p4v: Practical Verification for Programmable Data Planes

Jed Liu
Barefoot Networks
Ithaca, NY, USA

William Hallahan
Yale University
New Haven, CT, USA

Cole Schlesinger
Barefoot Networks
Santa Clara, CA, USA

Milad Sharif
Barefoot Networks
Santa Clara, CA, USA

Jeongkeun Lee
Barefoot Networks
Santa Clara, CA, USA

Robert Soulé
University of Lugano
Lugano, Switzerland

Han Wang
Barefoot Networks
Santa Clara, CA, USA

Călin Caşcaval
Barefoot Networks
Santa Clara, CA, USA

Nick McKeown
Stanford University
Stanford, CA, USA

Nate Foster
Cornell University
Ithaca, NY, USA

ABSTRACT

We present the design and implementation of p4v, a practical tool for verifying data planes described using the P4 programming language. The design of p4v is based on classic verification techniques but adds several key innovations including a novel mechanism for incorporating assumptions about the control plane and domain-specific optimizations which are needed to scale to large programs. We present case studies showing that p4v verifies important properties and finds bugs in real-world programs. We conduct experiments to quantify the scalability of p4v on a wide range of additional examples. We show that with just a few hundred lines of control-plane annotations, p4v is able to verify critical safety properties for `switch.p4`, a program that implements the functionality of on a modern data center switch, in under three minutes.

CCS CONCEPTS

• **Networks** → *Programming interfaces*; • **Software and its engineering** → **Software verification**;

KEYWORDS

Programmable data planes, P4, verification.

ACM Reference Format:
Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM '18: SIGCOMM 2018, August 20–25, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3230543.3230582

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230582

1 INTRODUCTION

Suppose you wanted to verify the correctness of a network data plane. How would you do it? One approach, which is widely used today, is to rely on exhaustive testing—i.e., generate a set of input packets and test whether the device produces the expected outputs. Testing is expensive, since modern devices handle dozens of different packet formats and protocols, each requiring distinct test inputs. But with a conventional device these costs are paid only once, because its capabilities are "baked in" at manufacturing time and cannot be changed by programmers.

Recently, the field has started to shift to more flexible platforms in which data-plane functionality is not controlled by vendors but can be defined by programmers. The idea is that the programmer describes the functionality of the device using a program in a domain-specific language such as P4 [5, 44, 45], and the compiler generates an efficient implementation for the underlying target device. This approach not only facilitates rapid innovation, since new protocols can be deployed without having to spin new hardware, it also opens up opportunities for novel uses of the network—e.g., in-band network telemetry [26] and in-network caching [28, 29] to name a few. While increased programmability offers benefits, it also creates challenges related to correctness.

Example. Consider a "bump in the wire" firewall that uses `acl` and `nat` tables to filter and rewrite incoming packets (Figure 1 gives an implementation in P4). Suppose we wish to verify that if `acl` is populated with rules that drop packets going to a given internal host, the host will be isolated from the external network. Even for this simple property, several complications can arise, illustrating the need for verification.

First, the behavior of the program that implements the firewall may be undefined on certain kinds of packets since, according to the P4 language specification [44], reading or writing an invalid header produces an arbitrary result. In particular, although the `acl` table correctly matches and filters away IPv4 packets sent by external hosts, it might incorrectly forward other types of packets such as IPv6. Second, there is potential for confusion between internal and external addresses. If the program executes the `acl` table before the `nat` table, then the rules intended to filter away external traffic

should match on external addresses, but if it executes the tables in the opposite order, then the rules should match on internal addresses instead. Confusing the ordering of tables in a firewall may seem like a mistake that would be easily caught, but it is not a hypothetical concern: Cisco changed the order of these tables going from version 8.2 to 8.4 of their Adaptive Security Appliance, invalidating numerous configurations and leaving networks vulnerable to attacks [31].

Verified data planes. After hearing about an example like this one, a pessimist might conclude that an increase in bugs is an inevitable side-effect of making data planes more programmable. We believe the opposite is actually true. Whereas the behavior of a conventional device is largely unspecified and must be discovered through testing, a P4 data plane has a precise, bit-level description of how it processes packets in a human-readable language. By providing developers with powerful, language-based verification tools, we should be able to decrease the prevalence of bugs that arise in practice. P4 is an ideal target for automated verification because the language carefully excludes features such as loops and pointer-based data structures, which typically require manual annotation or complex analyses. Moreover, the potential for impact for a P4 verification tool is high, as the language is also being used to describe the behavior of conventional, fixed-function devices [41].

Our vision of verified data planes is inspired in part by recent successes in the formal methods community, which has shown that it is feasible to verify a wide variety of complex systems including compilers [38], operating systems [21], databases [40], distributed systems [25], and network controllers [22]. In addition, SAT and SMT solvers, which underpin many automated tools, have become very fast in recent years and are now able to scale to extremely large problem instances in many common cases [10]. With such tools at our disposal, we embarked to demonstrate that it is possible to build a practical tool for verifying programmable data planes. We believe that such a tool would provide a foundation for the many other network verification tools that have been proposed in recent years [1, 3, 13, 17, 18, 20, 32, 34, 39, 46, 47, 50, 53, 55, 56] and might also serve as a catalyst for follow-on efforts that target higher layers of the networking stack.

Contributions. This paper presents p4v, a practical verification tool for P4, and makes the following contributions:

- We motivate the need for data-plane verification using real-world examples, and we identify classes of common properties that arise in many P4 programs.
- We present a novel approach to data-plane verification that incorporates symbolic control-plane interfaces.
- We develop a prototype implementation and domain-specific optimizations that improve on naive approaches.
- Through case studies and experiments, we demonstrate that p4v is effectively able to find bugs in real-world programs and provides good performance.

Challenges. There are several challenges that arise when building a practical P4 verification tool. One issue is that a P4 program is really only half of a program. The contents of the match-action tables are not known until they are populated by the control plane at run time. Some verification tasks can be carried out by over-approximating the behavior of the control plane—i.e., by non-deterministically executing any of the actions listed in each table. However, real-world control-plane programs are often carefully engineered to coordinate rules installed across multiple tables, so many important data-plane properties cannot be established without an understanding of the interactions with the control plane.

Our p4v tool allows the programmer to define a *control-plane interface* that constrains the behavior of the data plane, making it possible to verify that it will behave as specified when combined with a control plane into a single program. For example, in our running firewall example, the control-plane interface might specify that the `acl` table must execute the `deny` action on packets destined for the internal server, as well as non-IPv4 packets. Under the hood, these constraints can be incorporated into the data-plane program using symbolic predicates on "ghost variables" that are automatically inserted by p4v. It is worth noting that the control-plane interface must currently be written by hand and is not verified. Automated synthesis of the control-plane interface from examples is a promising direction for future work.

Another challenge concerns scalability. Although P4 programs are limited to simple data structures and control flow, practical programs can be quite large, often running to tens of thousands of lines of code. In addition, domain-specific constructs such as parser state machines and match-action tables have dense conditional structure. This means that standard software verification approaches, such as symbolic execution [7], which explicitly traverses all control flow paths in the program, are unlikely to scale well, at least not out of the box [30]. In contrast, p4v is based on symbolic techniques that avoid explicit run-time traversals of the program source code. In addition, we have incorporated domain-specific optimizations that enable p4v to scale to some of the largest open-source programs that have been written to date.

Implementation and evaluation. We built an implementation of p4v in OCaml and evaluated its effectiveness and scalability on a variety of real-world programs. To ensure that p4v does not inherit bugs that might be present in the open-source reference implementation of P4, we built an independent front-end that includes a parser, type checker, and a simple translation into Dijkstra's guarded command language [11] and tested our front-end against several other

```
/* Header Types */
header_type ethernet_t {
 fields {
  dst_addr:48;
  src_addr:48;
  ether_type:16;
 }
}
header_type ipv4_t {
 fields {
  pre_ttl:64;
  ttl:8;
  protocol:8;
  checksum:16;
  src_addr:32;
  dst_addr:32;
 }
}
/* Instances */
header ethernet_t ethernet;
header ipv4_t ipv4;
```

```
/* Parsers */
parser start
  extract(ethernet);
  return select(ethernet.ether_type) {
    0x800: parse_ipv4;
    default: ingress;
  }
}
parser parse_ipv4 {
  extract(ipv4);
  return ingress;
}
/* Actions */
action allow() { }
action deny() { drop(); }
action nop() { }
action rewrite(saddr,daddr, port) {
 modify_field(ipv4.src_addr, saddr);
 modify_field(ipv4.dst_addr, daddr);
 modify_field(standard_metadata.egress_spec, port);
}
```

```
/* Tables */
table acl {
 reads {
  ipv4.src_addr:lpm;
  ipv4.dst_addr:lpm;
 }
 actions { allow; deny; }
}
table nat {
 reads {
  ipv4.src_addr:lpm;
  ipv4.dst_addr:lpm;
 }
 actions { rewrite; nop; }
}
/* Controls */
control ingress {
 apply(acl);
 apply(nat);
}
control egress { }
```

Figure 1: Example program: `firewall.p4`.

implementations, including the P4 compiler and software simulator for Barefoot's Tofino chip [51].

The p4v back-end uses the Z3 [9] theorem prover to discharge verification conditions and compute counter-example traces that can be used for debugging. Using several hundreds of lines of control-plane annotations, we successfully verified a number of critical safety properties for `switch.p4`, a large program that handles dozens of different packet formats and protocols, in under three minutes. We have also used p4v to validate common optimizations used by P4 compilers, and to find bugs in existing open-source programs.

2 BACKGROUND ON P4

This section briefly reviews the main features of the P4 language to set the stage for the design and implementation of p4v, which is described in the following sections.

P4 [5, 44] is a domain-specific language organized around packet-processing abstractions such as headers, parsers, tables, actions, and controls. The execution of a P4 program follows a simple abstract forwarding model with five distinct phases: parsing, ingress processing, replication and queuing, egress processing, and deparsing [6]. The declarations in a P4 program define the behavior of each of these phases. During execution, the state comprises the data extracted from packet headers, metadata supplied by the device (e.g., the ingress port that the packet arrived on) or computed by the program, as well as mutable state in counters and registers.

Figure 1 gives the source code for a P4 program that implements the firewall example discussed in the last section. We illustrate the main features of P4 using this program. The left part of the figure defines the types of the headers that are manipulated by this program as well as instances of those types, one for Ethernet and another for IPv4. These instances are initially invalid, but can be made valid by the parser, which is defined in the middle part of the figure. Instances are statically allocated and globally accessible. In addition to the instances explicitly defined by the programmer, there is also an implicit instance for `standard_metadata` that keeps track of information such as whether the packet should be dropped, mirrored, or forwarded out a physical port. The parser is defined in terms of a finite state machine, where each state may extract bits out of the packet and copy them into an instance before transitioning to another state.

The bulk of the actual packet processing occurs in the `ingress` control, which executes the `acl` and `nat` tables in sequence. The effect of executing these tables on a given packet is not determined by the P4 program but rather by the match-action rules that are installed by the control plane at run time. Each action comprises an imperative block of code that manipulates header and metadata instances using built-in primitive actions such as `modify_field` and `drop`. The trivial `egress` control does not modify any program state. The deparser is constructed by the compiler from the parser state machine, and emits each valid instance in order of dependency—i.e., `ethernet` followed by `ipv4`.

3 DATA-PLANE PROPERTIES

At a high level, one can classify the properties that a P4 programmer might wish to verify into three categories: general safety properties, architectural properties, and application-specific properties. This section discusses each of these categories using concrete examples as motivation.

General safety properties. P4 abstracts away many device-specific details, but the language makes trade-offs between safety and performance to ensure that programs can be executed efficiently on a variety of hardware and software targets. For example, a header instance can either be valid or invalid, and reading or writing an invalid header produces

an undefined result. Some targets might guarantee that headers are initialized with zeros, but the language specification does not mandate this behavior as it has a non-trivial cost on some devices. Rather, implementations are free to return an arbitrary result—e.g., random bits or the value of the same header from an earlier packet. While reading an invalid header seems innocuous, it can lead to serious bugs, such as causing the wrong rules to be matched in a table or leaking information from one packet to the next.

In our running example, reading invalid headers can lead to violations of the intended access control policy for the firewall. After the parser completes, the `ipv4` header may either be valid or invalid, depending on the value of the `ether_type` field. In particular, when the `acl` table reads the `ipv4.src_addr` and `ipv4.dst_addr` fields on a non-IPv4 packet, any outcome is possible. Even if the `acl` has been carefully populated with "whitelist" rules and a catch-all `drop` rule, other packets may be forwarded to internal hosts.

There are several ways we can repair the firewall to ensure that it never reads or writes invalid headers. For example, we can wrap the `ingress` control in a conditional:

```
if(valid(ipv4)) {
  apply(acl);
  apply(nat);
}
```

Alternatively, we can modify `acl` to read `ether_type`,

```
table acl {
  reads {
    ethernet.ether_type:exact;
    ipv4.src_addr:lpm;
    ipv4.dst_addr:lpm;
   }
   actions { allow; deny; }
}
```

and take care to populate the table with non-wildcard rules only when `ether_type` is `0x800`. We can capture this assumption by introducing an annotation:

```
assume read(acl,ethernet.ether_type) != 0x800
  implies wildcard(acl,ipv4.src_addr)
      and wildcard(acl,ipv4.dst_addr)
```

Such an annotation is part of the *control-plane interface*, which `p4v` uses to constrain the rules that may be legally installed in match-action tables. This idiom, where one of the values read by the table "guards" other values, occurs often in real-world programs. Control-plane interfaces may also capture constraints that span multiple tables.

Beyond header validity, there are several other basic safety properties that are critical for ensuring that programs have consistent and portable behavior. These properties include ensuring that header stacks are only ever accessed within statically declared bounds, that arithmetic operations do not overflow, and that the compiler-generated deparser emits all headers that are valid at the end of the egress pipeline. While these properties are straightforward to verify by hand in small programs, they can quickly become unmanageable in larger programs—imagine trying to reason about whether a given header is valid in the middle of thousands of lines of code that implement multiple layers of tunneling. Fortunately, because these properties can be checked using simple, local tests on program state, we can annotate programs with suitable checks automatically and verify them using `p4v`.

Architectural properties. In the abstract forwarding model used to execute P4 programs [5, 6], forwarding decisions are communicated from the `ingress` phase to the queuing and replication phase through standard metadata. For example, to indicate that the packet should be forwarded out on a particular physical port, the `ingress` control can set the `egress_spec` field to the value of that port. The queuing and replication engine takes the metadata and interprets it, performing actions such as dropping the packet, creating a clone, or moving the packet across to the `egress` control. However, it is easy to make mistakes such as specifying conflicting forwarding operations (e.g., drop and multicast), specifying operations that are unimplementable (e.g., recirculating the packet an excessive number of times), or forgetting to make a forwarding decision at all, letting the target decide what to do with the packet. There are also restrictions on certain metadata fields—e.g., in the `egress` control, `egress_port` is read-only and writes to it are silently ignored.

To ensure that P4 programs behave predictably and are portable across different targets, it is important to ensure that metadata is used correctly. For example, we typically want to ensure that the `ingress` control either assigns a value to the `egress_spec` field or invokes the `drop` primitive. However, due to the large variety of packets that arise in practical programs and the fact that tables are fundamentally nondeterministic, it can be easy to forget to specify the behavior on some control paths. For example, in the firewall program, if we address the issues related to header validity by wrapping both tables in a conditional statement, then packets that lack a valid `ipv4` header will not have a well-defined forwarding behavior. More subtly, the same issue can arise with packets that have an `ipv4` header. Specifically, consider what happens if the the packet misses in the `acl` table and executes the `nop` action in the `nat` table. As should be clear, we cannot address the problem in either case, without making assumptions about the forwarding rules that the control plane will eventually install in the match-action tables. For example, we could insist that every packet not blocked by the `acl` table must be rewritten by the `nat` table:

```
assume action(acl) != deny
  implies action(nat) = rewrite
```

As with safety properties, the `p4v` tool can automatically annotate programs with local tests that check for violations of these and other architectural properties.

Program-specific properties. Of course, there are also many properties that are important for ensuring the correctness of specific programs. For example, the designer of a switch might want to check that broadcast traffic is handled correctly, while the designer of a router might want to check that the IPv4 `ttl` field is correctly decremented on every packet. Or, in the firewall example, as discussed previously, we might want to prove that a given internal server is isolated from the rest of the network. To do this, we can add annotations to the `ingress` control in the program itself:

```
if (valid(ipv4)) {
  @pragma assume ipv4.dst_addr == D
  apply(acl);
  apply(nat);
  @pragma assert D == 10.0.0.99 implies drop
}
```

The first annotation records the value of the IPv4 destination address using a "ghost variable" `D`. The `p4v` tool allows the programmer to introduce logical variables that facilitate formal reasoning, provided they do not affect the execution of the program. In this case, `D` records the initial value of the destination address, prior to possible modifications by the `nat` table. The second annotation asserts that if the packet is destined for the internal server (identified by `10.0.0.99`), then it will be dropped after both tables are executed. Again, this property cannot be proven without making extra assumptions about the forwarding rules installed in those tables. For example, we could stipulate that the appropriate forwarding rule must be installed in the `acl` table:

```
assume reads(acl, ipv4.dst_addr) == 10.0.0.99
  implies action(acl) = deny
```

This annotation, which mentions the particular host needed to prove a program-specific property, is somewhat unusual. More commonly the control-plane interface consists of symbolic predicates that express generic constraints on the rules that may be installed in match-action tables.

4 VERIFICATION METHODOLOGY

This section outlines the techniques we use to verify P4 programs. We closely follow Dijkstra's classic approach to program verification based on predicate transformer semantics [11]. That is, we first build a first-order formula that captures the execution of the program in logic, leveraging the fact that P4 programs denote functions on finite sequences of bits (i.e., packets) parameterized on finite state (i.e., switch registers), and then use an automated theorem prover to check whether there exists an initial state that leads to a violation of one or more correctness properties. Although much of this approach is standard (e.g., it also underpins modern program verifiers such as Boogie [2] and Dafny [36]) we review it here for the sake of completeness, and to provide background for the extensions to this approach, which are discussed in later sections.

Variables	x	
Expressions	e	
Predicates	$P ::= e_1 = e_2$	*Equality*
	$\mid P_1 \wedge P_2$	*Conjunction*
	$\mid P_1 \vee P_2$	*Disjunction*
	$\mid P_1 \Rightarrow P_2$	*Implication*
	$\mid \neg P$	*Negation*
Commands	$c ::= x := e$	*Assignment*
	$\mid c_1;\ c_2$	*Sequence*
	$\mid c_1\ []\ c_2$	*Choice*
	$\mid$ *assume(P)*	*Assumption*
	$\mid$ *assert(P)*	*Assertion*

Figure 2: Guarded Command Language (GCL).

One of the key challenges we faced in building `p4v` is that the P4 language lacks a formal semantics. The language specification is generally well-written [44], but the precise meaning of many constructs is not entirely clear. For example, because P4 lacks a static type system, the meaning of arithmetic operations is not always well-defined: depending on its bit width, adding `x` to itself might either produce `2x` or a value less than `x` if the addition overflows. Worse, if `x` is a parameter to an action, its width may be truly arbitrary.

To address this challenge, we defined a translation from P4 programs to Guarded Command Language (GCL), an imperative language with non-deterministic choice (see Figure 2) [11]. We chose to define the semantics of P4 by translation, rather than developing an operational semantics [33, 42], for several reasons. First, because our semantics is defined by translation into a core language, it will be easy to add support for extensions and even new language versions, such as $P4_{16}$ [45]. To add support for a new language version, we simply have to update the front-end. Second, using GCL allows us to leverage decades of prior work on program verification, including optimizations that are critical for scaling performance to large programs such as `switch.p4`.

Our translation is defined in terms of a compositional, top-down traversal of the P4 program. It handles the full P4 language including parsers, controls, tables, and actions, as well as parser exceptions, parser value sets, action profiles, checksums, registers, and meters. The front-end works by first allocating state for each header and metadata instance, and then translating each parser, action, and control into a top-level imperative procedure. Although P4 parsers may contain loops, we unroll them, following the reference compiler, using a simple analysis to detect unproductive cycles that do not extract any headers from the packet. We use a variant of a standard type-inference algorithm to assign types to expressions [48], inserting casts to convert between

$$
\begin{array}{rcl}
wlp(x := e, Q) & \triangleq & Q[e/x] \\
wlp(c_1;\ c_2, Q) & \triangleq & wlp(c_1, wlp(c_2, Q)) \\
wlp(c_1\ []\ c_2, Q) & \triangleq & wlp(c_1, Q) \wedge wlp(c_2, Q) \\
wlp(assume(P), Q) & \triangleq & P \Rightarrow Q \\
wlp(assert(P), Q) & \triangleq & P \wedge Q
\end{array}
$$

Figure 3: Verification conditions for GCL.

boolean values and bit values and adjust widths and signs as appropriate. Whenever we encountered a P4 construct with unclear semantics, we discussed the intended behavior with key members of the open-source community to ensure that our interpretation was consistent with their expectations.

The most interesting aspect of the translation is the case for match-action tables. As tables are populated by the control plane, we do not know which action (if any) will be executed on the packet. Accordingly, we translate each table application into a non-deterministic choice between the actions declared for the table and, if the table doesn't declare a default action, a "no-op" action for when the table misses. For example, the translation of `apply(acl)` is the following:

$$assume(true)\ []\ allow()\ []\ deny()$$

The first command encodes a "no-op" operation for the case where the packet misses in the table.

We have successfully run our front-end on all the programs in the test suite distributed with the reference P4 compiler, among others. To ensure that the p4v front-end captures the intended semantics of P4, we developed a tool that symbolically executes the GCL code to generate input-output tests. We ran these tests on the P4 compiler and software simulator for Barefoot's Tofino chip. Hence, we have strong evidence that our "tested semantics" is consistent with existing implementations of P4 [23].

To verify the GCL code, we first compute a formula that captures the weakest constraints on the initial state that are sufficient to ensure that no assertion will fail, and we check whether the predicate is valid using the Z3 theorem prover [9]. If the formula is not valid, Z3 gives us a counter-example that we can convert into a concrete trace through the program [37]. We have found these counter-example traces invaluable when debugging large programs.

Figure 3 gives the formal definition of a function *wlp* (for weakest liberal preconditions [11]) that computes verification conditions for a GCL command c and postcondition Q, which is initially *true*. Most of the cases are intuitive: assignment substitutes the expression for the variable in the postcondition, sequential composition threads the postcondition through c_2 then c_1, and non-deterministic choice computes the conjunction of the weakest preconditions for c_1 and c_2. The cases for assumptions and assertions handle annotations used in program-specific properties such as the firewall example. Assumptions produce an implication from the assumed formula to the postcondition, while assertions conjoin the asserted formula with the postcondition. The verification conditions for a P4 program p are given by $wlp(c, true)$, where c is the translation of p into GCL.

Tables	t	
Actions	a	
Keys	k	
Expressions	$e ::= \ldots$	
	$\mid reach(t)$	*Reaches*
	$\mid reads(t, k)$	*Reads*
	$\mid wildcard(t, k)$	*Wildcarded*
	$\mid hit(t)$	*Hits*
	$\mid miss(t)$	*Misses*
	$\mid action(t)$	*Action*
	$\mid action\_data(t, a, x)$	*Action data*

Figure 4: Expressions used in control-plane interfaces.

5 CONTROL-PLANE INTERFACE

By itself, a P4 program does not fully specify the semantics of a data plane, which makes it impossible to fully verify many programs without additional knowledge of the control plane. One way to work around this problem is to delay verification until the forwarding rules are known. By combining the program and the forwarding rules, one obtains a deterministic program that can be verified. Indeed, a recently proposed tool based on symbolic execution follows this approach [19]. However, this approach has several drawbacks: it changes verification from a compile-time to a run-time task, and it requires repeatedly verifying the program every time the rules change, which would become expensive if done naively.

We follow a different approach in p4v: we constrain the behavior of the control plane using symbolic constraints in a *control-plane interface*. Figure 4 defines syntax for the additional expressions that can be used to define the control-plane interface. The expression $reach(t)$ is set to 1 if the execution reaches an application of t. The expression $reads(t, k)$ is set to the data-plane value read by t identified by k. Similarly, $wildcard(t, k)$ evaluates to 1 if the value identified by k is matched against an all-wildcard pattern. The expressions $hit(t)$ and $miss(t)$ evaluate to 1 if executing the table hits and misses respectively. Finally, the expression $action\_data(t, a, x)$ returns the value of the action data for parameter x in action a of table t.

Note that the control-plane interface is formulated using symbolic constraints on the data-plane execution—i.e., we do not need to specify the exact values of the forwarding rules that will be installed into the match-action tables at run time. In fact, we can go a step further and write down constraints that involve multiple tables. For example, we can stipulate

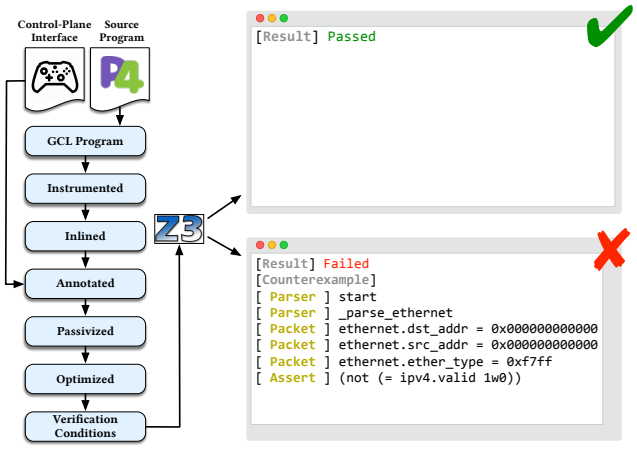

Figure 5: p4v system architecture.

that if table t hits, then table u must also hit, or that if table t executes action a, then table u must execute action b or action c. We used such multi-table assumptions in verifying properties of `switch.p4`—e.g., to rule out cases where an IPv4 packet processed by a table early in the pipeline is then processed using actions for IPv6 packets later in the pipeline.

A natural question to ask at this point is how these control-plane assumptions are integrated into the program. The next section presents the implementation in detail, but the high-level idea is as follows: we instrument the program with ghost variables to keep track of which tables and actions are executed, we translate the control-plane interface into a logical formula involving those ghost variables, and finally we predicate every assertion in the program on this formula.

A limitation of our approach is that we require programmers to write control-plane interfaces by hand. While it is likely that many interfaces could be automatically inferred, the tool does not currently provide this functionality. In addition, the control-plane interface is unverified code: overly constraining the control plane may ease data-plane verification but make it difficult or even impossible to implement the control plane. One way to guard against this situation is to statically check that the control-plane interface can be satisfied, and dynamically check that it is compatible with the rules actually installed at run time. For the latter task, we can use p4v itself, by combining the P4 program and the forwarding rules into a deterministic program as described earlier. In the future, we plan to explore more sophisticated approaches such as building an efficient run-time monitor or even statically verifying the control-plane code itself.

6 IMPLEMENTATION

This section discusses our p4v implementation, which comprises approximately 17,500 lines of OCaml code. Figure 5 shows the overall architecture, with each of the intermediate representations produced during verification of a program.

Parsing and type checking. The first phase of the p4v front-end parses and type-checks the P4 program. This phase is mostly standard, although some care is needed when inferring types—$P4_{14}$ is an untyped language, and yet the semantics of many arithmetic operations depends on the types of the operands. Our tool uses a standard algorithm to generate and solve type constraints [48]. We resolved ambiguities in the language specification by consulting with the developers of the open-source reference implementation of P4.

To gain further confidence in our semantics, we tested it using Barefoot's proprietary P4 compiler. We built a symbolic executor for GCL and combined it with Z3 to create a tool for generating packet tests and table configurations, similar to `p4pktgen` [43]. Some modifications to the GCL translation were required to model Tofino-specific features and primitives. We then used this tool to generate tests for a variety of programs, including a Tofino-specific variant of `switch.p4`. We ran the generated tests on a software model of the Tofino chip and checked that they passed.

Instrumentation. The next phase of the front-end instruments the program with "zombie" state that keeps track of information about the execution of tables and actions at run time.[1] The zombie state for each table records: whether the table was reached, which values were read by the table, whether the packet hit or missed, which action was executed, and which action data was supplied to the action. For example, given the following source program,

```
action a(x) { modify_field(m.g, x); }
table t {
 reads { m.f:exact; }
 actions { a; }
}
...
apply(t);
```

p4v generates the following instrumented program:

```
_p4v_zombie.reach_t := 1;
_p4v_zombie.reads_t := m.f;
{ /* Code for miss */
  _p4v_zombie.hit_t := 0; }
[]
{ /* Code for hit with action a */
  _p4v_zombie.hit_t := 1;
  _p4v_zombie.t_a_x := <?>;
  _p4v_zombie.action_t := 1;
  m.g := _p4v_zombie.t_a_x }
```

Here, <?> denotes an arbitrary "havoc" value, as the action parameter is supplied by the control plane and is unknown.

[1] We call this extra state "zombie" state because it is ghost state for the control plane, which is the "brains" of the network.

Inlining. The next phase uses a standard inlining algorithm to eliminate procedure calls and generate a GCL command that captures the semantics of the original P4 program. Inlining enables other optimizations and simplifies verification-condition generation but it can dramatically increase the size of the program, since it expands each procedure call into its body. Fortunately, by taking advantage of the domain-specific structure of P4 programs, we can avoid this blowup in some important cases. Recall the parser for the firewall example in Figure 1, which handles Ethernet and IPv4. If we naively inline the calls to `ingress`, we will end up with two copies of the code for the rest of the program. However, because the last statement in every parser state is a transition to another state (or an error handler), we can place one copy of the `ingress` code at the end of the `start` state, and allow the other paths to simply "fall through" to this code. This optimization significantly improves performance in programs with complex parsers.

Annotation. The next program transformation weaves the control-plane interface into the P4 program. The main challenge in doing this is overcoming the mismatch between the control plane's global, table-oriented perspective, and data plane's local, packet-oriented perspective. In particular, because the control-plane interface may contain constraints that involve multiple tables, weaving the constraints into the program is non-trivial. The `p4v` tool first converts the control-plane interface into a logical formula, using the ghost variables inserted during an earlier phase, and then predicates *every* assertion in the program on the resulting formula. For example, if I is the formula corresponding to the control-plane interface, then the translation maps an occurrence of assertion *assert(P)* to *assert($I \Rightarrow P$)*. The effect is to treat the control-plane interface as being in force at every program point. It follows that the programmer must not write constraints that are only valid at specific program points. This is not difficult to do in practice—e.g., we can predicate the constraint $action(t) == a$ on the assumption $reach(t)$.

Passivization. Careful readers may have noticed that the algorithm for generating verification conditions shown in Figure 3 is exponential in the worst case. The blowup results from the cases for assignment, which substitutes an expression for each copy of a variable in the predicate, and for choice, which contains two copies of the postcondition. A seminal paper by Flanagan and Saxe [16] developed an alternate algorithm that generates predicates that are only quadratic in the size of the program. The key insight behind their algorithm is to convert programs into "passive form," similar to single-static assignment, where every assignment is replaced with an assumption about the state of the program at that point. We rely on their efficient algorithm in `p4v`—indeed, it is critical to scale up to large programs.

Optimizations. The next phase implements several standard compiler optimizations, such as constant propagation and dead code elimination, to shrink the size of the program, and ultimately the size of the formula that must be passed to the SMT solver. These optimizations are key for improving the overall performance of `p4v`—e.g., checking header validity for `switch.p4` with these optimizations disabled did not terminate after 10 minutes. Note, however, that like the rest of the `p4v` front-end, these optimizations must be trusted—a bug could cause the tool to produce an incorrect result.

Verification conditions. The next phase takes the optimized, passive program and uses the Flanagan-Saxe algorithm to generate verification conditions, producing a single logical formula that we hand off to the Z3 theorem prover. To check whether the formula is valid, we ask Z3 if its negation is satisfiable. If not, then the program is guaranteed to be correct, because the weakest preconditions are valid. On the other hand, if it is satisfiable, then Z3 returns a model that provides a counter-example to the property being checked.

Counter-example generation. In the case where verification fails, the final step is to convert the model produced by Z3 back into a human-readable trace [37]. We use depth-first search on the program to find some assertion whose formula evaluates to false in the model, and then trace our steps backwards to populate the rest of the trace. We report the initial value of the packet headers, and the sequence of parser states and tables executed to reach the failed assertion. Figure 5 gives an example of a counter-example trace.

7 CASE STUDIES

This section presents our experiences verifying a variety of properties on a range of real-world programs using `p4v`.

7.1 Header validity for `switch.p4`

In the first case study, we verified that `switch.p4` never accesses a field of an invalid header. As discussed previously, this general safety property should hold in every program to avoid undefined behavior. It is also an interesting property to study because it requires reasoning about nearly every line of code in the program.

As background, `switch.p4` is a large program that consists of roughly 5600 lines of code and implements essentially all of the functionality found on a modern data center switch, including L2 switching, L3 routing, multicast, LAG, ECMP, tunneling, ACLs, MPLS, multi-device fabrics, and mirroring. These features can be selectively enabled or disabled to save resources, allowing for sets of feature *configurations*. We conducted our case study using the default configuration with in-band network telemetry disabled. This modified configuration has 58 parse states and 120 tables in total.

We used p4v to automatically insert an assertion before each read or write of a header field that checks whether the corresponding header instance is valid at that program point. For example, just before the following P4 statement,

```
modify_field(mpls[0].bos, 0x1);
```

p4v would insert the annotation,

```
assert valid(mpls[0].valid)
```

which checks that `mpls[0]` is valid at the point of access.

Verification took 2 minutes 48 seconds on an Intel Core i7-7500U laptop with 23 GiB RAM and required a control-plane interface with 143 distinct clauses (769 lines of code), along with 38 single-line pragmas to specify ghost variables for representing the validity of various headers at key points in the program. The annotations can be categorized as follows:

- *Default actions:* `switch.p4` does not assign default actions to many tables, but the control plane initializes all tables with a default action. Annotations describing default actions were needed for 31 of 120 tables.
- *Inter-device fabric traffic:* `switch.p4` assumes that inter-device fabric traffic is well-formed—e.g., if the packet has been classified as ordinary unicast, then none of the tunnel headers should be valid. 14 annotations express these well-formedness properties.
- *Table actions:* A P4 table supports all combinations of matches and actions, but the control plane rarely intends to use all combinations. 66 annotations explicitly disallow specific nonsensical action combinations.
- *Table reads:* Some tables match on the validity of a header while at the same time performing a ternary match on one of its fields. This could result in an undefined read unless the ternary match has "don't care" bits in every rule where validity is `false`. 10 annotations fall in this category.
- *Action data:* 14 annotations stipulate that only certain values will be supplied as parameters to actions.

The remaining annotations correspond to bugs in `switch.p4`, where each additional annotation is akin to an XFAIL, describing known bad behavior. In total, we found 10 bugs in `switch.p4`. Two were parser bugs, which parsed packets that were not supported by the rest of the pipeline. For example, the parser allowed L3 Geneve tunnels (which do not have an inner Ethernet header), whereas the tunnel-decapsulation code assumed L2 Geneve tunnels only and unconditionally copied the `inner_ethernet` header. One bug was an order-of-operations error, in which fields were modified in a header before the header was added. Two bugs were in tables that incorrectly permitted the `nop` action to be taken. Another bug was in the actions for terminating L3 MPLS tunnels, which erroneously read the `inner_ethernet` header. Three bugs correspond to multi-table constraints that the designers of `switch.p4` believe hold, but do not see how to enforce using the control plane. The final bug was found in which a table read invalid state in its match key.

Overall, the control-plane interface for switch was developed by a single programmer working for approximately three days in aggregate, spread over two weeks. All told, we found the annotation burden reasonable. The annotations total roughly 14% of the lines of P4 code, but this represents only a very small fraction of the tens of thousands of lines of code that make up the control plane. Moreover, two p4v features greatly eased the burden: fast verification times and intuitive counterexample traces, which quickly pointed the way to identifying missing control-plane assumptions.

7.2 NetCache parser roundtripping

In the second case study, we attempted to verify an important architectural property for NetCache, a program that implements an in-network key-value store on a P4-programmable target [29]. Many P4 targets, including PISA [5, 6], deparse the headers into a byte stream at the end of the ingress pipeline, and then reparse the byte stream back into headers after replication and queuing and before executing the code in the egress pipeline. To prevent data from being corrupted or lost, it is important that the programmer-specified parser and compiler-generated deparser compose to the identity function. For example, if the IPv4 header is removed in the ingress pipeline, but the programmer neglects to set the EtherType field to a new value, then the egress parser will attempt to populate the IPv4 header using bits from another header or the packet payload, producing a mangled result.

We used p4v to automatically insert assertions into NetCache to check that the information in each header is correctly preserved when processed using the deparser and the parser. For example, at the end of the ingress pipeline, p4v inserts assumptions to record the validity of the Ethernet header and the values of its fields as ghost variables:

```
assume _p4v_roundtrip.ethernet.valid
  iff valid(ethernet);
assume _p4v_roundtrip.ethernet.dst_addr ==
  ethernet.dst_addr;
assume _p4v_roundtrip.ethernet.src_addr ==
  ethernet.src_addr;
assume _p4v_roundtrip.ethernet.ether_type ==
  ethernet.ether_type;
```

At the start of the egress pipeline, it inserts the following assertions to check that the Ethernet header is preserved on the round-trip through the deparser and the parser.

```
assert valid(ethernet)
  iff _p4v_roundtrip.ethernet.valid == 1;
assert valid(ethernet)
  implies _p4v_roundtrip.ethernet.dst_addr ==
      ethernet.dst_addr
  and  _p4v_roundtrip.ethernet.src_addr ==
      ethernet.src_addr
  and  _p4v_roundtrip.ethernet.ether_type ==
      ethernet.ether_type;
```

```
action set_mirror_bd(bd) {
 modify_field(egress_metadata.bd, bd);
}
table mirror {
 reads { i2e_metadata.mirror_session_id : exact; }
 actions {
  nop;
  set_mirror_nhop;
  set_mirror_bd;
 }
}
action outer_replica_from_rid(bd, ...) {
  modify_field(egress_metadata.bd, bd); ...
}
action inner_replica_from_rid(bd, ...) {
 modify_field(egress_metadata.bd, bd); ...
}
table rid {
 reads { intrinsic_metadata.egress_rid: exact; }
 actions {
  nop;
  outer_replica_from_rid;
  inner_replica_from_rid;
 }
}
```

Figure 6: Action dependency between tables.

We wrote a control-plane interface with 93 annotations that specify the default actions of tables. This interface, which was derived in a few minutes from the static control-plane provided with NetCache, was sufficiently strong to establish header validity, with one exception. It turns out the NetCache header may be invalid after parsing, but is always accessed at the start of the `ingress` pipeline. We added an extra annotation to limit verification to packets with a valid NetCache header.

Unfortunately, the round-tripping property fails to hold for NetCache. It took p4v 3 minutes 35 seconds to discover this and print a counter-example on an Intel Core i7-7500U laptop with 23 GiB RAM. NetCache uses a packet format with standard Ethernet, IPv4, and UDP headers, followed by a custom header containing an op-code and key, and finally an optional header containing a value. Together, these headers encode operations such as $get(k)$ and $put(k, v)$. The root of the bug is in the P4 implementation of the $put(k, v)$ operation: the code correctly writes the value v into stateful registers and invalidates the optional header but it fails to update the op-code. Hence, the egress parser attempts to re-parse the optional value header. If the packet is sufficiently long then it will extract the required bits from the payload, but if it is not, then the parser will transition to an error state and the packet will be dropped. We reported this issue to the NetCache developers who confirmed it is indeed a bug.

7.3 NetPaxos bug

In the third case study, we analyzed an application-specific property of NetPaxos [8], a P4 implementation of the Paxos consensus protocol [35]. As originally discovered by the developers of P4-Assert [19], the published P4 implementation of NetPaxos contains a serious bug: the action that compares the round number from the arriving packet with the round number stored at the switch sets the drop flag of the arriving packet by default, under the assumption that the packet should be dropped. However, the code does not reset the drop flag if the round number in the packet is greater than the stored round number. As a result, consensus is not reached.

We were able to identify the root cause of this bug adding an assumption and assertion that, together, state that if the Paxos header is valid, and the round number of the packet is greater than the round number at the switch, then the packet should not be dropped.

```
assume valid(paxos)
  implies local_metadata.round <= paxos.rnd
assert valid(paxos)
  implies local_metadata.set_drop == 0
```

On an Intel Core i7-7500U laptop with 23 GiB RAM, it took p4v 159 milliseconds to produce a counter-example trace that exercises the bug. We also confirmed that the bug *is present* by inverting the assertion:

```
assert valid(paxos)
  implies local_metadata.set_drop == 1
```

This case study illustrates another use case for p4v: assert-style debugging. Hardware targets don't typically come with debuggers, making it difficult to diagnose the root causes of observed anomalies. By adding an assertion that captures the correct (non-anomalous) behavior of a program with a bug, we can use p4v to generate a counterexample trace. In the case of NetPaxos, the first counterexample returned by p4v indicated a missing control-plane annotation, and the second counterexample identified the actual bug.

7.4 Enabling compiler optimizations

In the fourth case study, we explored how p4v can be used to enable compiler optimizations that would be difficult or impossible to implement using traditional analysis techniques. Table placement is one of the key tasks performed by a P4 compiler, which attempts to maximize parallelism while respecting data dependencies. However, since the compiler lacks information about how the tables will be populated with rules by the control plane, it is often difficult to determine which dependencies are genuine and which are spurious. Using p4v and a suitable control-plane interface, we can verify that certain apparent dependencies are spurious.

To illustrate, consider Figure 6, which is based on code from `switch.p4`. The `mirror` and `rid` tables each contain an action that modifies metadata for the L2 bridge domain (`egress_metadata.bd`). In the absence of a control-plane interface, the compiler conservatively assumes that the tables may execute any of their actions, and allocates them in different stages. But assuming these tables are configured so

Program	LOC	Parse states	Tables	Total header length (bytes)	Uses state	No CP interface			Control-plane interface				
						Time (mm:ss.s)	Memory (MiB)		#	Lines	Time (mm:ss.s)	Memory (MiB)	
							p4v	z3				p4v	z3
simple_router	63	3	1	34	✗	0.05	30	15	0	0	—	—	—
calc	304	3	1	30	✗	0.06	31	15	0	0	—	—	—
easyroute	53	3	1	13	✗	0.06	30	16	1	2	0.06	30	15
flowlet_switching	246	4	7	54	✓	0.06	31	15	0	0	—	—	—
dead_drop	469	3	2	32	✓	0.07	32	15	0	0	—	—	—
paxos	205	5	4	90	✓	0.07	32	18	0	0	—	—	—
fox_fastflow	498	4	2	42	✓	0.07	33	16	1	1	0.11	33	20
vpc	272	6	10	68	✗	0.08	31	17	4	9	0.08	31	17
axon	99	6	2	29	✗	0.15	32	24	3	9	0.09	32	16
tor	456	10	13	96	✓	0.18	34	23	10	50	0.15	34	19
fox_2110	506	6	1	62	✓	0.19	34	29	1	1	0.20	34	29
netcache	538	17	96	187	✓	0.20	36	25	9	17	0.20	36	19
stful	1,167	10	27	58	✓	0.21	37	32	0	0	—	—	—
linear_road	846	14	24	59	✓	0.23	34	43	1	5	0.19	34	40
nat	293	5	6	65	✗	0.50	32	115	3	9	0.43	32	102
prog1	530	28	1	124	✗	3.41	129	43	0	0	—	—	—
ndn	495	31	7	2,357	✓	3.42	246	15	0	0	—	—	—
tlv_parsing	189	8	1	92	✗	6.49	174	89	3	8	6.42	190	90
prog2	536	7	10	78	✗	16.77	42	370	2	13	35.15	42	377
dapper	605	11	17	78	✓	49.35	1,051	527	2	3	49.34	1,051	522
switch	5,599	58	120	182	✓	2:36.11	444	2,761	143	769	2:47.51	576	1,521
prog3	929	33	5	140	✗	19:09.03	488	2,688	—	—	—	—	—
hyperp4	11,173	16	537	100	✗	—	179	(OOM)	—	—	—	—	—

Figure 7: Experimental results conducted on an Intel Core i7-7500U laptop with 23 GiB RAM. The standard error for each result is within 5% of the reported mean.

that replication and mirroring are only applied to L3 packets simultaneously, it is safe to place them in the same stage.

Formally, we can capture the absence of an action dependency as follows:

```
assert not
 ((action(rid) == inner_replica_from_rid or
   action(rid) == outer_replica_from_rid) and
   action(mirror) == set_mirror_bd)
```

Intuitively, this assertion states that no packet may be processed using actions from `rid` and `mirror` that modify the same L2 bridge domain metadata. Next, we annotate the program with ghost variables to record the values of `egress_rid` and `mirror_session_id` at the start of the egress pipeline:

```
assume
  R == intrinsic_metadata.egress_rid and
  M == i2e_metadata.mirror_session_id
```

Finally, we add a constraint stipulating that the `mirror` table must not apply its L2 action (`set_mirror_bd`) to packets that are both replicated and mirrored—i.e., to L3 packets:

```
assume
  (R != 0 and
   M == reads(mirror,
              i2e_metadata.mirror_session_id))
  implies
    not(action(mirror) == set_mirror_bd)
```

With these annotations, p4v can verify that the actions are disjoint, allowing it to optimize the placement of the tables. We plan to integrate p4v with the P4 compiler to optimize resources such as storage for header instances in future work.

8 EVALUATION

To evaluate the performance of p4v, we conducted experiments on a diverse collection of open-source and proprietary programs that vary in size and complexity. These programs implement a wide range of functionality, including conventional forwarding, source routing, data center routing, content-based networking, performance monitoring, complex packet parsing, and in-network processing. We developed a control-plane interface for all but two programs, and verified the header validity property—i.e., during every execution of the program, is every header valid when it is read or written. We believe that header validity is a good property for benchmarking as it is a global property that requires reasoning about nearly all control-flow paths.

The results of our experiments are given in Figure 7. We report running times and memory usage with and without control-plane interfaces, along with statistics about the programs: lines of code, parser states, match-action tables, total header lengths, use of stateful features, and total annotations

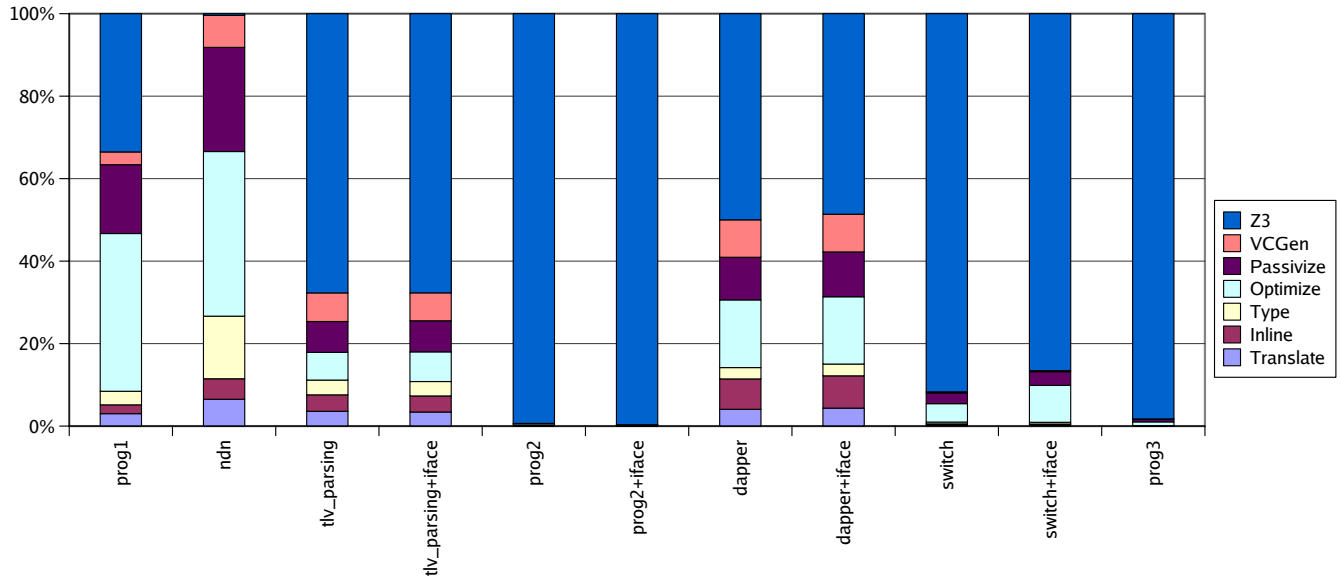

Figure 8: Performance breakdown by phase of verification for benchmark programs that required more than one second of running time to complete.

and lines of code in control-plane interfaces. All tests were done on an Intel Core i7-7500U laptop with 23 GiB RAM. Each experiment consisted of ten trials, and the results have standard error within 5% of the reported means.

The first point to notice is that most programs can be checked in under a minute—and in well under a second for many. For those programs that took more than a second, Figure 8 depicts the performance breakdown by each phase of verification. The programs with the longest running times are `switch`, `dapper`, and `prog3`, each of which are large and have complex state and control-flow. Another observation is that adding annotations can increase verification time. We believe there are two main reasons for this: (i) adding annotations increases the size of the program, and hence the size of the formula passed to Z3, and (ii) showing that a formula is unsatisfiable generally takes longer than showing that it is satisfiable—the former requires considering all possible models, while the latter only requires finding a single model of the formula. Finally, we note that our current `p4v` prototype does have some limitations: the tool failed to complete on HyperP4 [24], a complex program that implements virtual data planes, as Z3 ran out of memory.

Overall, these results show that `p4v` scales to real-world programs and provides a sufficiently high level of performance that it could be used in everyday work.

9 RELATED WORK

There are now more than 50 years of research on software verification. Hence, we are truly "standing on the shoulders of giants" such as Hoare [27] and Dijkstra [11], as well as recent tools such as ESC-Java [15], Boogie [2], and Dafny [36]. The main conceptual innovation in our tool is the use of zombie state to track assumptions about the control-plane interface to a P4 program. This approach can be seen as an instance of more general techniques for reasoning about unknown behavior in program analysis tools (e.g., see the article by Dillig et al. [12] for an overview), but we exploit the fact that P4 programs are loop-free.

Early work by Xie et al. [54] proposed the idea of network verification and developed techniques based on computing the transitive closure of transfer functions to statically analyzing device configurations to check reachability properties. HSA [32] and Veriflow [34] later applied this methodology to software-defined networking data planes, and developed optimized data structures to represent transfer functions to enable verification to scale. NetKAT [1] emerged from an earlier effort to develop a machine-verified implementation of the language in Coq [22] but added a sound and complete decision axiomatization and a decision procedure based on an automata representation [18]. Recent work in the area has focused on optimizations based on clever data structures [4], atomic predicates [55], and exploiting symmetry [47].

Control-plane verification is fundamentally more difficult than data-plane verification. Intuitively, a control plane can be thought of as generating a sequence of data planes, each of which must be verified. In addition, control-plane protocols typically have complex policies to facilitate interactions between multiple autonomous systems. Early work

on RCC [14] focused on finding bugs in BGP configurations but was neither sound nor complete. Recent tools such as Batfish [17], ARC [20], and Minesweeper [3] have developed efficient heuristics and abstract representations of control plane that allow verification to scale to much larger problems. Bagpipe [53] developed a mechanized semantics of BGP.

Middlebox verification is fundamentally more difficult than data-plane verification due to the pervasive use of state. The problem is undecidable in the general case but many common topologies and network functions remain tractable [52]. Recent work has focused on scaling verification using techniques such as abstract interpretation [13, 46, 50, 56]. P4 programs have bounded state, which keeps verification tractable.

Several other recent projects propose semantics and verification techniques for P4 programs. Early work by Lopes et al. developed an operational semantics for P4 and developed a tool based on Datalog for automatically verifying safety properties and checking program equivalence [42]. Subsequent work by Kheradmand and Rosu developed a complete operational semantics for P4 in the the K framework and proposed applications including a symbolic model checker and deductive verification tool [33]. Nötzli et al. developed `p4pktgen`, a tool that uses symbolic execution to generate an exhaustive set of input-output tests for a P4 program [43]. Freire et al. developed `p4-assert`, which translates P4 to a C-like representation and then uses Klee to symbolically execute the resulting program [19]. Finally, Stoenescu et al. developed Vera [49], which uses SymNet [50] a symbolic execution framework that uses network-specific algorithms and data structures, to verify P4 programs efficiently. A recent empirical study found that symbolic execution often outperforms tools based on verification condition generation, except when the program being verified is "branchy" [30]. P4 programs typically have dense conditional structure.

10 CONCLUSION

This paper presented `p4v`, a practical tool for verifying P4 data plane programs. It demonstrated that `p4v` scales to large programs and finds bugs in real-world implementations. In the future, we plan to investigate a number of follow-on questions. We would like to extend the P4 language with domain-specific constructs for specifying control-plane interfaces. Such constructs might provide intuitive abstraction for specifying properties such as "these tables are accessed by disjoint sets of packets" or even "these applications never generate conflicting rules." We are also interested in exploring applications of program synthesis—e.g., automatically generating control-plane interfaces from example traces. Finally, we are interested in exploring questions lower down the stack, building verified compilers for P4 and even verified hardware, as well as higher up the stack, investigating whether we can effectively map down to verified P4 implementations from high-level descriptions of network behavior, such as state machines and message sequence diagrams.

ACKNOWLEDGMENTS

We thank Michael Attig, Antonin Bas, Chris Dodd, Vladimir Gurevich, Theo Jepsen, Changhoon Kim, Prathima Kotikalapudi, Ramkumar Krishnamoorthy, Xin Jin, and Dan Lenoski for productive discussions related to the design of `p4v` and for assistance with the P4 compiler and Tofino software simulator. We thank Amin Vadhat, Steffen Smolka, George Varghese, participants at IFIP WG 2.8 Asilomar, the anonymous SIGCOMM reviewers, and our shepherd Costin Raicu for many helpful suggestions.

REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *POPL*. 113–126.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A modular Reusable Program Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*. 364–387.

[3] Ryan Becket, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *SIGCOMM*. 155–168.

[4] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. 2016. ddNF: An Efficient Data Structure for Header Spaces. In *Haifa Verification Conference*. 49–64.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* 44, 3 (July 2014), 87–95.

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *SIGCOMM*. 99–110.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*. 209–224.

[8] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *SIGCOMM CCR* 46, 2 (May 2016), 18–24.

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[10] Leonardo de Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: Introduction and applications. *CACM* 54, 9 (2011), 69–77.

[11] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. *CACM* 18, 8 (1975), 453–457.

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Reasoning About the Unknown in Static Analysis. *CACM* 53, 8 (2010), 115–123.

[13] Mihai Dobrescu and Katerina Argyraki. 2015. Software Dataplane Verification. *CACM* 58, 11 (2015), 113–121.

[14] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *NSDI*. 43–56.

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for

Java. In *PLDI*. 234–245.
[16] Cormac Flanagan and James B. Saxe. 2001. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *POPL*. 193–205. https://doi.org/10.1145/360204.360220
[17] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. 2015. A General Approach to Network Configuration Analysis. In *NSDI*. 469–483.
[18] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. 2015. A Coalgebraic Decision Procedure for NetKAT. In *POPL*. 343–355.
[19] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *SOSR*. 4:1–4:7.
[20] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*. 300–313.
[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. 653–669.
[22] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-Verified Network Controllers. In *PLDI*. 483–494.
[23] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *ECOOP*. 126–150.
[24] David Hancock and Jacobus Van der Merwe. 2016. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *CoNEXT*. 507–508.
[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving practical distributed systems correct. In *SOSP*. 1–17.
[26] Mukesh Hira and LJ Wobker. 2015. Improving Network Monitoring and Management with Programmable Data Planes. P4 Language Consortium Blog. Available at https://p4.org/p4/inband-network-telemetry/.
[27] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *CACM* 12, 10 (1969), 576–580.
[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*. 35–49.
[29] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*. 121–136.
[30] Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff. 2012. Comparing Verification Condition Generation with Symbolic Execution: An Experience Report. In *VSTTE*. 196–208.
[31] Peyman Kazemian. 2017. Network path not found? Forward Networks Blog. Available at https://bit.ly/2FzpEEZ.
[32] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI*. 113–126.
[33] Ali Kheradmand and Grigore Rosu. 2018. P4K: A Formal Semantics of P4 and Applications. https://arxiv.org/abs/1804.01468.
[34] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*. 15–29.
[35] Leslie Lamport. 1998. The Part-time Parliament. *TOCS* 16, 2 (1998), 133–169.
[36] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artifical Intelligence, and Reasoning*. 348–370.
[37] K Rustan M Leino, Todd Millstein, and James B Saxe. 2005. Generating error traces from verification-condition counterexamples. *Science of Computer Programming* 55, 1-3 (2005), 209–226.
[38] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.
[39] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *SIGCOMM*. 290–301.
[40] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a verified relational database management system. In *POPL*. 237–248.
[41] Nick McKeown, Timon Sloane, and Jim Wanderer. 2017. P4 Runtime–Putting the Control Plane in Charge of the Forwarding Plane. Available at http://bit.ly/2It6Ecn.
[42] Nick McKeown, Dan Talayco, George Varghese, Nuno Lopes, Nikolaj Bjorner, and Andrey Rybalchenko. 2016. *Automatically verifying reachability and well-formedness in P4 Networks*. Technical Report. Microsoft Research. http://bit.ly/2lxFVSW
[43] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. p4pktgen: Automated Test Case Generation for P4 Programs. In *SOSR*. 5:1–5:7.
[44] P4 Language Consortium. 2017. *P4 Language Specification, Version 1.0.4*. Available at https://p4.org/specs/.
[45] P4 Language Consortium. 2017. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.
[46] Aurojit Panda, Ori Lahav, Katerina J Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths.. In *NSDI*. 699–718.
[47] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *POPL*. 69–83.
[48] François Pottier and Didier Rémy. 2005. *Advaned Topis in Types and Programming Languages*. MIT Press, Chapter The Essence of ML Type Inference, 389–489.
[49] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with Vera. In *SIGCOMM*.
[50] Radu Stonescu, Matei Popovici, Lirina Negranu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *SIGCOMM*. 314–327.
[51] tofino 2015. Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.
[52] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. 2016. Some Complexity Results for Stateful Network Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*. 811–830.
[53] Konstanin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations With an SMT Solver. In *OOPSLA*. 765–780.
[54] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *IEEE INFOCOM*. 2170–2183.
[55] Hongkun Yang and Simon S. Lam. 2013. Real-time Verification of Network Properties Using Atomic Predicates. In *IEEE ICNP*.
[56] Arseniy Zaostrovnykh, Solal Pirelli, Luis David Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. *SIGCOMM* (2017), 141–154.