Foundational Verification of Stateful P4 Packet Processing

Qinshi Wang
Princeton University, USA

Mengying Pan
Princeton University, USA

Shengyi Wang
Princeton University, USA

Ryan Doenges
Cornell University, USA

Lennart Beringer
Princeton University, USA

Andrew W. Appel
Princeton University, USA

Abstract

P4 is a standardized programming language for the network data plane. But P4 is not just for routing anymore. As programmable switches support stateful objects, P4 programs move beyond just stateless forwarders into new stateful applications: network telemetry (heavy hitters, DDoS detection, performance monitoring), middleboxes (firewalls, NAT, load balancers, intrusion detection), and distributed services (in-network caching, lock management, conflict detection). The complexity of stateful programs and their richer specifications are beyond what existing P4 program verifiers can handle.

Verifiable P4 is a new interactive verification framework for P4 that (1) allows reasoning about multi-packet properties by specifying the per-packet relation between initial and final states; (2) performs modular verification, especially providing a modular description for stateful objects; (3) is foundational, i.e., with a machine-checked soundness proof with respect to a formal operational semantics of $P4_{16}$ (the current specification of P4) in Coq. In addition, our framework includes a proved-correct reference interpreter.

We demonstrate the framework with the specification and verification of a stateful firewall that uses a sliding-window Bloom filter on a Tofino switch to block (most) unsolicited traffic.

2012 ACM Subject Classification Security and privacy → Logic and verification

Keywords and phrases Software Defined Networking, Verifiable P4, Stateful data plane programming

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding This material is based upon work supported DARPA Contracts HR001120C0160 and HR001120C0107, and by NSF grant FMiTF-1918396.

1 Introduction

The data plane for software-defined network switches is increasingly programmed in P4 [3]. But P4 is a quirky language, and programs are often contorted to fit within the constraints of a particular target architecture, so the correctness of these programs has become a concern. To address that concern, there are several verification tools for P4 programs (see §7).

In many classic P4 applications, processing a packet does not change the state of the switch. However, recent applications are *stateful* and go far beyond making routing decisions: the processing of a packet may alter the state of registers or result in the

installation of new forwarding rules, and thus affect the processing of following packets. Stateful applications include network telemetry systems (SketchLib [17], BeauCoup [2], FlowRadar [14]), network functions (SilkRoad [16]), and distributed services (NetCache [12], NetLock [26]). Unfortunately, existing tools have limited reasoning capabilities for registers or multi-packet policies.

To begin addressing these shortcomings, we present a foundational framework for specifying and reasoning about data-plane packet processing on a stateful P4 switch. Implemented in the Coq proof assistant, our system facilitates semi-interactive verification of stateful P4 programs and is justified w.r.t. a precise operational semantics of $\mathrm{P4}_{16}$.

As an example for a multi-packet policy, consider a stateful firewall that protects the internal network from unsolicited traffic. External packets may pass through the firewall only if they are responses to recent outgoing requests to the same IP address, modulo a small tolerable rate of false positives. But *no valid incoming responses may be blocked.*

To specify the latter property more precisely, let T be the valid response time window, h be the history of packets processed, p be the current packet, and r the action on p (forward or drop). We want that for every integer i, we have

$$\begin{array}{l} p.\mathrm{dir} = \mathrm{in} \wedge h[i].\mathrm{dir} = \mathrm{out} \wedge h[i].\mathrm{dst} = p.\mathrm{src} \\ \wedge\ h[i].\mathrm{src} = p.\mathrm{dst} \wedge p.t - h[i].t \leq T \end{array} \implies r \neq \mathrm{drop} \tag{1}$$

To maintain the window of length T, a P4 implementation necessarily maintains state; but existing verifiers either do not handle state at all, have specification languages that are too weak to relate the pre-state to the post-state after processing a packet (see §7), or do not permit reasoning about multi-packet properties.

Property 1 holds even for a firewall admits every packet, but we prove the P4 program correct with respect to a functional model, for which properties such as 1 can be derived.

Our verifier connects P4 verification with reasoning about multi-packet policies using assertions that are syntactically constrained but permit reference to arbitrary Coq constructions. The user must equip each P4 function with a specification that asserts adherence to a model-level counterpart, i.e. a Coq function (or proposition) describing its effect in terms of a semantic model of packet processing. We justify the program logic in Coq by a soundness proof w.r.t. the operational semantics.

Our operational semantics builds on Petr4 [5] but is defined as an inductive relation in Coq rather than Petr4's executable Ocaml program. Indeed, part of our effort consisted in understanding aspects related to nondeterminism and partly uninitialized data structures, that are not modeled in Petr4 and are specified partially at best in the P4 manual [3]. We report numerous inaccuracies etc. in said manual later in the paper. We also provide our own interpreter and prove (in Coq) that it exhibits behavior consistent with our semantics, e.g. by resolving determinism in an appropriate manner.

Contributions

1. We present *Verifiable P4*, a system for verifying stateful specifications of P4 data-plane packet processing. Our logic does not cover packet parsing and deparsing, which we leave for future work. We provide automation support for proving that P4 code correctly implements a functional model; users prove interactively that a functional model establishes a high-level property, such as "no valid responses are blocked".
2. 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.*

3. We formalize the operational semantics of P4 (incl. parsing and deparsing) in Coq, and prove soundness of our verifier and of a reference interpreter. Our operational semantics is the first formal specification to include abstract methods (a P4 feature that encodes a P4 program in an extern object), and Verifiable P4 is the first verifier to support them. The development of our semantics uncovered several inaccuracies, bugs, and ambiguities in the $P4_{16}$ reference manual that had also escaped the Petr4 semantics and tools. We contributed bug-fixes as pull requests to the manual and discussed them with the $P4_{16}$ committee.
4. We develop a model-level axiomatization of temporal windows that can serve as a blueprint for other finite-horizon data structures over streams implemented in P4 or other languages. We give a concise specification of a sliding window Bloom filter (SBF) and a high-performance implementation in P4, with an application to a stateful firewall. We show how to connect model-level firewall verification to P4 verification in Coq, and briefly discuss other examples.

The source files associated with this submission are available at
`https://github.com/verified-network-toolchain/VerifiableP4/tree/Feb2023`

2 Example: A stateful firewall

The data-plane of a software-defined network (SDN) switch processes about 10^9 packets per second (or 3200 Gb per second) in each "pipeline" [13]. To accomplish that, each packet is allowed only one trip through a highly pipelined "match-action unit". To program this unconventional model of computation, the P4 language was designed with some inspiration from Verilog. It combines logical with architectural aspects but is reasonably modular—one can divide programs into reusable libraries and the client programs that make use of them.

P4 is a horrible but useful programming language

P4 was designed to support packet routing, so all of its control structures are designed around match-action tables. One generally cannot access the same (persistent) data more than once per packet, so things must be accomplished in a read-modify-write with user-specifiable "modify." Except for the parser component, P4 code does not contain loops, hence every packet must process in n pipeline stages (so at least we can mostly use a big-step semantics!). But P4 is *useful* because it gives a reasonably portable way to program high-performance network switches from different manufacturers.

The main elements of a typical P4 program are as follows.

Parsers extract packet headers and the payload from a packet. Headers – and typical auxiliary data structures throughout the code – are represented as `structs` that can be accessed as in the C language.

Control blocks describe how headers are processed. A control block contains a list of declarations, plus the control flow. Declarations include

- **extern objects,** which are architecture-specific constructs such as registers, cryptographic operations, or other domain-specific functions,
- **actions,** which assign flags (e.g. DROP) or other values (e.g. a specific output port) to suitable header fields,
- **match-action tables,** which match header fields against predefined keys and trigger actions or extern objects accordingly.

The control flow is specified by an imperative program ("apply function") that governs under which conditions or order ("pipeline") the match-action tables are invoked on a header.

Deparsers reasssemble the processed packet headers with the payload and emit the packet to the network.

Modularity of P4 arises from the ability to instantiate registers, other extern objects, and control blocks multiple times, and to define them in a style reminiscent of classes in e.g. Java, with instantiatable parameters.

The low-level complexities of P4 make programs difficult to read and write, so code (including our running example) is increasingly not written directly in P4 but in one of the various front-end languages that target P4 (to synthesize our SBF/firewall we used CatQL, a functional-style language under development by the second author). These experimental languages don't yet have what we would call a formal semantics or even a real reference manual, so we verify the P4 code rather than verify the ultimate source programs.

The difficulty of writing and reading P4 programs is a strong motivation to formally verify P4 with respect to model-level or higher-level specifications, which (even though in Coq) will be more accessible to engineers than the source code. Our tool can verify both reusable libraries – like the Bloom filter control block – and their clients – like the firewall, which is a control block separate from the Bloom filter. Where the library implements nontrivial algorithms, users may need more expertise, but verifying simple clients (even those using sophisticated libraries) is easier.

The Sliding Window Boom Filter and its Firewall Client

We continue with our running example. The stateful firewall must remember recent IP addresses in a data structure that can fit in the switch's persistent registers and can be accessed in constant time within the switch's pipeline constraints. To implement this in a P4 switch *running entirely in the data plane,* we use a succinct constant-time-access data structure: a sliding-window Bloom filter (SBF).

A Bloom filter is a hash table, without collision detection, in which value i is hashed with r different functions and a bit is set to 1 at those indexes $f_1(i), f_2(i), \ldots f_r(i)$. Lookup($i$) returns 1 if *all* of the $f_j(i)$ are 1; the probability of a false negative is 0, and the probability of a false positive is small (the product of the individual false-positive probabilities). On the Tofino switch [10], one cannot make r different accesses to the same array, so we implement the Bloom filter with r separate arrays.

A firewall that runs for a long time does not want its hash tables to fill up with stale data, but standard Bloom filters do not (soundly) support deletion; so we use a *sliding window* Bloom filter [30], as we will now explain.

A SBF has k "panes". Each pane is a Bloom filter containing r rows with S slots each. At any given time, $k-1$ panes store the "recent outgoing packets;" we add new IP addresses only to the most recent pane, and the extra pane is in the process of being incrementally cleared to prepare for reuse. We use $T = 60$ seconds as our window, i.e. the period for which we guarantee absence of false negatives. We rotate panes every $T/(k-2)$ seconds. That is because, in the worst case of querying, the most recent pane has just started but the remaining $k-2$ panes can still guarantee that keys inserted within T seconds are present. The client must incrementally clear the slots in the timed-out pane at least once every $T/C \approx 100\,\mu$s, where $C = (k-2)S$, so that the oldest pane will be fully cleared before reuse.

Our SBF has $k = 4$ and $r = 3$, and is lightly modular: the pane control contains r row instances – a row being a register of width $S = 2^{18}$ –, the SBF control contains k pane

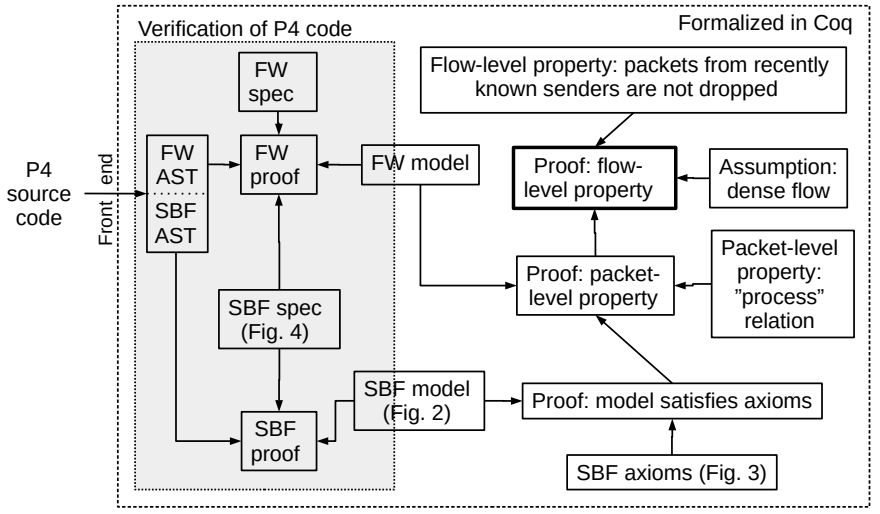

Figure 1 Overview of Stateful Firewall Verification.

instances, and the SBF itself is a control that is instantiated once by the firewall client to make decisions about which packets to drop. Our verification reflects this structure, exploiting the phase distinction between control-instantiation and packet-processing. The length of our program is largely independent of the number of panes in each window and the number of rows in each pane.

In total, our example is a 600-line P4 program that you really don't want to see, and we will accommodate you for the most part.

2.1 Proof organization and functional model

Fig. 1 shows the specifications and proofs, concluding with the main theorem: on any "dense flow", the firewall satisfies the property shown as equation (1)[1]

The *packet-level correctness property* is a relation, $process(s, p, s', r)$, where s, s' are the switch's state before and after processing the packet p and r is the result: forward or drop. To reason about multi-packet policies, we define (purely in Coq, independent of P4) a notion of transitive closure of *process* that maintains the histories of incoming and outgoing packets, the latter list in fact operating over optional packets to model *drop*. Thus, all that remains is to prove that the P4 program, on a single packet, satisfies the *process* relation.

Our tool's front end generates separate abstract syntax trees (ASTs) for separate control blocks, as indicated in Fig. 1. We use Verifiable P4 to prove semi-automatically that the SBF code implements the SBF model and that the firewall (FW) code implements the FW model. We prove directly in Coq that the SBF model obeys certain abstract SBF axioms (discussed below) which in turn feeds into the proof that the FW model satisfies the *process* relation.

[1] We model time in correspondence with the timestamp in metadata from the switch, which we assume the switch inserts correctly. The time $p.t$ in our flow-level specification is an unbounded mathematical integer; the timestamp in the packet header is a 48-bit unsigned integer measured in nanoseconds, $p.t \bmod 2^{48}$. Our proof that the program is correct does allow the timestamp to cross the 2^{48} boundary. A *dense flow* is one in which the mathematical timestamps are monotonically increasing—this is guaranteed by the switch ingress hardware—and in which there is never a gap greater than $100\mu s$ (that is, T/C) between packets. We need this assumption to ensure that the P4 program keeps up with its incremental clearing obligation. We can guarantee a dense flow 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 ($100\mu s$) intervals.

```
Definition row := list bool.
Definition row_insert (r : row) (i : Z) : row := upd_Znth i r true.

Definition pane := list row.
Definition pane_insert (a:pane) (is:list Z):pane:= map2 row_insert a is.

Record SBF := mk_SBF
  { SBF_panes : list pane; SBF_clear_index : Z; SBF_timer : Z * bool; }.

Definition get_clear_pane (t : Z * bool) : Z := fst t / pane_tick_tocks.

Definition SBFinsert (f : SBF)(tick : bool)(is : list Z):SBF :=
  let '(mk_SBF panes clear_index timer) := f in
  let new_clear_index := update_clear_index clear_index in
  let timer := update_timer timer tick in
  let c := get_clear_pane timer in
  let i := get_insert_pane c in
  let panes := panes[c := pane_clear panes[c] clear_index] in
  let panes := panes[i := pane_insert panes[i] is] in
  mk_SBF panes new_clear_index timer.
```

Figure 2 Functional model of an SBF, written in Coq's functional language (excerpt).

Fig. 2 shows the data structures of the SBF functional model, and the function modeling the insertion operation. The model mirrors the hierarchy of control blocks in the P4 code: a *row* is a list of Booleans (modeling the contents of a hash table), a *pane* is a list of rows, and an *SBF* is a list of panes, together with fields for the timer and the clearing maintenance. The full model contains additional functions `SBFquery`, `SBFclear`, and the initial state `SBFempty`.

The functional model will be referred to in specifications in the following section, but the proof of the packet-level policy only relies on certain axioms that specify these operations and express the "no recent false negatives" property; see Fig. 3. For example, **QueryInsertSame** has the premise $t \leq t' \leq t + T$, saying that an element inserted into the table at time t will be retrievable up to T seconds later. The three final laws express that the client must perform an incremental-clear (or an insert) at least every T/C seconds.

▶ **Theorem 1** (Coq). *The functional model of the SBF (Figure 2) satisfies the axioms given in Figure 3.*

2.2 Function specifications

Our logic equips each P4 function with a *function specification*, as illustrated in Fig. 4 for the SBF insertion operation. The specification components may be understood as follows:

MOD: the insertion code modifies stack variables with no restriction and modifies (only) within the external object rooted at path p (but not other objects).

WITH: the universally quantified abstract (logical) variables bound here (k, tstamp, f) have scope that extends to the end of the postcondition.

PRE: The precondition describes the state before executing the insertion, in three clauses— ARG: function parameter values, MEM: program stack variables (e.g. headers and structs, empty in our example), and EXT: external object contents (persistent registers).

POST: The postcondition describes the state after function execution, also divided into three parts (with return-value together with out-parameter-value).

```
"ok_until f t" means that the client has performed its
incremental clearing obligation at least through time t.

If state f is OK until (at least) deadline t, 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 3 Axioms of an SBF, written in Coq

```
Definition INSERT_spec : func_spec :=
  MOD None [p]
  WITH (k : key) (tstamp : Z) (f : filter),
    PRE (ARG [key_to_sval k; P4Bit 8 INSERT; P4Bit 48 tstamp; P4Bit 8 0]
          (MEM [ ]
          (EXT [SBF_repr p f])))
    POST (ARG_RET [P4Bit 8 0] ValBaseNull
           (MEM [ ]
           (EXT [SBF_repr p (SBFinsert f (tstamp, k))]))).
```

Figure 4 Specification of SBF insertion

In this case, the precondition says, "The arguments are the key, an operation code with constant value INSERT (this argument selects which SBF operation is performed), a timestamp *tstamp*, and an 8-bit value 0; there is no header or struct to take note of; and the external registers rooted at p represent a SBF with contents f." Similarly, the postcondition says, "The out parameter is 8-bit value 0, and the external registers rooted at p represent an updated SBF as described by the SBFinsert function from the functional model."

Similar to representation predicates in separation logic [21], SBF_repr is a user-defined predicate that relates the abstract view of an SBF to the corresponding P4 control, as laid out in Tofino registers. The predicate's definition again mirrors the nesting structure of controls and is detailed in §2.3 below.

To prove that the insertion operation of the SBF control satisfies INSERT_spec, our tool creates a symbolic state in Coq as described by the precondition. The user then steps through the P4 code, applying forward-mode Hoare rules from our program logic; when reaching the end of the control block one proves that the resulting symbolic state implies the postcondition. The proof proceeds mostly automatically, but in some places the user will have to direct it what to do.

▶ **Theorem 2** (Coq). *The SBF control block of our P4 program correctly implements the functional model (Figure 2).*

This is more than a *safety proof*: it proves not just "this implementation won't crash," it guarantees that the program really behaves like a lookup table with no false negatives.

2.3 Hierarchical State Assertions

According to the P4 language specification, each control or parser declaration is a class definition with local variable, object, and function members. The local variables are temporary for each call to the class, so they are considered as stack variables. Objects include tables, external objects, and control/parser instances. Stateful external objects (e.g., registers) have a persistent state that is preserved between packets.

In P4$_{16}$, controls and parsers are *instantiated* recursively. Each object has a global fully qualified name, generated by appending its local name to the global name of the parent object. Thus, instantiation forms a tree.

We exploit this hierarchy during specification and proof to make it easy to associate *abstract data types* to P4 state. We encapsulate all objects instantiated inside a root object in a single representation predicate, so that a change in the root object's implementation does not affect client verification. For this purpose, we define predicates along the structure of the hierarchical state, associating each predicate with a path-prefix containing an abstraction of the path names of all sub-objects.

The simplest assertion is of the form $p \mapsto r$, which means the register whose global name is p has value r. The path prefix of this assertion is $\{p\}$. A control block may have multiple registers. For assertions $P_1, \ldots, P_n$ with path prefixes $S_1, \ldots, S_n$, the path prefix of the assertion $P_1 \wedge \cdots \wedge P_n$ is $\bigcup_i S_i$. To encapsulate the state of an object rooted at p, we want to say an assertion has path prefix at most $\{p\}$, without mentioning its contents. So we define a "wrap" operator that wraps an assertion with a more abstract path prefix. Formally, for assertion P whose path prefix is S_1, $\mathrm{wrap}(S_2, P)$ is a valid assertion if $S_1 \sqsubseteq S_2$, that is, every mapping in S_1 is also in S_2. A path p "covers" all of its subtrees, for example $\{p.x\} \sqsubseteq \{p\}$.

The assertions for an SBF containing k panes with r rows each are defined as:

$$
\begin{aligned}
\texttt{row\_repr}(p, \mathit{row}) &:= \mathrm{wrap}([p], p.\texttt{reg} \mapsto \mathit{row}),\\
\texttt{pane\_repr}(p, \mathit{pane}) &:= \mathrm{wrap}([p],\ \texttt{row\_repr}(p.\texttt{row1}, \mathit{pane}.\texttt{rows}[0]) \wedge \ldots \wedge\\
&\qquad\qquad \texttt{row\_repr}(p.\texttt{rowR}, \mathit{pane}.\texttt{rows}[r-1])\\
\texttt{SBF\_repr}(p, \mathit{filter}) &:= \mathrm{wrap}([p],\ \texttt{timer\_repr}(p.\texttt{timer}, \mathit{filter}.\texttt{t}) \wedge\\
&\qquad\qquad \texttt{pane\_repr}(p.\texttt{pane1}, p.\texttt{panes}[0]) \wedge \ldots \wedge\\
&\qquad\qquad \texttt{pane\_repr}(p.\texttt{paneK}, p.\texttt{panes}[k-1])).
\end{aligned}
$$

This concludes the description of our example.

3 How the verifier works

Our verifier contains a verification-oriented P4 representation with a parser, an operational semantics, a program logic, and automation tactics.

3.1 P4light abstract syntax, front end

When designing our intermediate language, *P4light*, our desiderata were to

1. include a type-annotation at every expression node;
2. fully disambiguate names, i.e. distinguish local versus global variables;
3. avoid side-effect expressions inside subexpressions;[2]
4. and yet, stay sufficiently close to source-level P4 so that every P4light program can be pretty-printed as a legal and compilable P4 program.

Front end. We adapted Petr4's front end [5] (including its type-checker) to produce P4light ASTs from P4 source programs.

Hierarchical name space. To distinguish names in different scopes, we decorate names with *locator* annotations. A name has locator *glob* p if it is defined in the global scope, or locator *inst* p if it is defined inside a parser/control declaration. In either case, p is a qualified name (*path*) such as `myIngress.x`, so objects are uniquely named. Qualified names can further expand into fully qualified names in the instantiation phase (§4).

Unnesting expressions. To simplify reasoning about programs in P4light, we hoist expressions out of function calls, adding extra local variables in the process.

Representation. Our ASTs are expressed using Coq's inductive data types, with one type for each syntax class of the AST grammar (expression, statement, function, *etc.*).

3.2 Operational semantics

We define P4 execution as a big-step operational semantics that operates over states, defined as pairs of a stack frame and an external state. A stack frame (resp. external state) is a partial mapping from paths (i.e. fully qualified names) to values (resp. external object values).

$$\begin{aligned} \texttt{StackFrame} &:= \texttt{Path} \rightharpoonup \texttt{Value} \\ \texttt{ExternState} &:= \texttt{Path} \rightharpoonup \texttt{ExternObject} \\ \texttt{State} &:= \texttt{StackFrame} \times \texttt{ExternState} \end{aligned}$$

Only the external state persists from one packet to another. Because P4 does not have explicit pointers, we don't need to mention memory addresses, only paths.

Let Γ be the global static environment produced in §4 below. As P4 program statements are mostly inside controls and parsers, we need to know where a statement is in order to execute it. 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). Let s be a state. We write our big-step semantic

[2] 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.

judgments as

$\Gamma, p, s \vdash e \Downarrow v$	(expression, 18 rules)
$\Gamma, p, s \vdash e \Downarrow lv$	(l-expression, 5 rules)
$\Gamma, p, s \vdash stmt \Downarrow (s', sig)$	(statement, 16 rules)
$\Gamma, p, s \vdash e \Downarrow (s', sig)$	(call-expression, 2 rules)
$\Gamma, p, s \vdash f, a_{\text{in}} \Downarrow (s', a_{\text{out}}, sig)$	(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 P4light expression e evaluates to value v." Judgment $\Gamma, p, s \vdash \text{stmt} \Downarrow (s', \text{sig})$ 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, like return and exit statements.

As usual in operational semantics, each of these judgments is an inductive relation. If no rule applies, the operational semantics is *stuck*, a technical representation of *undefined behavior*. P4 is designed so that programs that type-check cannot have undefined behavior – a formal proof that this is indeed the case is under current development.

The operational semantics of a control block is given by that of its variable initialization, followed by that of its apply function. More details on the operational semantics presented in the forthcoming PhD thesis of the first author [25].

We briefly discuss two differences between P4 and more traditional languages, to illustrate the challenges we faced.

Undefined Values. According to the P4 specification, reading an uninitialized field or an invalid header yields an undefined value. As of 2021 the official description in the P4_{16} reference manual was ambiguous, but the P4 committee clarified that each bit of such a field can be either 0, 1, or uninitialized. To characterize this in assertions and our tool's symbolic execution, we use an abstract interpretation over bits. The abstract domain for an n-bit field is $\{0, 1, \bot\}^n$, where 0 and 1 characterize the two fully determined values and $\bot$ means the bit's value can be arbitrary, including undefined.

P4 data structures (headers, structs, bitfields) may be partially uninitialized, but P4 expressions and subexpressions are fully defined. That means, when reading from a data structure the semantics must choose arbitrary 0s and 1s for the undefined bits, so we use a *havoc* construct at the appropriate places.[3] Exactly when and how this happens was unclear in the P4_{16} document, but in discussions with the committee we clarified the document and formalized these clarifications in our semantics.

Our treatment of partially defined values in our operational semantics carries over to the logic and verification tool. Other verification tools for P4 do not treat this exactly; they either assume that all uninitialized bits are 0 (which is unsound) or cannot reason about uninitialized fields at all (which is incomplete).

Compiler-rejected programs. In many languages, if a program is legal then you can expect that the compiler will compile it. In P4 that's explicitly not the case: a legal P4 program may violate architecture-specific pipeline and resource constraints, and be rejected by the compiler. For this reason, our operational semantics (and verifier) does not model architecture-specific constraints, so one should really read our soundness theorem

[3] "Havoc" is a standard term in operational-semantic reasoning to indicate arbitrary behavior where it is not necessary to know which choice a program will make.

as, "If your compiler agrees to compile the program *and* you prove some correctness property in *Verifiable P4*, then your program will indeed respect that property."[4]

Validating the operational semantics, and debugging P4.

We validated that the operational semantics accurately captures the behavior of real compilers and hardware (e.g., Tofino, V1model) using three approaches: correspondence to the $P4_{16}$ reference document; correctness of a P4 reference interpreter w.r.t. our semantics; and testing against Tofino. We comment on the former two activities. A fourth approach, validating our reference interpreter against existing P4 test suites, is ongoing work.

Comparing to the $P4_{16}$ reference manual. We claim that our formalization specifies the same as what the official P4 standard means and what the commercial compilers do. Our process has been to rigorously formalize what is written down informally in the specification; when we find ambiguities, errors, or disagreements between $P4_{16}$ and commercial compilers, we discuss those with the P4 committee—so the official English-language standard gets refined, or bugs get fixed in the commercial compilers. See Table 1. By the time our process has finished, there is real evidence that the formal semantics has meaningful utility, and agrees with the $P4_{16}$ specification.

3.3 Reference Interpreter

In order to execute P4 programs and test our semantics, we wrote a reference interpreter in Coq's embedded functional language. We proved correctness of the interpreter with respect to our operational semantics and used extraction to obtain an executable OCaml program.

The main task in programming the interpreter was translating inductive relations (from the big-step semantics) to Coq functions. Many of the inductive relations are nondeterministic, so the interpreter determinizes them uniformly. Wherever an undefined value is involved in a computation, the interpreter initializes it using zero bits. This is sound, but means that some behaviors exhibited by the big-step semantics cannot be observed in the interpreter.

The interpreter shares some code with the operational semantics, which uses functions for many important (but deterministic) subroutines. For instance, instantiation (see §4) is handled by a functional program in the operational semantics. The interpreter reuses this program, preserving the phase distinction between instantiation and evaluation.

P4 allows "architectural extensions" such as V1model registers or Tofino's registers (which are different from each other). Our semantics handles those extensions as a kind of plug-in. Our verification tool and the reference interpreter treat the extensions by directly using this plug-in, so the reference interpreter can serve for core P4, for V1model P4, or for Tofino P4. Our example (the stateful firewall) happens to be a Tofino P4 program.

3.4 Program Logic and Proof Automation

We have designed a program logic for P4 (except for packet parsing[5]), as a set of proof rules that have been proven sound in Coq with respect to our operational semantics. As all P4 programs are terminating, the distinction between partial and total correctness vanishes, but P4's hardware-orientation, and the goal to not be overly specific to a concrete architectural

[4] And if your compiler is correct, then your program *as compiled* will respect the property too; our formalization of the operational semantics should also support the development of compiler proofs.

[5] Verification of P4 parsing is the subject of a separate project [6], with which we expect to connect.

	Section	Issue	Git	Status	p4c Bug
Expression	8.5, 8.6	Types of the bit slicing index are vague.	955	Released	No
	8.5	Concatenation is missing from the operations on the bit type.	956	Released	No
	8.5, 8.6	Concatenation is not excluded from the binary operations that require same-type operands.	956	Released	No
	8.9.2	Concatenation and shift are not excluded from the binary operations that allow implicit casts.	957	Released	No
	8.9.2	Unclear where implicit casts of serializable enum are allowed.	958	Released	Yes
	8.7	Right operand types of integer shifts are vague.	959	Released	No
	8.10-12 8.14-15	Allowed comparisons between lists, tuples, structs, and headers are unexplained.	960	Pending	Yes
	8.11, 8.12	Implicit casts between lists, tuples, structs, and headers are unexplained.	953	Pending	No
	8.10	Explicit casts between non-base types are unexplained.	961	Released	No
	8.7	Slicing integers is not allowed.	1015	Pending	No
	8.22	Reading uninitialized values is confusing and vaguely defined during argument passing.	988	Pending	Maybe
	8.13	Types of set operations are vague.	969	Pending	Maybe
Function	10.3.1	Abstract extern methods open multiple back doors, e.g., allowing recursion and invoking parsers in controls.	973, 976, 979	Pending	Maybe
	App. F	Parameter restrictions for extern functions are missing.	972	Released	No
	6.7.2	Optional parameters are not allowed in parser and control types.	977	Released	Maybe
Instantiation	11.3,A.H	Instantiation should not be a statement.	975	Released	Yes
	17.2	The concept of instantiation-time known constants is missing.	932	Pending	Maybe
	12.10, 13.4	The difference between stateful and stateless instantiations in parsers and controls is poorly explained.	926	Pending	No
Table	13.2	Default action is not set as NoAction when undefined.	933	Released	No
	13.1	The possible sources of action data are defined in a misleading way.	914	Released	No
Name	17.3	Value sets are not included in control plane objects.	962	Released	No
	6.8	Name duplication and name shadowing is undefined.	974	Pending	Maybe
	6.4	Inconsistent name style for built-in methods & fields.	1004	Pending	Yes

Table 1 Specification issues clarified during our operational-semantic formalization.

Section refers to section numbers in the $P4_{16}$ manual [3]. *Git* refers to our issue reports in the repo for that manual, https://github.com/p4lang/p4-spec. *Status* "pending" means under discussion with the steering committee; "released" means that a published version of the manual incorporates our merged pull-request. As indicated, some of these issues also reflect bugs in the p4c compiler, some of which have now been corrected.

model make the logic challenging to design. Concrete challenges arose in the treatment of headers, from the fact that the category of values includes `structs`, and from the combination of hierarchical path names and instantiation. We do not show the proof rules in this paper, and also omit a detailed discussion of automation support that uses Coq's tactic language *Ltac*. As explained in the previous section for the firewall example, the typical user must provide data representation predicates (how the abstract values of interest are represented in the data structures of P4 programs), function specification (preconditions, postconditions), specifications of control blocks (i.e., pre/postconditions of apply functions), and specifications of registers or other architecture-specific externals.

Our logic is modular in the sense that for any control block definition in the code we can write a single proof script that can be adapted to multiple instantiations. Thus, a library module such as our SBF can be proven to satisfy its specification (set-membership check with no false negatives) quite independently of the correctness proof of its client.

▶ **Theorem 3** (Coq). *(Soundness) The proof rules of our program logics are sound w.r.t. the operational judgments in §3.2.*

Soundness of our verifier then follows, as the verifier's tactics build a machine-checked proof using the logic's inference rules.

4 Instantiation

In accordance with Sec. 18 of the P4 specification [3], we divide evaluation into two phases: instantiation and execution. In the instantiation phase, constructor calls are evaluated to produce a static environment of *objects* (instances of control blocks, tables, parsers, packages, and external objects). This instantiation phase determines all relationships between objects and thus avoids any kind of dynamic allocation or closure passing. Indeed, instantiation more closely resembles creation of statically allocated objects/structs in C-like languages and instantiation of Verilog modules than dynamic object allocation.

The instantiation phase assigns a distinct *fully qualified name* to each object. For tables and external objects, it is convenient that this name is the same as the *control plane name* described in the $P4_{16}$ reference manual, which can be used by the control plane to access the objects, *e.g.* updating table entries and reading registers. Then for each object, its name is bound to its record of code and references to generate the static environment. The static environment is passed later into the execution phase, where it is read from but never modified. This is only possible because P4 is designed to avoid dynamic allocation.

In contrast, Petr4 [5] combines the two phases, using closures and dynamic allocation in the evaluation semantics. As a consequence, Petr4 requires a more complex state model with stores (mapping locations to values) and environments (mapping names to locations).

The (simplified) pseudocode of the instantiation phase is in Fig. 5. Function `instantiate_prog` iterates over all declarations in the program. For control and parser declarations, it puts them in `decl_env` to look them up when encountering their instances. For instantiation declarations, it calls `instantiate` with the empty path ϵ, which indicates to instantiate at the top level. `instantiate` takes a path `p` and an instance declaration `decl` to instantiate `decl` that appears in `p`. Let `inst_name` be the local name of the instance of `decl`. Then its fully qualified name is `p.inst_name`. `instantiate` allocates the object at `p.inst_name`, and then allocates its inner declarations recursively.

This procedure allocates the fully qualified names and constructs the global static environment at the same time. Inner declarations are allocated under the path `p` and they

```
global ge := ...
global decl_env := ...

procedure instantiate(p, decl) :=
    inst_name := instance name of decl
    class_name := class name of decl
    body := decl_env[class_name]
    ge := ge[p.inst_name ↦ record of the instance]
    for each decl in body
        if decl is an instantiation then instantiate(p.inst_name, decl)

procedure instantiate_prog(prog) :=
    for each decl in prog
        if decl is an instantiable declaration then
            decl_env := decl_env[(name of decl) ↦ decl]
        else if decl is an instantiation then instantiate(ϵ, decl)
```

Figure 5 Pseudocode of instantiation

have distinct local names, so their global fully qualified names are distinct and different from names from other parts of the program. It also provides a local view under path `p`.

We have implemented the instantiation phase as a function in Coq. Users can just use Coq's computation mechanism to evaluate P4 programs.

4.1 Abstract Methods

Suppose you want to increment a (persistent) register on a Tofino P4 switch. You might read the register into a local variable, add one, and write back to the register. But that violates a Tofino pipeline constraint: accessing the same register in two different pipeline stages.

The $P4_{16}$ spec permits architectural extensions including stateful objects on which the P4 program may invoke so-called *abstract methods*. Tofino's registers are such extern objects, and provide a read-modify-write abstract method: the user specifies what *modify* to perform (like incrementing a value, as in our SBF insertion), but not the surrounding register read/write operations – these are implicitly performed by Tofino's invocation mechanism. Without abstract methods, Tofino registers are not much use, but their 'almost-object-oriented' realization is arguably even more complex than function pointers in C or virtual or abstract methods in C++ or Java.

Our operational semantics is the first formal specification of P4 to include abstract methods, and *Verifiable P4* is the first verifier to support them.

5 Proof statistics

Our verification of the sliding-window Bloom filter (SBF) is summarized in Table 2. The P4 proof of "Filter" is so large in part because the P4 program has many branches about which we need to reason. The high-level "Filter" proof is large for a different reason: in part because the correctness argument for sliding-window Bloom filters is not trivial; and because we use two intermediate models to facilitate reasoning, which may not have been the best way to organize that proof.

	P4 code	Func. Model	Funcspecs	P4 proof	High-level proof
Row	53	85	165	140	106
Pane	22	62	235	140	87
Filter	341	333	858	1579	1068
Total	416	480	1258	1859	1261

Table 2 Lines of P4 code, specifications, Coq proofs. For each module of the P4 program we have (in the order shown), a functional model, Verifiable P4 function-specifications with preconditions and postconditions, a Coq proof script showing that the P4 code implements the model, and a proof that the model satisfies the high-level specification.

6 Count-min-sketch

We also verify count-min-sketch (CMS) [4] to illustrate that the existing proof script can be easily adapted to verify similar data structures. A CMS is essentially the same as a counting Bloom filter, which counts the frequency of the different types of events in the stream.

Most parameters of a simple CMS data structure are the same as a simple Bloom filter, such as r rows and S slots. The difference is that each slot now indicates how many times an event occurred rather than whether it occurred at all. The insert operation thus increments the relevant slots, *saturating* at `Int<k>.max` if each slot is k bits wide. The query operation returns the minimum value of the matching slots in all rows.

Our CMS implementation employs the same sliding window mechanism as the SBF, and similarly for the functional model. The axiomatization employs the same predicate `ok_until` and reuses most axioms, except for the two axioms shown in Figure 6. From these axioms one can derive the analogue of the Bloom filter's "no false negatives", that CMSquery returns a number not less than the true count. In the function specification (not shown here but analagous to Figure 4) one can see that the value returned is the minimum of this count and the saturation value `Int<k>.max`.

```
QueryInsertSame : ∀f t t' h k, ok_until f t → t ≤ t' ≤ t+T →
     CMSquery f (t', h) = Some k →
     CMSquery (CMSinsert f (t, h)) (t', h) = Some (k + 1).

QueryInsertOther : ∀f t t' h h' k, ok_until f t → t ≤ t' →
     CMSquery f (t', h') = Some k →
     CMSquery (CMSinsert f (t, h)) (t', h') = Some k ∨
     CMSquery (CMSinsert f (t, h)) (t', h') = Some (k + 1).
```

Figure 6 Axioms of Count-Min-Sketch (excerpt)

7 Discussion

Related work. Vera [23], p4v [15], and ASSERT-P4 [18] are automatic P4 verifiers for simple properties—including safety properties, architectural constraints, and simple stateless application properties. They translate the P4 program into a guarded command language (p4v), into a network verification language called SEFL (Vera), or into C (ASSERT-P4). ASSERT-P4 does not claim to support stateful objects. Vera [23] uses an extremely expensive encoding that is proportional to the size of registers (impractical when the register contains

an entire hash table). Although p4v supports stateful objects, we cannot find any evidence that p4v can relate initial and final states.

Aquila [24] 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 [22] (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.

Π4 [8] is a dependent refinement type system for a language similar to a subset of P4. Although described as a type system, it includes assertions in types and requires an SMT solver to check the types, so it is very similar to a verifier with assertions in middle of programs. Of all these verifiers, it is the only one that supports modular specification and verification function-by-function—its dependent function type is equivalent to a function contract. But Π4 expands the instantiation hierarchy before verifying, but does not exploit the similarity of the resulting instances. Π4 does not support stateful objects.

There are some other tools that focus on particular properties. bf4 [7] is a tool that automatically infers constraints of tables to make a P4 program safe. Leapfrog [6] is a parser equivalence checker for P4.

Petr4 [5] 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 and execution (§4). We also identify and fix some bugs in Petr4 during the formalization in Coq.

The approach taken by Vigor [28, 20, 27] is conceptually similar to our envisioned separation into verification of reusable data structures and more abstract packet- or flow-level network functionality. Vigor is a verifier for C programs on top of DPDK; it employs Verifast [11] for the reusable data structures and the Klee symbolic execution engine [1] for the C code that is a client of these data structures. Vigor has been applied to a number of data structures and network functions, with more automation than our framework currently supports. We expect that as top-level policies get more complex, the expressivity of a general-purpose proof assistant (that we use) will prove beneficial. The use of ADT operations as the interface between the two parts of a Vigor proof is conceptually similar to our use of functional or axiomatic models, but—unlike in our approach—there is no machine-checkable proof that connects these layers together. Continuing this line of work, KLINT [19] applies to binary code, with a focus on the code that implements network functions on top of a small fixed set of map-like data structures whose implementations are assumed correct.

Gravel [29] verifies C++ implementations of Click-style middlebox functions against functional specifications (represented as Python programs), using symbolic execution on LLVM and an SMT solver backend. Like Vigor and KLINT, Gravel achieves a higher degree of automation than our work but has a larger trusted code base, does not support expressive reasoning about specifications, and does not apply to line-rate processing on a P4 switch.

Gopinathan and Sergey [9] show how to prove in Coq that a Bloom filter (functional model) has not only a zero false-negative rate but also a small false-positive rate. Our system could accommodate that style of proof: use Verifiable P4 to prove (as we do) that the Bloom

filter exactly implements its functional model; then prove (as we did not) that this functional model has a small false-positive rate, using the method that they demonstrate.

Future work. We currently achieve modular verification by writing a proof script for one instance of a control block (such as *pane* or *row*) that can be re-used unchanged for all the other instances. In future work, we expect to make this more formal, to *guarantee* that each control needs only one proof of correctness.

To support switches with multiple pipelines (each with its own persistent register state), we may either modify the program logic, or treat the independent pipelines as separate switches and account for concurrency at the model level, when proving the relation between the single-packet correctness property and the flow-level property.

Some stateful applications mentioned in Section 1 involve packet recirculation, but we have not explored this in detail, especially the issue of buffering synthesized packets.

Finally, a long-term goal may be to augment tools that synthesize P4 code with mechanisms to generate *Verifiable P4* specifications, or even proofs. For example, we use CatQL to synthesize our bloom filter programs but have no tools to derive specifications or proofs for the synthesized P4 code from CatQL directly.

Our formalization assumes that each P4 packet is handled atomically, which is a justifiable assumption for single Tofino-like pipelines.[6] But Tofino (and other P4-programmable switches) may have multiple parallel pipelines with independent register sets, and multiple sequential pipelines (i.e., ingress and egress pipelines) with nontrivial scheduling between them. Reasoning about packet management *outside* of P4 is future work—though it may not require any changes to our P4 semantics itself.

Conclusion. P4-programmable switches enable a new generation of applications that use sophisticated data structures, modular software engineering, and persistent state. The constraints of P4-capable hardware often mean that those programs are written in a somewhat contorted way that makes them difficult to reason about. Therefore, verification tools are even more useful for P4 than they are for conventional languages.

We have built a verifier for P4 and demonstrated it on a sophisticated application. Our verifier is the first one that can handle rich specifications or nontrivial uses of persistent state. It is the first to be proved sound with respect to a formal semantics of P4. And our formal semantics is the first one that attempts to be complete with respect to the English-language $P4_{16}$ reference manual.

References

1 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. URL: `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

2 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*

[6] But not a *trivial* assumption: a multistage pipeline may have stateful registers at different pipeline stages, and we can treat the whole pipeline atomically only because (1) each register is accessed only at a given pipeline stage by *all* packets that access it and (2) packets cannot overtake each other within the pipeline.

Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 226–239, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3387514.3405865.

3 The P4 Language Consortium. $P4_{16}$ language specification, version 1.2.3. Technical report, July 2022. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.3.pdf.

4 Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. URL: https://www.sciencedirect.com/science/article/pii/S0196677403001913, doi:10.1016/j.jalgor.2003.12.001.

5 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. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi:10.1145/3434322.

6 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. ACM, 2022.

7 Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: towards bug-free P4 programs. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 571–585. ACM, 2020. doi:10.1145/3387514.3405888.

8 Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. Dependently-typed data plane programming. *Proc. ACM Program. Lang.*, 6(POPL):40:1–40:28, 2022. doi:10.1145/3498701.

9 Kiran Gopinathan and Ilya Sergey. Certifying certainty and uncertainty in approximate membership query structures. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II*, volume 12225 of *LNCS*, pages 279–303. Springer, 2020. doi:10.1007/978-3-030-53291-8_16.

10 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.

11 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Proceedings*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

12 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. ACM, 2017. doi:10.1145/3132747.3132764.

13 Patrick Kennedy. Intel Tofino2 next-gen programmable switch detailed. https://www.servethehome.com/intel-tofino2-next-gen-programmable-switch-detailed/, August 2020.

14 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. USENIX Association, 2016.

15 Jed Liu, William T. Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Calin Cascaval, Nick McKeown, and Nate Foster. p4v: Practical verification for programmable data planes. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2018)*, pages 490–503. ACM, 2018. doi:10.1145/3230543.3230582.

16 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. ACM, 2017. doi:10.1145/3098822.3098824.

17 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. URL: https://www.usenix.org/conference/nsdi22/presentation/namkung.

18 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. ACM, 2018.

19 Solal Pirelli, Akvile Valentukonyte, Katerina J. Argyraki, and George Candea. Automated verification of network function binaries. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 585–600. USENIX Association, 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/pirelli.

20 Solal Pirelli, Arseniy Zaostrovnykh, and George Candea. A formally verified NAT stack. *Comput. Commun. Rev.*, 48(5):77–83, 2018. doi:10.1145/3310165.3310176.

21 John Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002.

22 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.

23 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.

24 Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. 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.

25 Qinshi Wang. *Foundationally Verified Data Plane Programming.* PhD thesis, Princeton University, 2023. In preparation.

26 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. doi:10.1145/3387514.3405857.

27 Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. Verifying software network functions with no verification expertise. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 275–290. ACM, 2019. doi:10.1145/3341301.3359647.

28 Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina J. Argyraki, and George Candea. A formally verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 141–154. ACM, 2017. doi:10.1145/3098822.3098833.

29 Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages

221–239. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/zhang-kaiyuan.

30 Linfeng Zhang and Yong Guan. Detecting click fraud in pay-per-click streams of online advertising networks. *2008 The 28th International Conference on Distributed Computing Systems*, pages 77–84, 2008.