Uncovering Bugs in P4 Programs with Assertion-based Verification

Lucas Freire*, Miguel Neves*, Lucas Leal*, Kirill Levchenko†
Alberto Schaeffer-Filho*, Marinho Barcellos*
*UFRGS †UC San Diego

ABSTRACT

Recent trends in software-defined networking have extended network programmability to the data plane through programming languages such as P4. Unfortunately, the chance of introducing bugs in the network also increases significantly in this new context. Existing data plane verification approaches are unable to model P4 programs, or they present severe restrictions in the set of properties that can be modeled. In this paper, we introduce a data plane program verification approach based on assertion checking and symbolic execution. Network programmers annotate P4 programs with assertions expressing general security and correctness properties. Once annotated, these programs are transformed into C-based models and all their possible paths are symbolically executed. Results show that the proposed approach, called ASSERT-P4, can uncover a broad range of bugs and software flaws. Furthermore, experimental evaluation shows that it takes less than a minute for verifying various P4 applications proposed in the literature.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Software and its engineering** → **Software verification and validation**;

KEYWORDS

P4; Verification; Programmable Data Planes

1 INTRODUCTION

Data plane programmability allows operators to quickly deploy new protocols and develop network services. Through programming languages such as P4 [2], it is possible to specify in a few instructions how packet headers should be manipulated by different forwarding devices in the infrastructure. Despite the flexibility, this paradigm also increases the chance of introducing bugs into the network.

Several tools have been developed in order to check if a given network configuration satisfies a set of intended properties [7, 21, 22, 25]. However, they are either unable to model P4 programs or cannot reason about program-specific properties. In this paper, we propose a network verification technique capable of modeling and checking (at compile time) general security and correctness properties of P4 programs[1]. Called ASSERT-P4, it provides an expressive assertion language allows programmers to specify their intended properties by simply annotating their P4 programs. Its language allows the specification of both location-restricted and location-unrestricted invariants. For example, verifying that packets marked to be dropped at a specific point of the code (location-restricted) are not eventually forwarded (location-unrestricted). Once annotated, a program is symbolically executed, with assertions being checked while all its paths are traversed.

We built a prototype of ASSERT-P4 using KLEE [3] and the P4 Reference Compiler [18]. To evaluate our approach, we tested it on four real P4 applications [5, 12, 19, 24] collected from the literature, and found correctness bugs in the first three.

Our results show that ASSERT-P4 can uncover a broad range of bugs and software flaws, either in a data plane program itself or in its control plane configuration. A detailed performance analysis also shows that, although the verification time grows exponentially with the number of tables and assertions, ASSERT-P4 needs less than a minute to verify various P4 applications [5, 11, 12, 14, 20, 23].

2 MOTIVATING EXAMPLES

A P4 program is a collection of domain-specific constructs such as headers, metadata, parsers, actions, tables, control blocks, and extern objects. Some basic data types for representing and manipulating packets (e.g., *packet_in* and *packet_out*) are also defined in the P4 specification [27]. Bugs in P4 programs can result from, e.g., erroneous assignments, poor logic or control misconfiguration. Next, we present two motivating examples to better explain the type of problems we face, and the difficulty of finding them.

Code circumvention. Figure 1(a) shows an example of a vulnerability stemming from a logic error in a P4 program. This code snippet specifies a packet processing pipeline containing two match-action tables (*tcp_table* and *acl_table*), invoked inside an *ingress* control block. While one would reasonably expect *acl_table* to be applied to both TCP and UDP traffic, UDP packets can bypass this filtering mechanism. Considering that the network security policy disallows this type of practice, the program in question could be used as a starting point for many attacks (e.g., UDP flooding). Even though correcting this problem is simple (moving the table that implements the access control list outside the conditional structure is enough), finding it may not be trivial in large and complex programs.

Control misconfiguration. Many faults in networks arise from bugs in forwarding rules (i.e., control plane configurations). In this

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185499

[1] A poster of an early version of this work appeared at the ACM Conference on Communication Security [10].

```
1 control ingress() {
2   apply {
3     ...
4     if (headers.ip.nextHeader == TCP) {
5       tcp_table.apply();
6       acl_table.apply();
7     }
8     ...
9   }
10 }
```

(a) **Code circumvention**: ACL filtering is only applied to TCP packets, allowing other transport layer protocols (e.g. UDP, SCTP) to pass through.

```
1 table mirror {
2   key = { metadata.egress_port : exact; }
3   actions = { NoAction; clone_packet; }
4   default_action = NoAction;
5
6   const entries = {
7     0x00000001 : clone_packet(0x00000002);
8     0x00000002 : clone_packet(0x00000002);
9   }
10 }
```

(b) **Control misconfiguration**: The forwarding rule in line 8 is cloning packets to the same egress port as the original packet.

Figure 1: Examples of bugs in P4 programs

sense, Figure 1(b) shows an example of a data plane program whose tables are erroneously configured at compile-time. The *mirror* table clones packets based on their output port (line 2), setting a new port for cloned packets based on its action parameters. In this example, one of the forwarding rules is assigning the output port of the cloned packet to the same value as the original packet (line 8). As a consequence, both packets will be sent to the receiver.

3 SYSTEM DESIGN

3.1 Overview

This section provides an overview of ASSERT-P4, which is built on two key ideas: (i) using assertions for specifying properties about P4 programs; and (ii) verifying models derived from annotated programs. The former allows programmers to easily express their intended properties, while the latter enables programs to be efficiently verified. Using models to represent real programs is a common practice in the verification literature [8, 21, 22, 26], and although formally proving our models are equivalent to their original programs is an ongoing effort, we manually checked all the results presented in this paper.

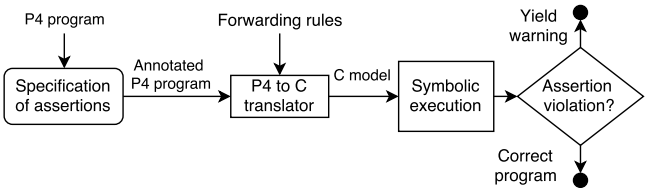

Figure 2: Overview of ASSERT-P4.

Figure 2 shows the ASSERT-P4 workflow. The P4 developer first annotates his code with assertions expressing general properties of interest. These properties can reflect a network security policy or simply represent the expected program behavior. Once annotated, the P4 program is translated into a C-based model. During this process, forwarding rules can be optionally added as input to the translator for restricting the verification to a given network configuration. The generated model is finally checked by a symbolic execution engine, which tests all its execution paths looking for assertion failures. If no assertion is violated during this process, the P4 program is considered correct with respect to the analyzed properties. Otherwise, the violation is reported, allowing the developer to correct the program. In the following sections, we describe in detail each step of the ASSERT-P4 verification process.

3.2 Specifying assertions

Programmers use assertions to express properties of P4 programs. To this end, we provide an assertion language specially designed to capture packet processing behaviors (e.g., header extraction and emission, packet dropping and forwarding) and facilitate the task of specifying complex networking properties. The purpose of our language is not to innovate on syntax, but to simplify the way network programmers encode properties in a programmable data plane. In fact, we draw inspiration from Beckett et al. [1] to define primitives capable of minimizing the number of language elements, while still being expressive.

Figure 3 summarizes the grammar of our assertion language. Although it resembles C-style assertions found in traditional programming languages, our concept of assertion is a bit more general, in the sense it can involve both location-restricted and location-unrestricted elements. A location-restricted element is one that tests the value of a program variable at a specific location (i.e., where the assertion is specified), as in traditional programming languages like C or Java. A location-unrestricted element, in contrast, tries to capture the evolution of the program and how it manipulates its variables (i.e., packet headers in this case) along its execution as a whole. Examples are provided at the end of this section.

Syntactically, each assertion is composed of a boolean expression b, which may include primitive methods m. The set of allowed methods is {*forward, traverse_path, constant, if, extract_header, emit_header*}. Both expressions and methods can operate over one or more values v, header fields f or headers h. Semantically, each assertion represents a boolean that should evaluate to true or false, where values and header fields evaluate to true if they are non-zero and false otherwise. Integer expressions i have the same semantics as their equivalents in the P4 language, as well as boolean expressions, which include the equality and inequality relational operators to compare logical values.

Each method m has its own meaning. Specifically, *forward* returns true when the packet being processed is not dropped by the program. *traverse_path* indicates if a given structure in the program

```
b ::= v          m ::= forward
   | f              | traverse_path
   | m              | constant(f)
   | !b             | if(b, b, [b])
   | b || b        | extract_header(h)
   | b && b        | emit_header(h)
   | b == b     i ::= v
   | b != b         | f
   | i >= i         | i * i
   | i <= i         | i / i
   | i < i          | i % i
   | i > i          | i + i
   | i == i         | i - i
   | i != i
```

Figure 3: Assertion language grammar.

(e.g., an action) was executed. *constant(f)* is true if the field f is not changed from the assertion location to the end of the program execution. *if(b_1, b_2, [b_3])* is similar to traditional conditional statements (i.e., if the condition represented by expression b_1 is true, then the expression b_2 will be evaluated, otherwise the optional expression b_3 will be verified). *extract_header(h)* is true if the header h is extracted from the packet during the *parsing* process. Finally, *emit_header(h)* returns true if the outgoing packet contains the header h at the end of the program execution.

Figure 4 shows an example containing an annotated P4 program, where assertions are in bold. Due to space restrictions, only the most relevant parts of the program are displayed. This program describes a packet processing pipeline with a single table (*dmac*), which is instantiated inside the *TopPipe* control block. Each entry of this table can invoke one of two actions (*Drop* or *Set_dmac*). The annotated assertions aim to verify that: (i) packets marked to drop are never forwarded (line 7), and (ii) only packets with TTL greater than zero are forwarded (line 21). The two assertions contain both location-unrestricted elements (e.g., the method $forward$ captures the state of the program at the end of its execution) and also location-restricted ones (e.g., the expression "$headers.ip.ttl > 0$" tests the value of $headers.ip.ttl$ at the point in which the assertion is inserted).

3.3 Constructing C models

Once a P4 program is annotated, our tool generates an equivalent program in the C language through a translation process. This section describes how we designed this process in detail, taking the main P4 structures as a base (i.e., headers, tables, actions, parsers, control blocks, and external objects). Figure 5 exemplifies the translation process.

Headers. Given their similar representations, P4 headers are properly modeled by *structs* in C. Each header field is mapped to a *struct* member, and *bit fields* in C are used to match between the size of the header field and the size of its corresponding member in the generated *struct*. Each basic type in P4 is mapped to a corresponding type in C, based on its declared size. Fields with more than 64 bits can be modeled using bit arrays.

Tables. Each table in a P4 program is modeled as a function in C. Functions created from tables are constructed in different ways depending whether the forwarding rules are supplied to the translator or not. If the rules are provided, the *match* fields in the P4

```
1  ...
2 control TopPipe(inout Parsed_packet headers,
3                 out OutControl outCtrl) {
4  ...
5  action Drop {
6    outCtrl.outputPort = DROP_PORT;
7    @assert("if(traverse_path, !forward)");
8 }
9  action Set_dmac(EthernetAddress dmac) {
10   headers.ethernet.dstAddr = dmac;
11 }
12 table dmac {
13   key = { nextHop : exact; }
14   actions = { Drop; Set_dmac; }
15   default_action = Drop;
16 }
17 apply {
18   ...
19   dmac.apply();
20   ...
21   @assert("if(forward, headers.ip.ttl > 0)");
22 }
23 }
```

Figure 4: Example of an annotated P4 program.

table are tested against their corresponding rule values using the specified matching approach (e.g., exact, ternary or longest-prefix match). Otherwise, the decision of which action to execute is made based on a symbolic value specially declared to force the creation of multiple execution paths by the symbolic engine (one for each action listed in the table). To avoid conflicts caused by tables from different scopes having the same name, we append an ID to their names. This solution is also applied in any situation where name conflicts may be an issue (e.g. action names).

Actions. Like tables, actions are also modeled as C functions. The action parameters should be translated taking into account the table modeling strategy. When the forwarding rules are unknown, the action parameter values are also unknown. In this case, the actions parameters are treated as symbolic variables. If the forwarding rules are supplied, then the values specified by the rules are assigned to the corresponding parameters.

Control Blocks. Since a control block in P4 also includes its action and table declarations, each block is translated to multiple C functions. Local scope variables in control blocks are declared as global variables in the model to allow them to be referenced by any table and action in the block. The block body usually contains invocations to tables and actions, which are modeled as their corresponding C function invocations.

Parser. Parsers are translated to multiple C functions: one for the parser declaration itself and another for each of its states. Since local parser parameters and variables can be accessed by any state in its scope, both structures are modeled as global variables in C. Parser output parameters, which represent the packet headers, are modeled as symbolic variables, as they correspond to inputs in the model.

Assertions. Each assertion element is modeled in C using a particular approach. Numeric and boolean expressions, as well as the *if* method, are directly translated to their equivalent statements

	P4 Program	C Model	Comments
Header	1 header ethernet_t { 2 bit<48> dstAddr; 3 bit<48> srcAddr; 4 bit<16> etherType; 5 } 6	typedef struct { uint8_t isValid : 1; uint64_t dstAddr : 48; uint64_t srcAddr : 48; uint32_t etherType : 16; } ethernet_t;	Header fields are mapped to struct members. The C struct also contains a header validity field.

Table	7 table forward_table() {
	8 actions = {
	9 forward1;
	10 NoAction;
	11 }
	12 key = {
	13 hdr.ethernet.dstAddr: exact;
	14 }
	15 size = 32;
	16 default_action = NoAction();
	17 }
		void forward_table() {
		int symbol;
		make_symbolic(symbol);
		switch(symbol) {
		case 0: forward(); break;
		default: NoAction(); break;
		}
		}
			Tables are modeled as C functions. In this example, the table rules are unknown. A symbolic variable is used to make the symbolic execution traverse both actions.

Action	18 action forward(bit<9> port) {
	19 standard_metadata.egress_spec = port;
	20 }
	21
	22
		void forward() {
		uint32_t port;
		make_symbolic(port);
		standard_metadata.egress_spec = port;
		}
			Actions are mapped to C functions. The action parameters are modeled with symbolic values when forwarding rules are unknown.

Control Block	23 control ingress(inout headers hdr,
	24 inout metadata meta) {
	25 apply {
	26 forward_table.apply();
	27 }
	28 }
	29
		// global variables
		headers hdr;
		metadata meta;
		
		void ingress() {
		forward_table();
		}
			Control blocks correspond to functions in the C model. Their parameters are mapped to global variables in the model.

Parser	30 parser TopParser(packet_in b,
	31 out Parsed_packet hdr) {
	32
	33 state start {
	34 transition parse_ethernet;
	35 }
	36
	37 state parse_ethernet {
	38 b.extract(hdr.ethernet);
	39 transition select(hdr.ethernet.etherType) {
	40 0x0800: parse_ipv4;
	41 default: accept;
	42 }
	43 }
	44 }
	45
	46
	47
		Parsed_packet hdr;
		
		void TopParser() {
		make_symbolic(hdr);
		start();
		}
		
		void start(){
		parse_ethernet();
		}
		
		void parse_ethernet() {
		hdr.ethernet.isValid = 1;
		switch(hdr.ethernet.etherType){
		case 0x0800: parse_ipv4(); break;
		default: accept(); break;
		}
		}
			A function is created in the model for each parser and parser state. The Parsed_packet parameter is made into a global variable in the C model.

Figure 5: Example of P4 to C translation for the main P4 structures.

in C. For the remaining methods, a separate process (which we omit due to the lack of space) is applied. Location-unrestricted methods, such as *forward*, are modeled as a single boolean variable that is initialized at the point where the assertion is declared, set at other relevant points, and usually tested at the end of the model, after it gets its final state when executed.

External objects. This type of structure is specific to each forwarding device, and P4 programs only interact with their interfaces. For this reason, the behavior of each external object should be previously known. In practice, this means integrating its corresponding model into the translator by using libraries, for example. In this work, we support the external objects necessary to translate the examples presented in Section 4 (e.g. counters and meters of the standard architecture).

3.4 Symbolically executing program models

After being generated by the process described in the previous section, the C model of a P4 program is verified by a symbolic engine. The symbolic execution of a program requires that all its feasible control flows (i.e., its execution paths) are evaluated through symbolic input variables.

Essentially, P4 programs describe how a data packet should be processed when entering a forwarding device, generating an output packet at the end or simply dropping the original packet. In this scenario, the incoming packet headers entering the device are treated as inputs to the model and thus are always assigned to symbolic values. The number of execution paths of a P4 program, in turn, is essentially given by its packet processing pipeline structure. Whenever a table can only be accessed under some condition (e.g., depending on the used protocol), a new execution path is created.

Table 1: Expressiveness of the proposed assertion language

Program	Properties / Assertions
MRI [17]	Switch IDs added to packets are authentic *constant(swid)* Added IDs are not removed *if(extract_header(swid), emit_header(swid))*
Timestamp switching [11]	Out of range timestamps are not forwarded to receivers *if(forward, rtp.ts < max_timestamp)*
sTag [21]	Hosts connected to ports of different colors cannot communicate *if(ingress_port == color_a &&* *ipv4.dstAddr == color_b_host, !forward)*
Dapper [12]	Only SYN packets register new flows *If(traverse_path*, tcp.ack == false)* *path that register new flows Load flow registers when is Ack packet *if(tcp.ack == 1, traverse_path*)* *path that load registers
DC.p4 [24]	L3 ACL is effective *if(ipv4.dstAddr == blocked_addr, !forward)* Cloned and original packet have different output ports ! *(cloned_outport == original_port &&* *constant(cloned_outport))*

The same happens whenever multiple actions can be invoked by the same table, generating a new branch for each possibility.

4 EVALUATION

We have prototyped ASSERT-P4 on top of the KLEE symbolic execution engine (version 1.3.0). To build C models, we first convert a P4 program to its JSON representation using the reference compiler provided by the *P4 Language Consortium*, and then translate the JSON representation (a DAG) to C code using a translator we developed specifically for this purpose. The translator contains approximately 750 lines of Python code. Shell scripts are used to automatically coordinate the invocation of each tool in the verification process. We make all the source code as well as the workloads employed in this evaluation publicly available.[2] The tool may be used by other researchers, who may want to reproduce our results. All experiments have been performed using a Linux virtual machine (kernel version 4.8.0) with a 3 GHz core and 16 GB of RAM.

4.1 Bug finding

First, we demonstrate the effectiveness of ASSERT-P4 in finding bugs and policy violations in programmable data planes. We uncovered several of them in recent P4 applications, a few of which we present here. All identified bugs were manually confirmed in their respective source code. With the exception of the DC.p4 example, it was not necessary to provide forwarding rules to expose these issues.

Dapper [12]: Dapper is a data plane performance diagnosis tool that infers TCP bottlenecks by analyzing packets in real time. It forwards traffic based on IPv4 addresses and uses SYN flags as well as sequence and ack numbers for calculating metrics such as loss rate and path latency. A failed assertion (packets with a time to live (TTL) value of zero are not forwarded) found that Dapper forwards packets without checking their TTL field, which can enable routing loops in the network.

NetPaxos [5]: NetPaxos is a network-based implementation of the Paxos consensus protocol. There are two different types of P4 programs in this application, one for Leaders/Coordinators and another for Acceptors. According to the protocol, Leaders determine a round number and ask acceptors for acknowledging it. Acceptors, in turn, decide whether they acknowledge or not a given request from a Leader. This process is repeated until a quorum of acceptors acknowledges the same round number, allowing the leader to establish a value for a given variable and consensus is achieved. ASSERT-P4 was able to find a bug in the current acceptor implementation. More specifically, each acceptor marks every packet that contains a round number to be dropped before it decides whether to acknowledge the packet or not. However, packets are not unmarked even if they are acknowledged, which means they will not be forwarded and consequently, the vote will not be counted by the Leader. Ultimately, this will prevent the protocol from achieving consensus. According to the authors feedback on this bug, the code was ported to $P4_{16}$, leaving the old code base unmaintained and exposed to bugs.

DC.p4 [24]: DC.p4 implements the behavior of a data center switch. It contains multiple functionalities such as L2/L3 forwarding, ECMP, VLAN, packet mirroring, tunneling and multiple ACLs (i.e., L2, L3 or based on more specific headers). Interestingly, ASSERT-P4 found that configuring only a layer-3 ACL (i.e., an ACL based on IP addresses) is not enough for dropping IPv4 packets regardless of the policy being enforced. In fact, we checked that the L3 ACL only flags packets to be filtered by another module in the system, which must also be appropriately configured. Although this is not effectively a bug, it can be a dangerous design decision since there is no documentation explaining how to properly configure the program.

Switch [19]: Since the introduction of the DC.p4 paper, its code base has evolved to the switch.p4 program, where it is actively maintained. We have used ASSERT-P4 to reproduce a known, reported bug on its repository: The modification of a field of an invalid header.[3] This is demonstrated by testing with an assertion if the header is valid before setting its fields.

4.2 Language expressiveness

To evaluate our assertion language, we tested its expressiveness in terms of the properties we can specify for different P4 programs. Table 1 shows a subset of the properties we tested for each P4 application. The associated assertions are italicized. We can specify a large set of properties, both program-dependent (e.g., the ones testing if registers are correctly manipulated in Dapper) and generic

[2] https://github.com/ufrgs-networks-group/assert-p4

[3] https://github.com/p4lang/switch/pull/102

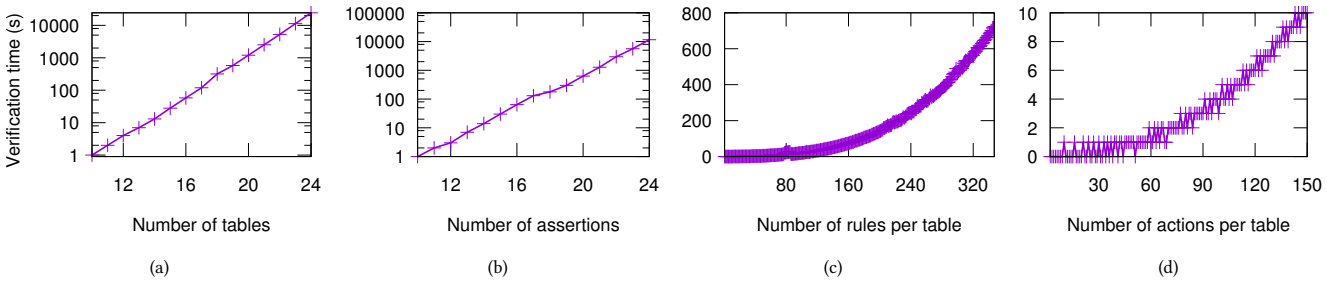

Figure 6: Performance analysis of the proposed tool.

ones (e.g., testing whether headers have been removed from packets or not). Furthermore, both security and correctness properties can be specified, like header integrity and well-formedness, respectively. As a research direction, we envision the automatic translation of high-level networking and security policies to our assertion language. This will allow network operators to easily verify complex network topologies.

4.3 Performance analysis

In this section, we assess how our verification approach scales according to different characteristics of P4 programs. To this end, we used the Whippersnapper [6] benchmark to generate data plane programs and verified instances varying the number of the following elements, while maintaining the other parameters constant: (i) tables in the packet processing pipeline; (ii) actions associated with each table; (iii) forwarding rules used to configure a program; and (iv) assertions used to express properties. Figure 6 shows the results. When held constant, we used no forwarding rules and assertions, 1 (Fig. 6(b)) and 2 tables (Figs. 6(c) and 6(d)), and 3 actions in the first table and 2 actions in every subsequent table.

Verification time grows exponentially with all the tested factors. However, it is less susceptible to the increase in the number of rules (Fig. 6(c)) and actions per table (Fig. 6(d)) compared to the number of tables (Fig. 6(a)) and assertions (Fig. 6(b)). This complexity is intrinsic to symbolic execution, and although we understand this is a limitation for our tool, we envision optimizations to this scalability problem as a research direction. Furthermore, ASSERT-P4 was efficient on most of the tested programs, while revealing relevant bugs. Pragmatically, most of the existing P4 programs fit into this same efficiently verifiable class at this time [5, 11, 12, 14, 20, 23].

5 RELATED WORK

Network verification. Many tools were proposed for verifying correctness and security properties in computer networks over the last few years. They are based on a myriad of techniques and address different properties and/or network architectures. Our intention while describing them is not to be exhaustive, but summarize the main techniques developed over the years. Flover [25] and NICE [4] use model checking to prove that the set of rules installed in OpenFlow switches satisfies a given networking policy. Symnet [26] verifies data plane models built with SEFL, a language designed to be symbolically executed. VMN [22] focuses on verifying reachability and isolation in networks containing stateful middleboxes. HSA [15] proposes header space algebra as a technique for checking reachability, isolation of network slices and packet leakage. VeriFlow [16] and DeltaNet [13] are customized solutions that use special representations of the data plane for allowing property verification in real time. None of them, however, are able to model P4 programs. NOD [21] uses Datalog to model both the network and its reachability properties. Recently, a solution that translates P4 programs to Datalog was proposed in the literature [21], but unlike ASSERT-P4 it cannot reason about program-specific properties. Finally, [9] is a concurrently ongoing work that shares the goals of ASSERT-P4 and some of its design details. Our paper, however, provides an in-depth evaluation of the effectiveness and scalability of ASSERT-P4 in finding bugs and verifying properties in real P4 applications.

Assertion language. Beckett *et al.* [1] present an assertion language to verify SDN applications. It enables expressing properties that the data plane should satisfy at different points of a control program. The assertions are verified using the VeriFlow [16] tool, which, like Flover, acts over forwarding rules instantiated in OpenFlow devices. While the language Becket *et al.* propose is used in SDN applications, our approach is to directly annotate a data plane program to prove properties of interest.

6 CONCLUSION AND FUTURE WORK

State-of-the-art network verification tools are unable to prove general security and correctness properties of P4 programs. Our solution, ASSERT-P4, provides an expressive assertion-based language for specifying properties, and a reasonably efficient tool for verifying them on data plane programs. Our evaluation shows that ASSERT-P4 can find a broad range of bugs on real P4 applications. In future work, we plan to investigate alternatives to improve its performance on programs containing complex packet processing pipelines (i.e., with dozens of match-action tables) as well as control configurations.

ACKNOWLEDGMENTS

We thank Nate Foster and the anonymous reviewers for the feedback. This work has been supported by grants NSF CNS-1740911 and RNP/CTIC (P4Sec), as well as FAPERGS (APE).

REFERENCES

[1] Ryan Beckett, Xuan Kelvin Zou, Shuyuan Zhang, Sharad Malik, Jennifer Rexford, and David Walker. 2014. An Assertion Language for Debugging SDN Applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN '14)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2620728.2620743

[2] 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 Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[4] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test Openflow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=2228298.2228312

[5] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.* 46, 2 (May 2016), 18–24. https://doi.org/10.1145/2935634.2935638

[6] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. 2017. Whippersnapper: A P4 Language Benchmark Suite. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 95–101. https://doi.org/10.1145/3050220.3050231

[7] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 101–114. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/dobrescu

[8] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 275–289. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/fayaz

[9] Nate Foster, Cole Schlesinger, Robert Soulé, and Han Wang. 2017. A Program Logic for Automated P4 Verification. In *P4 Workshop*.

[10] Lucas Freire, Miguel Neves, Alberto Schaeffer-Filho, and Marinho Barcellos. 2017. POSTER: Finding Vulnerabilities in P4 Programs with Assertion-based Verification. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2495–2497. https://doi.org/10.1145/3133956.3138837

[11] Tomas G. Edwards and Nick Ciarleglio. 2017. Timestamp-Aware RTP Video Switching Using Programmable Data Plan. Industrial Demo. In *ACM SIGCOMM*.

[12] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data Plane Performance Diagnosis of TCP. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 61–74. https://doi.org/10.1145/3050220.3050228

[13] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 735–749. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex

[14] Theo Jepsen, Leandro Pacheco de Sousa, Huynh Tu Dang, Fernando Pedone, and Robert Soulé. 2017. Gotthard: Network Support for Transaction Processing. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 185–186. https://doi.org/10.1145/3050220.3060603

[15] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=2228298.2228311

[16] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2012. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN '12)*. ACM, New York, NY, USA, 49–54. https://doi.org/10.1145/2342441.2342452

[17] The P4.org language consortium. 2017. MRI Exercise. https://github.com/p4lang/tutorials/blob/master/SIGCOMM_2017/exercises/mri/solution/mri.p4. (2017).

[18] The P4.org language consortium. 2017. P4 reference compiler. https://github.com/p4lang/p4c. (2017).

[19] The P4.org language consortium. 2018. Switch. https://github.com/p4lang/switch. (2018).

[20] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 481–495. https://doi.org/10.1145/2999572.2999609

[21] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

[22] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 699–718. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths

[23] Salvatore Signorello, Radu State, Jérôme FranÃğois, and Olivier Festor. 2016. NDN.p4: Programming information-centric data-planes. In *NetSoft*.

[24] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. DC.P4: Programming the Forwarding Plane of a Datacenter Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2774993.2775007

[25] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. Model checking invariant security properties in OpenFlow. In *2013 IEEE International Conference on Communications (ICC)*. IEEE, 1974–1979.

[26] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 314–327. https://doi.org/10.1145/2934872.2934881

[27] The P4.org language consortium. 2017. P4_16 Language Specification. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.html. (2017). Accessed: 2017-11-08.