Verification of P4 Programs in Feasible Time using Assertions

Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, Marinho Barcellos
Institute of Informatics
UFRGS

ABSTRACT

Recent trends in software-defined networking have extended network programmability to the data plane. Unfortunately, the chance of introducing bugs increases significantly. Verification can help prevent bugs by assuring that the program does not violate its requirements. Although research on the verification of P4 programs is very active, we still need tools to make easier for programmers to express properties and to rapidly verify complex invariants. In this paper, we leverage assertions and symbolic execution to propose a more general P4 verification approach. Developers annotate P4 programs with assertions expressing general network correctness properties; the result is transformed into C models and all possible paths symbolically executed. We implement a prototype, and use it to show the feasibility of the verification approach. Because symbolic execution does not scale well, we investigate a set of techniques to speed up the process for the specific case of P4 programs. We use the prototype implemented to show the gains provided by three speed up techniques (use of constraints, program slicing, parallelization), and experiment with different compiler optimization choices. We show our tool can uncover a broad range of bugs, and can do it in less than a minute considering various P4 applications.

CCS CONCEPTS

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

KEYWORDS

P4; Verification; Programmable Data Planes

ACM Reference Format:
Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, Marinho Barcellos. 2018. Verification of P4 Programs in Feasible Time using Assertions. In *The 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18), December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281421

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 which packet headers should be manipulated, and how, by different forwarding devices in the infrastructure. Despite the flexibility, this paradigm also increases the chance of introducing bugs into the network due to incorrect implementations.

Testing/debugging, verification and enforcement are complementary approaches that can help solve this problem. During development, data plane programs can be debugged and tested, providing a wide range of inputs and checking if the corresponding outputs match the expected behavior. Verification, on its turn, can be used on programs to find bugs that would violate any of the properties stated by their requirements, including bugs that are hard to reproduce in testing. Lastly, with enforcement, the data plane can be monitored during execution to trap and block actions that would result in property violations.

In this paper, we focus on *verification*: we propose an approach to model and check (at compile time) general security and correctness properties of P4 programs, and implement it in a tool that provides network verification in feasible time. Several approaches have been developed to check if a given fixed-function (non-P4) data plane satisfies a set of intended properties [8, 25, 29, 32]. Moreover, verifying P4-programmed data planes is an active area of research, with recent projects proposing verification techniques based on SMT solving [24, 27] and custom symbolic execution [33][1]. In contrast, this work shows how to efficiently verify P4 programs leveraging a popular, off-the-shelf symbolic execution engine [4].

We propose an expressive assertion language (highly influenced by P4) that enables programmers to specify their intended properties by annotating their P4 code. Once annotated, a program is symbolically executed, with assertions being checked while all its paths are traversed. Given that the time taken to perform the symbolic execution grows exponentially with the program complexity, we show how a variety of speed up techniques can be employed to reduce the verification time and number of executed instructions. These techniques consist of using annotations in code to constrain the paths to be traversed according to properties and/or protocols of interest, program slicing to reduce the complexity of the model under verification, and parallelization of symbolic execution. Besides, we experiment with code optimization features offered by current compilers.

To evaluate our approach, we built a prototype using KLEE [4] and the P4 Reference Compiler [20] for the current language version, $P4_{16}$. We applied it to four real P4 applications collected from the literature: Switch [21], NetPaxos [5], Dapper [11], and DC.p4 [31]. Our results show that the proposed verification process can uncover a broad range of bugs either in the data plane program itself or in its control plane configuration. A detailed performance analysis also shows that, although the verification time grows exponentially with

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12…$15.00
https://doi.org/10.1145/3281411.3281421

[1] [24, 33] were independently developed at the same time as this work.

the number of tables and assertions, pragmatically our approach needs less than a minute to verify various P4 applications [6, 10, 11, 13, 23, 30].

This paper presents the following contributions:

(1) a language for specifying general correctness and security properties of P4 programs;
(2) an approach for verifying properties of P4 programs using assertion checking and symbolic execution, and its implementation in a tool;
(3) investigation of a set of techniques to speed up the execution of the verification tool;
(4) examples of the tool being applied to find bugs in P4 applications proposed in the literature;
(5) a detailed performance evaluation of the tool implementing our verification approach.

2 MOTIVATING EXAMPLES

Bugs in P4 programs can originate from a myriad of sources (e.g. erroneous assignments, poor logic or control misconfiguration) and have different consequences depending on the program. Next, we present two motivating examples to illustrate how bugs and their effects can be specific to each P4 implementation. To allow illustration in limited space, we use simplistic bugs from likely copy-paste mistakes.

Code circumvention. Figure 1 shows an example of a vulnerability in a P4 program. This code snippet specifies a packet processing pipeline containing three match-action tables (udp_table, tcp_table and tcp_acl_table), invoked inside a L4 control block. Clearly, udp_acl_table should be applied to UDP traffic, but accidentally tcp_acl_table was used instead, allowing UDP packets to bypass the filtering mechanism. Even though correcting this problem would be simple, finding the problem may not be trivial in large and complex programs.

```
1 control L4() {
2   apply {
3     if (headers.ip.nextHeader == TCP) {
4       tcp_table.apply();
5       tcp_acl_table.apply();
6     } else if (headers.ip.nextHeader == UDP) {
7       udp_table.apply();
8       tcp_acl_table.apply();
9     }
10  }
11 }
```

Figure 1: Code circumvention bug allows UDP to pass through

Control misconfiguration. Many faults in networks arise from bugs in forwarding rules (i.e., control plane configurations). In this sense, Figure 2 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.

As hinted earlier, these bugs could be identified and prevented during development by verifying correctness properties of interest.

```
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 }
```

Figure 2: Control misconfiguration bug because rule in line 8 clones packets to the same egress port

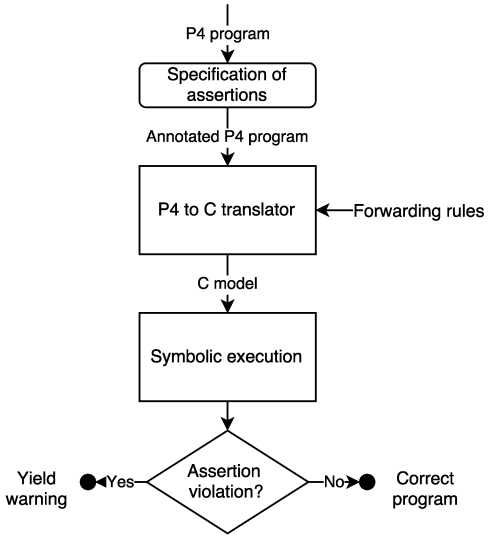

Figure 3: Overview of the verification process

3 P4 PROGRAM VERIFICATION

We first present an overview of our approach, based on Figure 3. There are two key ideas behind it: (*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 automatically verified. Using models to represent real programs is a common practice in the verification literature [9, 25, 29, 34].

The P4 developer first annotates the 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 then checked by a symbolic execution engine, which tests all 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 violations are reported, allowing the developer to correct the program. In the following subsections, we describe in detail each step of the verification process.

3.1 Specifying assertions

Developers use assertions to express properties of P4 programs. An *assertion language* is needed to capture packet processing behaviors and facilitate the task of specifying complex networking properties. This includes reasoning about packet formation, forwarding, and control flow properties, whose behavior may depend not only on the state of the program variables at a specific location, but also on how the program manipulates the packets at other points of the code. To accomplish this goal, we introduce an assertion language using the code annotation mechanism available in P4. We define an annotation called assert, enabling the developer and/or a third party to express/interpret properties in an intuitive manner.

Figure 4 summarizes the grammar of the assertion language. It resembles C-style assertions found in traditional programming languages, lowering the barrier to adoption. However, our concept of assertion is more general and includes both *location-restricted* and *location-unrestricted* elements. A location-restricted element is one that tests the value of a program variable where the assertion is specified, as in traditional programming languages like C or Java. The location-unrestricted ones, in contrast, apply to the entire data plane program. They can be used for example to guarantee higher level properties that the program is expected to satisfy, such as isolation – asserting certain packets would never be forwarded to certain ports, or to guarantee that some actions will be taken on certain headers. The idea was inspired by Beckett et al [1].

```
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 4: Assertion language grammar

Syntactically, each assertion is composed of a boolean expression, which may include primitive methods. Allowed methods are forward, traverse_path, constant, if, extract_header, and emit_header. Both expressions and methods can operate over one or more values, header fields or headers. There is no syntax difference between location-restricted and location-unrestricted elements. 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. Expressions can be integer or boolean, and in both cases with the same semantics as their counterparts in the P4 language.

The methods work as follows. if(b_1, b_2, [b_3]) is similar to traditional conditional statements: if expression b_1 is true, then expression b_2 will be evaluated, otherwise the alternative b_3 will be evaluated). This is the only location-restricted method, with all other ones being unrestricted. traverse_path() indicates if a given

```
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 5: Example of an annotated P4 program

structure in the program (e.g., an action) will be eventually traversed before program execution ends. constant(f) is true if field f will not change from the assertion location onwards, i.e., until the program terminates. forward() returns true when the packet will not be dropped at the end of the program. extract_header(h) is true if a header h has been, or will be, extracted from the packet. Finally, emit_header(h) returns true if packet will be transmitted with header h.

The methods presented in the language enable the specification of types of properties that would be either difficult or impossible to express using only traditional assertions. The addition of forward() enables the expression of forwarding properties, which are essential to data plane programs. traverse_path() allows reasoning about the control flow of the source code. constant() facilitates checking the integrity of variables across the program. Both extract_header() and emit_header() allow the expression of packet formation properties at the parser and deparser level, respectively. Finally, if() assists the process of combining methods and expressions in a conditional expression.

Figure 5 shows an example of an annotated P4 program, with assertions 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 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., forward() captures the state of the program at the end of its execution) and 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 found).

	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 6: Example of P4 to C translation for the main P4 structures

3.2 Constructing C models

Once a P4 program is annotated, it is translated to an equivalent model expressed in C language. This section describes how the main P4 structures, namely headers, tables, actions, parsers, control blocks, and external objects, are translated into the C model. Figure 6 (in the next page) summarizes the translations by means of examples. Implementation details appear in ♯3.4.

Headers. Given their similar representations, P4 headers can be easily modeled by structs in C. Each header field is mapped to a struct member, and *bit fields* in C are used to keep the matching 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, considering its declared size. Fields with more than 64 bits can be modeled using bit arrays.

Tables. Each forwarding table in a P4 program is modeled as a function in C. Functions created from tables are constructed in different ways depending on whether the forwarding rules are supplied to the translator or not. If the rules are provided, the match fields in the P4 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 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. If so, the action parameters are treated as symbolic variables. Otherwise, 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. Given that the global variables are uniquely named in the model, and that they are not reused across different packets, this modeling approach does not cause side effects on the verification result. Finally, 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 in C. To model location-unrestricted methods, we use boolean values that are set at different places depending on the method, and tested when getting to its final state.

To model methods extract_header(), emit_header(), and traverse_path(), a global boolean value is created for each one of their occurrences in the P4 program. Such variables assume an initial false value, and are assigned to true at different model locations depending on its corresponding method. In occurrences of extract_header(x), the assignment is made just after an extract() method invocation, which receives the header x as a parameter in the P4 program. Similarly, the assignment corresponding to emit_header(x) is made immediately after an emit() invocation (associated to the packet_out basic type) containing header x as a parameter. For traverse_path(), the assignment occurs just before the assertion that declares it. Method forward() is modeled with a single boolean value initially set to true. Its value is assigned to false inside the drop action and reject parse state. constant(f) is translated by storing the field f in a C variable right after (or before) an assertion, and testing if the variable value remains the same at the end of the symbolic path.

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. This limitation is inherent to the design of P4, which consists of both architecture-dependent and architecture-independent code. In this work, we support the external objects necessary to translate the examples presented in ♯5 (e.g. counters and meters of the standard architecture).

3.3 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 possible control flows (i.e., its execution paths) are evaluated through symbolic input variables. To this end, the implementation described in this paper uses the KLEE symbolic engine [4].

Essentially, P4 programs describe how a data packet should be processed when entering a forwarding device, potentially leading to the emission of an output 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. The same happens whenever multiple actions can be invoked by the same table, generating a new branch for each possibility. This leads to the "path explosion problem", as the number of paths increases exponentially with program size. In ♯4, we investigate approaches to speed up symbolic execution in the specific context of our proposal, P4 programs.

3.4 Prototype implementation

We have prototyped our approach in a tool to show its feasibility and to investigate the gains in performance given by different speed up strategies. The tool uses a set of Python and shell scripts, and is based on the KLEE symbolic execution engine and the LLVM Compiler Infrastructure [22].

As shown in Figure 3, the tool first converts the annotated P4 program to its JSON representation, and it does so using the reference compiler for $P4_{16}$ provided by the P4 Language Consortium. It then translates the JSON representation (a DAG) to a corresponding model in C language, as detailed in ♯3.2. The implementation of the translation process is straightforward and takes approximately 950 lines of Python code. The C model is symbolically executed by KLEE, which first uses the LLVM compiler (version 3.4) to translate the C program to a corresponding LLVM assembly language representation.

We make the source code and data sets used in this evaluation publicly available[2]. As such, the tool may be used by other researchers to reproduce our results (see ♯5).

4 SPEEDING UP P4 VERIFICATION

Because our approach is based on symbolic execution, we inherit both its benefits and its limitations, including the path explosion problem. While many P4 programs currently found in the literature are fairly small [6, 10, 11, 13, 23, 30], real-world P4 programs are expected to become larger and more intricate, specially when having to deal with several protocols (e.g. programs DC.p4 and Switch.p4).

To address this limitation, we investigate optimization techniques to speed up our verification tool. The number of paths to be traversed in a single execution can be reduced by means of packet and control flow constraints and program slicing, and implicitly by means of compiler optimizations. The paths that still need to be traversed can be examined concurrently, with advantages when executing on top of a parallel architecture.

[2] https://github.com/gnmartins/assert-p4

4.1 Packet and control flow constraints

During the verification process, a developer may be interested in the verification of properties pertaining only to certain types of packets or control flows. An example is a P4 program that supports both TCP and UDP, but a developer needs to check properties related to UDP only. To avoid symbolically executing non-relevant paths (in the example, TCP ones), we propose to add packet and control flow constraints to the verification process, to "direct" the symbolic execution to the paths of interest while ignoring the others.

We implemented this strategy by allowing P4 programs to be annotated with assumptions. We define the annotation assume(), which receives as argument a boolean value expected to be true. This annotation can be directly translated to a klee_assume() method in the C model. The method belongs to the KLEE API and is responsible for implementing the assumption within the KLEE symbolic engine.

To illustrate this approach, consider a P4 program that implements multiple L3 protocols, but the properties of interest concern only the IPv4 protocol. This example is shown in Figure 7. The annotated P4 code is shown in Figure 7(a). It consists of the parser state responsible for reading the Ethernet bits of the packet, which indicate the L3 protocol in the packet. By assuming that (the type of) the next protocol is IPv4, the symbolic execution will ignore the other paths, and either traverse the parse_ipv4 parser state or halt if the assumption is impossible to hold. Figure 7(b) shows how the assumption and transition statements are translated to the C model.

```
1 state parse_ethernet {
2    b.extract(hdr.ethernet);
3    @assume(hdr.ethernet.etherType == IPV4);
4    transition select(hdr.ethernet.etherType) {
5        IPV4 : parse_ipv4;
6        IPV6 : parse_ipv6;
7        ICMP : parse_icmp;
8        ...
9    }
10 }
```

(a) P4 code annotated with a constraint

```
1 klee_assume(hdr.ethernet.etherType == IPV4);
2 parse_ipv4();
```

(b) Constraint translated to the C model

Figure 7: Constraint example

4.2 Program slicing

To verify an assertion, the verification process symbolically executes the whole P4 program. However, the assertion result may depend only on a subset of the program instructions. Thus, the verification procedure spends unnecessary time processing paths that do not affect the outcome. This creates the opportunity to use a *program slicing technique* [36], which is used to automatically remove the subset of a program that does not affect a *selected criteria*, such as the value of a variable in a given point of the code.

To illustrate, assume a developer annotates a P4 program with a set of assertions that depend *only* on the TCP destination port (the selection criteria). The slicing algorithm in this case can automatically generate a "program slice" that contains only parsers, actions, tables, and control flow instructions that can directly or indirectly modify the value of the TCP destination port. This simplifies the program, removing the parts of the code related to other protocols, and even the sections associated with TCP that do not modify the destination port.

Slicing could be performed on the C model or P4 program. We decide for the former, as it allows the use of existing program slicing tools. To this end, we incorporate the Frama-C [17] slicing plug-in into the verification tool, applying Frama-C before symbolically executing the C model.

4.3 Compiler optimizations

Our approach relies on KLEE symbolically executing a LLVM assembly language representation of the C code, which itself is a representation of the P4 data plane program. Both KLEE and the LLVM compiler support optimization passes. These can be control flow graph simplifications, which can remove dead code and merge blocks of instructions, as well as global variable optimization, which marks unchanged variables as constant and removes unused variables. These passes can alter the source code in order to make it more efficient, potentially reducing the symbolic execution time.

Given the expected performance gains from compiler optimization passes, we experiment with these parameters (using the compiler as a black box) to measure how optimizations affect verification time of the generated model.

4.4 Parallelization

Even though the branches of symbolic execution could be traversed concurrently, KLEE does not take advantage of parallelization. Cloud9 [3] allows KLEE to use multiple processing elements while symbolically executing any C code[3]. We follow a different approach, and propose a simple parallelization strategy that is specific to P4 programs.

The strategy consists of dividing the model into *submodels*, which are statically generated from decision points (e.g. if and switch). One submodel is created for each branch, and run via a concurrent KLEE process. Figure 8 exemplifies this process with the corresponding code fragments, with the original model and its two submodels shown respectively in Figures 8(a), 8(b) and 8(c). In each submodel a value is assumed for the condition and only its corresponding instructions are executed. The submodels generated are completely independent and thus can be executed concurrently, in any order. If multiple processing elements are available in the underlying hardware, the procedure can be repeated on submodels to increase the degree of concurrency.

To maximize the performance gains of parallelization, we should minimize the maximum height of the execution trees of all submodels. This is so because the longest path is the one that will take the most time to traverse. Since the symbolic execution tree is

[3]the tool was discontinued in 2013.

```
1 if(hdr.ethernet.etherType == IPV4){
2    parse_ipv4();
3 } else {
4    accept();
5 }
```

(a) Original model

```
1 klee_assume(hdr.ethernet.etherType == IPV4);
2 parse_ipv4();
```

(b) Submodel with IPv4 packet

```
1 klee_assume(hdr.ethernet.etherType != IPV4);
2 accept();
```

(c) Submodel with non-IPv4 packet

Figure 8: Example of submodel generation

unknown beforehand, it cannot be used to find the optimal solution. Therefore, we propose a heuristic based on the anatomy of P4 programs to partition the model.

Decision points that occur earlier in the program have more chances of being part of feasible paths, as they require less conditions to be traversed. In the case of a P4 program, initial decision points are typically found at the parser. Our heuristic then starts by creating the submodels from these first conditions seen by the parser. Once the first set of submodels is generated by this strategy, further divisions may not be as efficient because they have increasing chances of generating submodels on branches of unreachable paths. Alternatively, decision points associated with tables are appropriate candidates to submodel creation, since packets that traverse the longest paths usually pass through P4 tables after being accepted by the parser. In this case, each action in a table is traversed using a different submodel. The heuristic creates submodels, before any processing starts, by applying both approaches (parser and table branches). Once all the submodels have been generated, they are dynamically assigned (in arbitrary order) to any idle processing elements.

5 EVALUATION

Earlier on we described the approach and how it was implemented in a prototype, followed by strategies to speed up the verification. We used the prototype to perform experiments and show that our approach: (*i*) can detect a broad spectrum of bugs and policy violations in programmable data planes; (*ii*) allows the specification (and proof) of general correctness and security properties of P4 programs; (*iii*) can be optimized with the techniques presented; (*iv*) is efficient even for relatively complex P4 programs and control configurations.

All experiments have been performed using a Linux virtual machine (kernel version 4.8.0) with four 3 GHz cores and 16 GB of RAM.

5.1 Bug finding

First, we demonstrate the effectiveness of our proposal in finding bugs and policy violations in programmable data planes. We use assertions to find bugs in four recent P4 applications, and confirm manually examining the source codes.

Dapper [11]: Dapper is a data plane performance diagnosis tool that infers TCP bottlenecks by analyzing packets in real time. We placed a set of basic assertions at the beginning of the ingress control block, and via assertion if(ipv4.ttl == 0, !forward()), found that Dapper can forward IPv4 packets even when the field TTL is zero. By manual inspection, we noticed that even though the TTL field is decremented as expected, its value is never checked before forwarding. Because of this bug, while debugging a network with a loop, Dapper-based devices could keep forwarding packets forever. The prototype encountered this bug in less than a second.

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. All the other actors are assumed to be entirely implemented in end hosts. We examined the current version of the implementation and the set of forwarding rules made available by the authors, adding assertions to their code. As part of the implementation, an Acceptor votes by adding voting information to the incoming packets before forwarding them. Specifically, the execution violated the assertion if(traverse_path(), forward()), located inside the action that performs the vote. This indicates that there are valid packets (containing voting information) being dropped. By manually inspecting the code, we found that the problem occurs because the packets are first marked to be dropped by another action, and not unmarked by the voting actions. This bug can be corrected by marking the packets to be forwarded inside the actions which perform the vote. According to the feedback from authors on this bug, the code was ported to $P4_{16}$, leaving the old code base unmaintained and exposed to bugs. Our prototype found this assertion violation in less than a second.

DC.p4 [31]: 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). This program contains more than 2500 lines of code distributed among 37 tables. Tables, in turn, are organized assuming two sequential packet processing pipelines, one for incoming/ingress packets and another for outgoing/egress packets, interleaved by a queue system.

We verify if configuring the L3 ACL table to drop traffic with a specific destination IP address properly filters this type of packet. We used the assertion if(ipv4.dstAddr == FILTER_ADDR, !forward()) to express that packets with IPv4 destination addresses equal to FILTER_ADDR should be dropped. We found that just configuring the L3 ACL 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 an actual bug, it is still a misconfiguration in the program.

Switch [21]: Since the introduction of the DC.p4 paper, its code base has evolved to the Switch.p4 program, which is actively maintained. We have used our verification approach to reproduce two

known bugs, reported on its repository. The first one is the modification of a field of an invalid header.[4] This bug is replicated by testing with an assertion if the header is valid before setting its fields. The second bug is related to tunnel encapsulation[5], where encapsulated headers are overwritten whenever multiple nested levels are present. We included an assertion to test if the inner headers are not valid before performing the encapsulation. The assertion failed, confirming that encapsulated headers can be overwritten and their original contents, discarded.

5.2 Language expressiveness

To evaluate our assertion language, we assessed its expressiveness in terms of the properties we can specify for different P4 programs: VSS, MRI, TS switching, sTag, Dapper, NetPaxos and DC.p4. Table 1 shows a subset of the properties we tested for each P4 application.

The table demonstrates the use of a wide set of properties, both program-dependent (e.g., the ones testing if registers are correctly manipulated in Dapper) and generic ones (e.g., testing whether headers have been removed from packets or not). Furthermore, both security and correctness properties can be specified, such as header integrity and well-formedness, respectively.

5.3 Performance analysis

We assessed how our verification approach scales according to different characteristics of P4 programs. We performed experiments with and without the speed up strategies described in ♯4. This subsection shows the performance values obtained originally, without the optimizations.

We used the Whippersnapper [7] benchmark to generate data plane programs, and measured the impact of multiple parameters in verification times: (*i*) tables in the packet processing pipeline; (*ii*) actions associated with each table; (*iii*) forwarding rules used to configure a program; and (*iv*) number of assertions used to express properties.

Figure 9 shows the results, considering average verification time in seconds for various number of tables, assertions, rules per table and actions per table. Note that the first two plots have y presented in log scale. We adopted the following default values for parameters: no forwarding rules and assertions, 1 table in Fig. 9(b), 2 tables in Figs. 9(c) and 9(d), and 3 actions in the first table and 2 actions in every subsequent table.

The results show that verification time grows exponentially with all the factors, with the exception of the number of assertions, which grows linearly after an initial exponential growth. We can observe that verification time increases rapidly with the number of tables (Fig. 9(a)), actions per table (Fig. 9(d)), and rules per table (Fig. 9(c)). The number of assertions presented both the quickest and slowest growth in execution time, with a change in trend after 14 assertions (Fig. 9(b)).

Our approach was able to verify within a few seconds most of the programs in ♯5.1 and ♯5.2. However, the plots show clearly that our non-optimized version does not scale well, and that the verification of larger programs, with more tables and assertions, is likely unaffordable. This prompted us to investigate the adoption of optimization strategies to the context of P4 program verification.

Table 1: Examples of assertion language being used to specify different properties in several P4 applications

Program	Properties / Assertions

VSS [18]	Packets with zero TTL values are dropped
	if(ipv4.ttl == 0, !forward())
	Marked to drop packets are not forwarded
	if(traverse_path(), !forward())

MRI [19]	Switch IDs added to packets are authentic constant(id) Added IDs are not removed if(extract_header(id), emit_header(id))
Timestamp switching [10]	Out of range timestamps are not forwarded to receivers if(forward(), rtp.ts < max_timestamp)
sTag [25]	Hosts connected to ports of different colors cannot communicate if(ingress_port == color_a && ipv4.dstAddr == color_b_host, !forward())

Dapper [11]	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

NetPaxos [6]	Acceptor correctly votes according to paxos phase
	if(traverse_path()*, paxos.msgtype == 1A)
	*at the handle_1a action
	Leader increases round number at each instance
	if(traverse_path()*, paxos.msgtype == 2A)
	*at the increase_instance action

DC.p4 [31]	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))

5.4 Benchmarking optimization strategies

The benchmarks shown in ♯ 5.3 can be re-executed in combination with some of the optimization techniques proposed in Section 4. The parallelization and compiler optimizations are general techniques that are suited to be analyzed with synthetic programs generated by the Whippersnapper benchmark, whereas the results of applying the other optimization techniques are tightly coupled to a particular program and the properties of interest. Therefore, we compare the original benchmark results with their execution alongside the parallelization and compiler optimization techniques in Figure 10, where *O3* and *Opt* represent the LLVM and KLEE optimization flags respectively.

[4] https://github.com/p4lang/switch/pull/102
[5] https://github.com/p4lang/switch/issues/97

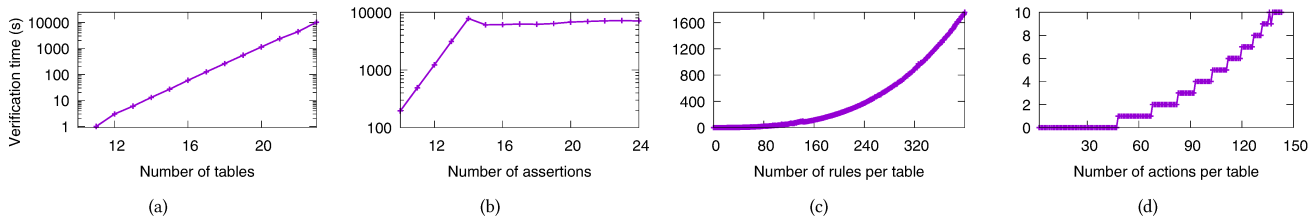

Figure 9: Performance analysis of the proposed tool

We can observe that, using the four available cores of the machine, the parallelization technique was effective in reducing the verification time when the number of tables of the program varied (Figure 10(a)), while the parallelization overhead increased the verification time in the other cases. Since the number of submodels created by the parallelization strategy grew in proportion to the number of rules per table (Figure 10(c)) and actions per table (Figure 10(d)), the number of concurrent executions quickly exceeded the number of processing elements of the machine, generating a proportionally larger overhead as the value of the x-axis increases.

Parallelization was also ineffective when we varied the number of assertions (Figure 10(b)). This can be explained by the use of a synthetic workload. Recall the submodel generation strategy is focused on parser and table branches; the submodels generated in this synthetic case were heavily unbalanced. Verification time was high whenever the submodel included the action containing the assertions, but requiring a negligible verification time otherwise.

The O3 LLVM optimization flag resulted in moderate performance gains in Figure 10(d) and the first trend of Figure 10(b). No gains were obtained in Figure 10(a) and the performance deteriorated in the second trend of Figure 10(b). When the number of rules per table varied (Figure 10(c)), a significant reduction in verification time from an exponential to a linear growth is observed. This can be explained by compiler optimizations applied to the cascading if-else statements used to decide which action should be executed based on the matched values of the forwarding rules.

The "optimize" flag provided by KLEE (named Opt in the graphs) resulted in small performance gains in Figure 10(b) and no gains in Figures 10(a) and 10(c). The optimize flag changed the behavior of the program under verification, as shown in Figure 10(d): it reduced the program to a constant number of instructions regardless of the number of actions used.

We can conclude that the efficiency of these optimization techniques depends on the characteristics of the program under verification. Hence, in the next section we analyze the impacts of using all the proposed techniques with various P4 programs found in the literature with the goal of obtaining insights on their application on existing programs.

5.5 Analysis of optimization strategies on existing P4 programs

We now present the results obtained from measuring the impact of each optimization technique applied to different existing P4 programs. We study their behavior by using the techniques in isolation, as well as by combining them in a single execution. To this end, we employ two metrics: (i) the verification time it takes to explore all the paths of the model, and (ii) the total number of instructions executed by the symbolic engine.

Table 2 presents the performance gains of each technique in comparison to using no optimizations, where the *O3* and *Opt* columns respectively represent the compiler optimization flag, and the KLEE optimization flag. Most of the verifications last less than a second even if no optimizations are applied. The exception is Dapper which takes roughly a minute to be verified without the speed up techniques. We do not show results for the other programs evaluated in this paper (e.g., switch.p4) because our analysis is still not able to scale to an all-paths exploration in those cases (within a two-hour timeout). Our ongoing work tries to cope with this by investigating heuristics that simplify dense conditional structures in P4 programs (e.g., by manipulating forwarding rules) as well as breaking the symbolic execution into separate program blocks (e.g., parser, ingress and egress pipelines), similarly to what was done in [8].

Packet and control flow constraints. Although this technique has the additional cost of requiring the developer to annotate the code with assumptions, the results presented in Table 2 reveal that it can greatly reduce the number of instructions symbolically executed. This reduction of instructions also leads to an equivalent reduction of the verification time of complex programs in which the symbolic execution is the verification bottleneck, as can be observed in the Dapper example. Furthermore, even though the Switch.p4 program was not included in the analysis presented in Table 2 due to its complexity, by using packet constraints in its bug finding examples (see Section 5.1), we were able to reduce the time taken to reveal the bugs from the order of days to the order of seconds. This was achieved by annotating the code with assumptions that led to the tested assertions.

Program slicing. Program slicing is capable of reducing the number of instructions considerably, from about 10 percent to more than 99 percent. However, the cost of performing the slicing with the Frama-C framework far outweighs the reduction of program size on smaller programs, which can be verified without the additional overhead in milliseconds. Furthermore, the Frama-C approach to slicing has no support for programs with recursion. This resulted in a failed attempt to slice the MRI program, which contains a recursion in its parser section of the code. Therefore, we conclude that while program slicing can be an effective technique in some cases, the development of efficient slicing approaches capable of dealing with parser recursions is necessary to enable the full adoption of this technique.

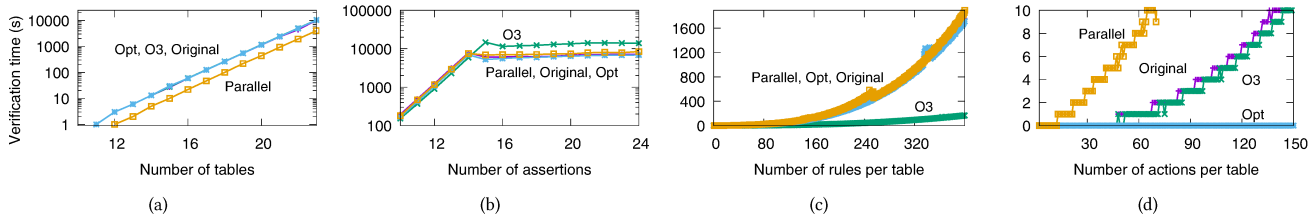

Figure 10: Effect of speed up techniques applied to synthetic benchmarks

Table 2: Performance gains of each technique

	Reduction in Verification Time					Reduction in Number of Instructions				
Program	**O3**	**Opt**	**Constraints**	**Parallel**	**Slice**	**O3**	**Opt**	**Constraints**	**Parallel**	**Slice**
Dapper	47.80%	15.29%	42.68%	33.03%	27.35%	54.86%	49.90%	50.04%	69.41%	30.70%
sTag	-13.54	3.13%	-4.17%	-39.58%	-421.88	56.79%	53.37%	58.14%	66.56%	10.59%
NetPaxos	-32.37	5.04%	28.78%	-47.48%	-273.38	39.43%	42.06%	81.90%	-0.36%	74.37%
TS Switching	8.51%	0.00%	2.13%	-40.43%	-423.40%	73.53%	54.55%	63.38%	38.71%	71.25%
VSS	-2.24%	1.60%	2.88%	30.13%	-71.79%	20.61%	33.35%	28.60%	38.07%	99.69%
MRI	-10.12%	0.39%	6.23%	-4.67	-	73.56%	74.36%	37.86%	15.84%	-

Compiler optimizations. Even though optimization passes were able to reduce the number of instructions from approximately 20 percent to up to almost 75 percent, our results indicate that they can have both positive and negative effects on the time taken to symbolically execute P4 program models. This can be explained by the short verification time of most of the tested programs, which makes the overhead of applying these passes larger than the gains obtained from reducing the number of instructions. The potential to reduce the total number of instructions even further in this domain can motivate the research of optimization passes specific for the symbolic execution of P4 programs.

Parallelization. The results show that the proposed parallelization approach can reduce verification time of some programs. The total verification time was reduced in approximately 33% with Dapper and 30% with VSS using only 4 cores. For very small programs that can be verified in milliseconds, the added overhead due to parallelization generally does not justify the use of this technique, increasing the total verification time in most cases, as can be observed with sTag, NetPaxos, Timestamp Switching, and MRI.

When generating submodels using the approach described in ♯ 4.4, each submodel ends up with a fraction of the total number of instructions of the original model. To achieve satisfying performance gains, the parallelization technique should try to minimize the difference between the number of instructions of the submodels. The tenth column of Table 2 presents the reduction on the number of instructions achieved by the submodel with the greatest number of instructions (i.e., worst case) when compared to the the original model. In this case, we can observe that this approach can reduce the number of instructions in most cases.

Combining the techniques. Since the use of the optimization techniques are not mutually exclusive, we analyze the potential of combining them to achieve optimal verification time. We observed that there is no optimization technique that can guarantee a reduction in verification time in every case. The combination of techniques that are effective in isolation may not yield optimal results when combined. This is especially observed with program slicing. The cost of executing Frama-C takes an increasingly larger part of the total verification time as the other techniques are combined, making its overhead exceed the gains obtained from a reduction in program complexity. The Dapper example, which has higher verification time than the other programs analyzed, benefited the most from the combination of techniques. By using constraints, parallelization, and compiler optimization flags, its verification time was decreased by 81.76% and its number of instructions by 89.25%.

6 DISCUSSION

Interaction with the control plane. P4 programs can describe the capabilities of a packet processing pipeline, but not the specific rules that dictate forwarding behavior. In general, these rules are defined by the control plane during network operation, and a P4 program just reflects the maximal set of possible behaviors the control plane can express. As a result, P4 verification tools need to deal with this possibly missing information (i.e., the lack of a control plane configuration) somehow. Our tool is quite flexible in that sense, and allows programmers to take the control plane into account in three different ways: i) input an specific set of rules, reflecting a given control plane configuration; ii) leave the rules unspecified, which will cause all possible behaviors to be checked; and iii) use assume statements to capture restrictions on the control plane behavior.

Specifying a set of forwarding rules has the advantage of accelerating symbolic execution [33]. However, it requires verifying the program again every time the rule set changes. Leaving all rules unspecified, on the other hand, results in a complete analysis of the P4 program, but allows the occurrence of false positives when violations are reported to cases that may never happen in practice

(e.g., because of restrictions at the controller code). Finally, using constraints to model the behavior of the control plane yields a more precise solution, but potentially requires a non-negligible programming effort for capturing and correctly specifying the control plane semantics. Possible alternatives to the burdens identified in the solutions above include automatically capturing the semantics of the control plane, checking for false positives or using customized data structures to store verification state (e.g., in line with tools like VeriFlow [16] or DeltaNet [12]). We leave these investigations as future work.

Validation of C models. To increase confidence in the accuracy of our P4 to C translator, we validate its generated models using input-output tests. More specifically, we select a set of packets P and use it as input to both the BMv2[6] switch (configured with the P4 program under test) and the associated model. We then compare both outputs to check if there is any discrepancy. Our ongoing work aims to automate this process, where we use a packet generator (e.g., p4pktgen [28]) to systematically generate test cases.

Stateful verification. P4 programs may contain persistent state (i.e., state that depends on a sequence of packets) in the form of registers, meters, counters and other types of extern objects. While reasoning about stateful networks or network functions is undecidable in the general case [35], the fact that P4 programs have bounded state (i.e., the amount and nature of the information that can be stored is known *a priori*) makes the problem tractable. To verify programs that contain registers, for example, we first model them using equivalent data structures in C, and then leave the verification proceed in one of two different ways: i) assume that registers can take any value, which is equivalent to making them symbolic; or ii) restrict their domain to a particular set. This approach is similar to [8].

Unsupported features. Although our prototype supports many of the P4 constructs, others still remain to be implemented. Examples of features our tool cannot handle at the time of writing include variable length fields, ternary matching keys (when forwarding rules are used), parsing exceptions, and parser value sets. Checksums are a special case because they cannot be calculated when headers are symbolic, situation in which we assume they are correct by default. We intend to pursue full compliance with the P4 specification in the future.

7 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. While the types of properties vary across the literature, they are mainly related to host reachability, including isolation, absence of black holes, and loop-freedom. Some focus on the control plane, while others, on the data plane.

Efforts that focus on the data plane are more similar to our approach. They operate by verifying if a particular snapshot of the data plane satisfies the network-wide properties. This strategy can be traced back to Anteater [26], which models the data plane as boolean functions that are analyzed with a SAT solver to check for reachability, network loops, black holes, and consistency. Similarly, Header Space Analysis (HSA) [15] proposes header space algebra as a technique for checking reachability, isolation of network slices and packet leakage. Based on HSA, NetPlumber [14] incrementally updates the network model as changes occur in the data plane. This allows efficient verification in real time. Other tools that perform real time verification of the data plane are VeriFlow [16], DeltaNet [12], and Flover [32]. VMN [29] focuses on verifying reachability and isolation in networks containing stateful middleboxes. NOD [25] uses Datalog to model both the network and its reachability properties. A solution that translates P4 programs to Datalog and verifies reachability and well-formedness was proposed in [27]. Further, p4v [24] converts P4 programs into Guarded Command Language (GCL) models and uses a theorem prover (i.e., Z3) to show that various safety, architectural and program-specific properties hold. p4v optimizes the constraints passed to Z3 using techniques such as constant propagation and dead code elimination in order to scale the verification to larger programs.

The symbolic execution technique has been previously used to verify data planes. [8] proves that pipelines composed of Click elements satisfy *crash-freedom*, *bounded execution*, and packet filtering properties. The authors try to handle the path explosion problem by symbolically executing the Click elements separately. p4pktgen [28] uses symbolic execution to generate test cases for P4 programs. It applies backtracking techniques to prune unfeasible paths during the execution and thus reduce the search space. Symnet [34], in turn, is a verifier of data plane models built using the SEFL language, also proposed by the authors. This language contains instructions that simplify its symbolic execution, allowing the efficient verification of complex programs.

Vera [33] extends Symnet to support the verification of P4 programs. It automatically inserts checks that capture general safety bugs (e.g., invalid memory accesses) and also provides a property specification language based on Computation Tree Logic (CTL) for checking program-specific invariants. We believe our assertion language provides a more accessible way for P4 programmers to express intricate properties compared to temporal logic or even first-order logic. For example, domain specific methods such as extract_header and emit_header benefit from being largely drawn from the same language as the program under test (i.e., P4). Developing automatic ways to instrument P4 programs with assertions as well as providing a detailed performance comparison between our tool and Vera are interesting research directions for our work.

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.

8 CONCLUSION

We presented in this work an assertion language that can be used by P4 programmers to express correctness and security properties of a specific implementation. Our solution is more expressive than

[6] https://github.com/p4lang/behavioral-model

other data plane verification approaches, being the first work to allow proving properties specific to P4 source code and optionally the forwarding rules used by its tables. Our mechanism verifies the assertions using symbolic execution over C models automatically generated from the program and assertions.

We evaluated our approach by finding a broad range of bugs in real P4 programs found in the literature. The performance analysis of the proposed mechanism revealed that despite its efficiency in verifying small programs, the execution time grows exponentially with relation to the number of tables, actions, forwarding rules, and assertions. Thus, alongside our tool, we presented a range of techniques that can be used to speed up the verification time of complex programs. We also demonstrated in our experiments that combining the proposed optimization techniques we can reduce the verification time of non-trivial P4 programs in 81 percent.

As future work, we intend to explore the application of our approach in verifying network-wide properties of networks composed of P4 programs. The assertion language can also be investigated with the goal of providing the automatic insertion of assertions. These assertions could be used to verify general properties such as reading fields of invalid headers or checking the bounds of arrays. The P4 to C translation can be improved by proving the correctness of the process, as well as increasing the number of external objects modeled. Finally, the compiler flags and program slicing optimization techniques can be fine-tuned to our proposal by investigating optimization passes and slicing approaches optimal to our use cases.

ACKNOWLEDGMENTS

We are grateful to our shepherd, Cole Schlesinger, and the anonymous reviewers for their constructive feedback. This work has been supported by grants from RNP/CTIC (P4Sec), FAPERGS (APE), CNPq (201481/2017-0, 310408/2017-2 and 311088/2015-5), and also by CAPES/Brazil – Finance Code 001.

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] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 183–198. https://doi.org/10.1145/1966445.1966463

[4] 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

[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, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at Network Speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR '15)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2774993.2774999

[7] 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

[8] 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

[9] 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

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

[11] 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

[12] 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

[13] 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

[14] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 99–111. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[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] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A Software Analysis Perspective. *Form. Asp. Comput.* 27, 3 (May 2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[18] The P4.org language consortium. 2016. VSS Example. https://github.com/p4lang/p4c/blob/master/testdata/p4_16_samples/vss-example.p4. (2016).

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

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

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

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[23] 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

[24] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[25] 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

[26] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/2018436.2018470

[27] George Varghese Nuno Lopes Nikolaj Bjorner Andrey Rybalchenko Nick McKeown, Dan Talayco. 2016. *Automatically verifying reachability and well-formedness in P4 Networks.* Technical Report.

[28] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4Pktgen: Automated Test Case Generation for P4 Programs. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/3185467.3185497

[29] 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

[30] S. Signorello, R. State, J. Francois, and O. Festor. 2016. NDN.p4: Programming information-centric data-planes. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. 384–389. https://doi.org/10.1109/NETSOFT.2016.7502472

[31] 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

[32] 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.

[33] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 518–532. https://doi.org/10.1145/3230543.3230548

[34] 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

[35] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. 2016. Some Complexity Results for Stateful Network Verification. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Springer-Verlag New York, Inc., New York, NY, USA, 811–830. https://doi.org/10.1007/978-3-662-49674-9_51

[36] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557