index_1 {
180         actions = {
181             act_hash_index_1();
182         }
183         default_action = act_hash_index_1();
184         size = 1;
185     }
186     CRCPolynomial<bit<32>>(32w517762881, true, false, false, 32w0, 32
            w4294967295) poly_idx_2;
187     Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly_idx_2) hash_idx_2;
188     action act_hash_index_2() {
189         ds_md.hash_index_2 = hash_idx_2.get(ds_key)[17:0];
190     }
191     table tbl_hash_index_2 {
```

```
192         actions = {
193             act_hash_index_2();
194         }
195         default_action = act_hash_index_2();
196         size = 1;
197     }
198     CRCPolynomial<bit<32>>(32w2821953579, true, false, false, 32w0, 32
            w4294967295) poly_idx_3;
199     Hash<bit<32>>(HashAlgorithm_t.CUSTOM, poly_idx_3) hash_idx_3;
200     action act_hash_index_3() {
201         ds_md.hash_index_3 = hash_idx_3.get(ds_key)[17:0];
202     }
203     table tbl_hash_index_3 {
204         actions = {
205             act_hash_index_3();
206         }
207         default_action = act_hash_index_3();
208         size = 1;
209     }
210     Register<bit<32>, bit<1>>(32w1, 32w0) reg_clear_index;
211     RegisterAction<bit<32>, bit<1>, bit<32>>(reg_clear_index) regact_clear_index =
            {
212         void apply(inout bit<32> val, out bit<32> rv) {
213             rv = val;
214             val = (val + 32w1);
215         }
216     };
217     action act_clear_index() {
218         ds_md.clear_index_1 = regact_clear_index.execute(1w0)[17:0];
219     }
220     table tbl_clear_index {
221         actions = {
222             act_clear_index();
223         }
224         default_action = act_clear_index();
225         size = 1;
226     }
227     Register<window_pair_t, bit<1>>(32w1, {16w0, 16w0}) reg_clear_window;
228     RegisterAction<window_pair_t, bit<1>, window_t>(reg_clear_window)
            regact_clear_window_signal_0 = {
229         void apply(inout window_pair_t val, out window_t rv) {
230             bool flip = (val.lo != 16w0);
231             bool wrap = (val.hi == 16w28135);
232             if (flip)
233             {
```

```
234                 if (wrap)
235                 {
236                     val.lo = 16w0;
237                     val.hi = 16w0;
238                 }
239                 else
240                 {
241                     val.lo = 16w0;
242                     val.hi = (val.hi + 16w1);
243                 }
244             }
245             else
246             {
247                 val.lo = val.lo;
248                 val.hi = val.hi;
249             }
250             rv = val.hi;
251         }
252     };
253     RegisterAction<window_pair_t, bit<1>, window_t>(reg_clear_window)
        regact_clear_window_signal_1 = {
254         void apply(inout window_pair_t val, out window_t rv) {
255             if ((val.lo != 16w1))
256             {
257                 val.lo = 16w1;
258             }
259             rv = val.hi;
260         }
261     };
262     action act_clear_window_signal_0() {
263         ds_md.clear_window = regact_clear_window_signal_0.execute(1w0);
264     }
265     action act_clear_window_signal_1() {
266         ds_md.clear_window = regact_clear_window_signal_1.execute(1w0);
267     }
268     table tbl_clear_window {
269         key = {
270             ingress_mac_tstamp : ternary;
271         }
272         actions = {
273             act_clear_window_signal_0();
274             act_clear_window_signal_1();
275         }
276         const entries = {
277             48w0 &&& 48w2097152 : act_clear_window_signal_0();
```

```
278                 _ : act_clear_window_signal_1();
279             }
280             default_action = act_clear_window_signal_1();
281             size = 2;
282         }
283         action act_set_clear_win_1(bit<8> api_1,
284                                    bit<8> api_2,
285                                    bit<8> api_3,
286                                    bit<8> api_4) {
287             ds_md.win_1.index_1 = ds_md.clear_index_1;
288             ds_md.win_1.index_2 = ds_md.clear_index_1;
289             ds_md.win_1.index_3 = ds_md.clear_index_1;
290             ds_md.win_2.index_1 = ds_md.hash_index_1;
291             ds_md.win_2.index_2 = ds_md.hash_index_2;
292             ds_md.win_2.index_3 = ds_md.hash_index_3;
293             ds_md.win_3.index_1 = ds_md.hash_index_1;
294             ds_md.win_3.index_2 = ds_md.hash_index_2;
295             ds_md.win_3.index_3 = ds_md.hash_index_3;
296             ds_md.win_4.index_1 = ds_md.hash_index_1;
297             ds_md.win_4.index_2 = ds_md.hash_index_2;
298             ds_md.win_4.index_3 = ds_md.hash_index_3;
299             ds_md.win_1.api = api_1;
300             ds_md.win_2.api = api_2;
301             ds_md.win_3.api = api_3;
302             ds_md.win_4.api = api_4;
303         }
304         action act_set_clear_win_2(bit<8> api_1,
305                                    bit<8> api_2,
306                                    bit<8> api_3,
307                                    bit<8> api_4) {
308             ds_md.win_1.index_1 = ds_md.hash_index_1;
309             ds_md.win_1.index_2 = ds_md.hash_index_2;
310             ds_md.win_1.index_3 = ds_md.hash_index_3;
311             ds_md.win_2.index_1 = ds_md.clear_index_1;
312             ds_md.win_2.index_2 = ds_md.clear_index_1;
313             ds_md.win_2.index_3 = ds_md.clear_index_1;
314             ds_md.win_3.index_1 = ds_md.hash_index_1;
315             ds_md.win_3.index_2 = ds_md.hash_index_2;
316             ds_md.win_3.index_3 = ds_md.hash_index_3;
317             ds_md.win_4.index_1 = ds_md.hash_index_1;
318             ds_md.win_4.index_2 = ds_md.hash_index_2;
319             ds_md.win_4.index_3 = ds_md.hash_index_3;
320             ds_md.win_1.api = api_1;
321             ds_md.win_2.api = api_2;
322             ds_md.win_3.api = api_3;
```

```
323         ds_md.win_4.api = api_4;
324     }
325     action act_set_clear_win_3(bit<8> api_1,
326                                bit<8> api_2,
327                                bit<8> api_3,
328                                bit<8> api_4) {
329         ds_md.win_1.index_1 = ds_md.hash_index_1;
330         ds_md.win_1.index_2 = ds_md.hash_index_2;
331         ds_md.win_1.index_3 = ds_md.hash_index_3;
332         ds_md.win_2.index_1 = ds_md.hash_index_1;
333         ds_md.win_2.index_2 = ds_md.hash_index_2;
334         ds_md.win_2.index_3 = ds_md.hash_index_3;
335         ds_md.win_3.index_1 = ds_md.clear_index_1;
336         ds_md.win_3.index_2 = ds_md.clear_index_1;
337         ds_md.win_3.index_3 = ds_md.clear_index_1;
338         ds_md.win_4.index_1 = ds_md.hash_index_1;
339         ds_md.win_4.index_2 = ds_md.hash_index_2;
340         ds_md.win_4.index_3 = ds_md.hash_index_3;
341         ds_md.win_1.api = api_1;
342         ds_md.win_2.api = api_2;
343         ds_md.win_3.api = api_3;
344         ds_md.win_4.api = api_4;
345     }
346     action act_set_clear_win_4(bit<8> api_1,
347                                bit<8> api_2,
348                                bit<8> api_3,
349                                bit<8> api_4) {
350         ds_md.win_1.index_1 = ds_md.hash_index_1;
351         ds_md.win_1.index_2 = ds_md.hash_index_2;
352         ds_md.win_1.index_3 = ds_md.hash_index_3;
353         ds_md.win_2.index_1 = ds_md.hash_index_1;
354         ds_md.win_2.index_2 = ds_md.hash_index_2;
355         ds_md.win_2.index_3 = ds_md.hash_index_3;
356         ds_md.win_3.index_1 = ds_md.hash_index_1;
357         ds_md.win_3.index_2 = ds_md.hash_index_2;
358         ds_md.win_3.index_3 = ds_md.hash_index_3;
359         ds_md.win_4.index_1 = ds_md.clear_index_1;
360         ds_md.win_4.index_2 = ds_md.clear_index_1;
361         ds_md.win_4.index_3 = ds_md.clear_index_1;
362         ds_md.win_1.api = api_1;
363         ds_md.win_2.api = api_2;
364         ds_md.win_3.api = api_3;
365         ds_md.win_4.api = api_4;
366     }
367     table tbl_set_win {
```

```
368         key = {
369             api : ternary;
370             ds_md.clear_window : range;
371         }
372         actions = {
373             act_set_clear_win_1();
374             act_set_clear_win_2();
375             act_set_clear_win_3();
376             act_set_clear_win_4();
377             .NoAction();
378         }
379         const entries = {
380             (INSERT, 16w0 .. 16w7033) : act_set_clear_win_1(CLEAR,
381                                                             NOOP,
382                                                             NOOP,
383                                                             INSERT);
384             (INSERT, 16w7034 .. 16w14067) : act_set_clear_win_2(INSERT,
385                                                                 CLEAR,
386                                                                 NOOP,
387                                                                 NOOP);
388             (INSERT, 16w14068 .. 16w21101) : act_set_clear_win_3(NOOP,
389                                                                  INSERT,
390                                                                  CLEAR,
391                                                                  NOOP);
392             (INSERT, 16w21102 .. 16w28135) : act_set_clear_win_4(NOOP,
393                                                                  NOOP,
394                                                                  INSERT,
395                                                                  CLEAR);
396             (QUERY, 16w0 .. 16w7033) : act_set_clear_win_1(CLEAR,
397                                                            QUERY,
398                                                            QUERY,
399                                                            QUERY);
400             (QUERY, 16w7034 .. 16w14067) : act_set_clear_win_2(QUERY,
401                                                                CLEAR,
402                                                                QUERY,
403                                                                QUERY);
404             (QUERY, 16w14068 .. 16w21101) : act_set_clear_win_3(QUERY,
405                                                                 QUERY,
406                                                                 CLEAR,
407                                                                 QUERY);
408             (QUERY, 16w21102 .. 16w28135) : act_set_clear_win_4(QUERY,
409                                                                 QUERY,
410                                                                 QUERY,
411                                                                 CLEAR);
412             (CLEAR, 16w0 .. 16w7033) : act_set_clear_win_1(CLEAR,
```

```
                                                                NOOP,
                                                                NOOP,
                                                                NOOP);
            (CLEAR, 16w7034 .. 16w14067) : act_set_clear_win_2(NOOP,
                                                                CLEAR,
                                                                NOOP,
                                                                NOOP);
            (CLEAR, 16w14068 .. 16w21101) : act_set_clear_win_3(NOOP,
                                                                NOOP,
                                                                CLEAR,
                                                                NOOP);
            (CLEAR, 16w21102 .. 16w28135) : act_set_clear_win_4(NOOP,
                                                                NOOP,
                                                                NOOP,
                                                                CLEAR);
            (_, _) : .NoAction();
        }
        default_action = .NoAction();
        size = 13;
    }
    Bf2BloomFilterWin() win_1;
    Bf2BloomFilterWin() win_2;
    Bf2BloomFilterWin() win_3;
    Bf2BloomFilterWin() win_4;
    action act_merge_wins() {
        query_res = 8w1;
    }
    action act_merge_default() {
        query_res = 8w0;
    }
    table tbl_merge_wins {
        key = {
            api : ternary;
            ds_md.win_1.rw_1 : ternary;
            ds_md.win_1.rw_2 : ternary;
            ds_md.win_1.rw_3 : ternary;
            ds_md.win_2.rw_1 : ternary;
            ds_md.win_2.rw_2 : ternary;
            ds_md.win_2.rw_3 : ternary;
            ds_md.win_3.rw_1 : ternary;
            ds_md.win_3.rw_2 : ternary;
            ds_md.win_3.rw_3 : ternary;
            ds_md.win_4.rw_1 : ternary;
            ds_md.win_4.rw_2 : ternary;
            ds_md.win_4.rw_3 : ternary;
```

```
458             }
459             actions = {
460                 act_merge_wins();
461                 act_merge_default();
462                 .NoAction();
463             }
464             const entries = {
465                 (QUERY, 8w1, 8w1, 8w1, _, _, _, _, _, _, _, _, _) : act_merge_wins();
466                 (QUERY, _, _, _, 8w1, 8w1, 8w1, _, _, _, _, _, _) : act_merge_wins();
467                 (QUERY, _, _, _, _, _, _, 8w1, 8w1, 8w1, _, _, _) : act_merge_wins();
468                 (QUERY, _, _, _, _, _, _, _, _, _, 8w1, 8w1, 8w1) : act_merge_wins();
469                 (QUERY, _, _, _, _, _, _, _, _, _, _, _, _) : act_merge_default();
470                 (_, _, _, _, _, _, _, _, _, _, _, _, _) : .NoAction();
471             }
472             default_action = .NoAction();
473             size = 6;
474         }
475         apply {
476             tbl_hash_index_1.apply();
477             tbl_hash_index_2.apply();
478             tbl_hash_index_3.apply();
479             tbl_clear_index.apply();
480             tbl_clear_window.apply();
481             tbl_set_win.apply();
482             win_1.apply(ds_md.win_1);
483             win_2.apply(ds_md.win_2);
484             win_3.apply(ds_md.win_3);
485             win_4.apply(ds_md.win_4);
486             tbl_merge_wins.apply();
487         }
488 }
489
490 control SwitchIngress(inout header_t hdr,
491                       inout metadata_t ig_md,
492                       in ingress_intrinsic_metadata_t ig_intr_md,
493                       in ingress_intrinsic_metadata_from_parser_t ig_intr_prsr_md,
494                       inout ingress_intrinsic_metadata_for_deparser_t ig_intr_dprsr_md,
495                       inout ingress_intrinsic_metadata_for_tm_t ig_intr_tm_md)
496                       {
497     action act_for_tbl_1_action_0() {
498         ig_md.solicited = 1;
499     }
500     table tbl_for_stmt_1 {
501         actions = {
502             act_for_tbl_1_action_0();
```

```
503         }
504         default_action = act_for_tbl_1_action_0();
505         size = 1;
506     }
507     action bf2_act_set_insert_key(bit<8> api) {
508         ig_md.bf2_api = api;
509         ig_md.bf2_key = (hdr.ipv4.dst_addr ++ hdr.ipv4.src_addr);
510     }
511     action bf2_act_set_query_key(bit<8> api) {
512         ig_md.bf2_api = api;
513         ig_md.bf2_key = (hdr.ipv4.src_addr ++ hdr.ipv4.dst_addr);
514     }
515     action bf2_act_set_clear_key() {
516         ig_md.bf2_api = CLEAR;
517     }
518     table bf2_tbl_set_key {
519         key = {
520             hdr.ipv4.src_addr : ternary;
521         }
522         actions = {
523             bf2_act_set_insert_key();
524             bf2_act_set_query_key();
525             bf2_act_set_clear_key();
526         }
527         const entries = {
528             2154823680 &&& 4294901760 : bf2_act_set_insert_key(INSERT);
529             _ : bf2_act_set_query_key(QUERY);
530         }
531         default_action = bf2_act_set_clear_key();
532         size = 2;
533     }
534     Bf2BloomFilter() bf2_ds;
535     action act_for_tbl_3_action_0() {
536         ig_intr_dprsr_md.drop_ctl = 1;
537     }
538     action act_for_tbl_3_action_1() {
539         ig_intr_dprsr_md.drop_ctl = 0;
540     }
541     table tbl_for_stmt_3 {
542         key = {
543             ig_md.solicited : ternary;
544         }
545         actions = {
546             act_for_tbl_3_action_0();
547             act_for_tbl_3_action_1();
```

```
548         }
549         const entries = {
550             0 : act_for_tbl_3_action_0();
551             _ : act_for_tbl_3_action_1();
552         }
553         default_action = act_for_tbl_3_action_1();
554         size = 2;
555     }
556     apply {
557         tbl_for_stmt_1.apply();
558         bf2_tbl_set_key.apply();
559         bf2_ds.apply(ig_md.bf2_key,
560                      ig_md.bf2_api,
561                      ig_intr_md.ingress_mac_tstamp,
562                      ig_md.solicited);
563         tbl_for_stmt_3.apply();
564     }
565 }
566
567 control SwitchIngressDeparser(packet_out pkt,
568                               inout header_t hdr,
569                               in metadata_t ig_md,
570                               in ingress_intrinsic_metadata_for_deparser_t
                                      ig_intr_dprsr_md)
571                               {
572     apply {
573         pkt.emit(hdr);
574     }
575 }
576
577 parser SwitchEgressParser(packet_in pkt,
578                           out header_t hdr,
579                           out metadata_t eg_md,
580                           out egress_intrinsic_metadata_t eg_intr_md) {
581     TofinoEgressParser() tofino_parser;
582     EtherIPTCPUDPParser() layer4_parser;
583     state start {
584         tofino_parser.apply(pkt, eg_intr_md);
585         layer4_parser.apply(pkt, hdr);
586         transition accept;
587     }
588 }
589
590 control SwitchEgress(inout header_t hdr,
591                      inout metadata_t eg_md,
```

```
592                         in egress_intrinsic_metadata_t eg_intr_md,
593                         in egress_intrinsic_metadata_from_parser_t eg_intr_from_prsr,
594                         inout egress_intrinsic_metadata_for_deparser_t
                                eg_intr_md_for_dprsr,
595                         inout egress_intrinsic_metadata_for_output_port_t
                                eg_intr_md_for_oport)
596                         {
597     apply { }
598 }
599
600 control SwitchEgressDeparser(packet_out pkt,
601                              inout header_t hdr,
602                              in metadata_t eg_md,
603                              in egress_intrinsic_metadata_for_deparser_t
                                   eg_intr_dprsr_md)
604                              {
605     apply {
606         pkt.emit(hdr);
607     }
608 }
609
610 Pipeline(SwitchIngressParser(), SwitchIngress(), SwitchIngressDeparser(),
        SwitchEgressParser(), SwitchEgress(), SwitchEgressDeparser()) pipe;
611
612 Switch(pipe) main;
```

Bibliography

[1] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program logics for certified compilers*. Cambridge University Press, 2014.

[2] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[3] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[7] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61:367–422, 2018.

[8] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Programming Languages and Systems: 9th European Symposium on Programming*, pages 56–71. Springer, 2000.

[9] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual Conference of the ACM Special Interest Group*

on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 226–239, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.

[11] Joshua M Cohen, Qinshi Wang, and Andrew W. Appel. Verified erasure correction in Coq with MathComp and VST. In *Computer Aided Verification: 34th International Conference, CAV 2022*, pages 272–292. Springer, 2022.

[12] The P4 Language Consortium. $P4_{16}$ language specification, version 1.2.3. Technical report, July 2022.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, 1977.

[14] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Software Engineering and Formal Methods: 10th International Conference, SEFM 2012*, pages 233–247. Springer, 2012.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] David Delahaye. A tactic language for the system Coq. In *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, pages 85–95. Springer, 2000.

[17] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[18] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. Petr4: formal foundations for P4 data planes. *Proceedings of the ACM on Programming Languages*, 5(POPL), 2021.

[19] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. Leapfrog: certified equivalence for protocol parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 950–965, 2022.

[20] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. Dependently-typed data plane programming. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022.

[21] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. *ACM SIGPLAN Notices*, 50(1):595–608, 2015.

[22] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561, 2023.

[23] Intel. Intel® Tofino™ programmable ethernet switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Accessed: 2023-01-18.

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Thomas Kleymann. *Hoare logic and VDM: Machine-checked soundness and completeness proofs*. PhD thesis, University of Edinburgh, 1998.

[26] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16*, pages 348–370. Springer, 2010.

[27] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 311–324, USA, 2016. USENIX Association.

[28] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. p4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, pages 490–503, 2018.

[29] David MacQueen. An implementation of standard ML modules. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 212–223, New York, NY, USA, 1988. Association for Computing Machinery.

[30] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling efficient sketch-based monitoring on programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 743–759, Renton, WA, April 2022. USENIX Association.

[32] Moni Naor and Eylon Yogev. Tight bounds for sliding bloom filters. *Algorithmica*, 73(4):652–672, 2015.

[33] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. Verification of P4 programs in feasible time using assertions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 73–85, 2018.

[34] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional, 2003. pages 30-33.

[35] Mengying Pan. CtrlApp: A query-based language for control applications on data plane. https://github.com/MollyDream/CtrlApp.

[36] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Electronic textbook, 2022. Version 6.2, http://softwarefoundations.cis.upenn.edu.

[37] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2022. Version 6.2, http://softwarefoundations.cis.upenn.edu.

[38] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 683–699, 2020.

[39] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 731–747, 2021.

[40] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 518–532, 2018.

[41] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 314–327, 2016.

[42] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Chen Tian, and Minlan Yu. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 17–32, 2021.

[43] Verified Network Toolchain. petr4/coq/lib/P4light at Feb2023. https://github.com/verified-network-toolchain/petr4/tree/Feb2023/coq/lib/P4light. Accessed: 2023-03-07.

[44] Qinshi Wang and Andrew W. Appel. A solver for arrays with concatenation. *Journal of Automated Reasoning*, 67(1):4, 2023.

[45] Qinshi Wang, Mengying Pan, Shengyi Wang, Ryan Doenges, Lennart Beringer, and Andrew W. Appel. Foundational verification of stateful P4 packet processing. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*, 2023.

[46] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 126–138, New York, NY, USA, 2020. Association for Computing Machinery.