POSTER: Finding Vulnerabilities in P4 Programs with Assertion-based Verification

Lucas Freire, Miguel Neves, Alberto Schaeffer-Filho, Marinho Barcellos
UFRGS
{lmfreire,mcneves,alberto,marinho}@inf.ufrgs.br

ABSTRACT

Current trends in SDN extend network programmability to the data plane through the use of programming languages such as P4. In this context, the chance of introducing errors and consequently software vulnerabilities in the network increases significantly. Existing data plane verification mechanisms are unable to model P4 programs or present severe restrictions in the set of modeled properties. To overcome these limitations and make programmable data planes more secure, we present a P4 program verification technique based on assertion checking and symbolic execution. First, P4 programs are annotated with assertions expressing general correctness and security properties. Then, the annotated programs are transformed into C code and all their possible paths are symbolically executed. Results show that it is possible to prove properties in just a few seconds using the proposed technique. Moreover, we were able to uncover two potential vulnerabilities in a large scale P4 production application.

KEYWORDS

P4; Verification; Programmable Data Planes

1 INTRODUCTION

Data plane programmability allows operators to deploy new communication protocols and develop network services with agility. Through the use of programming languages such as P4, it is possible to specify in a few instructions which and how packet headers are manipulated by the different forwarding devices in the infrastructure. Despite the flexibility provided by this paradigm, the security of data plane programs is a challenge that needs to be addressed before it can be widely adopted.

If, on the one hand, P4 adds a new programming axis (the network data plane), on the other hand, it also increases the chance of introducing bugs due to incorrect protocol implementations. Such bugs can be easily transformed into vulnerabilities if exploited towards the violation of network security policies. The traditional way to overcome this problem is by checking if the network satisfies the intended properties using formal verification techniques (e.g., model checking or symbolic execution). Several mechanisms have been developed in this direction [3, 5, 6], but none of them is capable of efficiently verifying security properties of (P4) data plane programs.

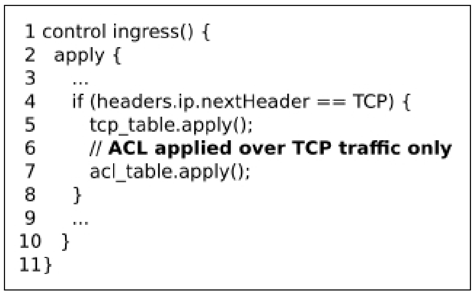
CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4946-8/17/10.
https://doi.org/10.1145/3133956.3138837

With the goal of enabling the verification of general correctness and security properties in P4 programs, we propose a mechanism based on assertions and symbolic execution. First, a P4 program is annotated with assertions expressing the intended properties. The annotated program is then automatically translated into a C-based model, which is symbolically executed by an engine that traverses all its possible paths. Our mechanism can prove if a property is satisfied by verifying if any execution path in the model violates the assertions specified.

We prototyped the proposed mechanism using the KLEE symbolic engine and the reference P4 compiler provided by the *P4 Language Consortium*[1]. Results show that the proposed mechanism is capable of verifying security properties in the order of seconds for programs expressing packet processing pipelines with up to 15 tables.

2 VERIFYING P4 PROGRAMS

The P4 programming language aims at enabling the data plane programming of network devices in a simple and architecture-independent manner. A P4 program describes how incoming packet bits are parsed into headers and manipulated by actions specified in tables, which in turn are organized in pipelines described by control blocks. Figure 1 shows an example of what would be a vulnerability in a P4 program. This code snippet specifies a packet processing pipeline containing two 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 bypass the filtering.



```
control ingress() {
 apply {
   ...
   if (headers.ip.nextHeader == TCP) {
     tcp_table.apply();
     // ACL applied over TCP traffic only
     acl_table.apply();
   }
   ...
 }
}
```

Figure 1: An example of vulnerability in a P4 program

Assuming that the network security policy disallows this kind of practice, the program in question could be used as the starting point of many attacks. Since current verification approaches do not enable expressing and automatically verifying security properties

[1]p4.org

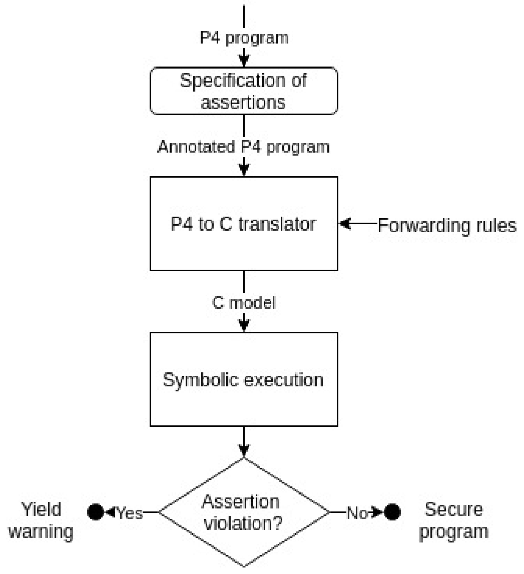


Figure 2: Control flow of the verification process.

of P4 programs, we propose a novel mechanism to identify this type of vulnerability.

Overview. The key idea consists of verifying models of the original programs annotated with assertions. Figure 2 shows the verification workflow. First, the P4 program developer or network operator annotates the code with assertions to guarantee general properties of interest. These properties can reflect a network security policy or simply represent the expected program behavior. Once annotated, a translation process takes place to automatically generate an equivalent program model using the C language. Optionally, forwarding rules can also be used as input to the translator to restrict verification to a given network configuration. The generated model is then verified by a symbolic engine, which either proves that the assertions are true for all the program execution paths, or reports assertion violations as soon as they are found. If no assertion is violated, the P4 program is considered secure regarding the analyzed properties. Otherwise, the respective violation is reported, allowing the developer or operator to correct the program.

Specifying assertions. To include assertions in a P4 program, we defined a specification language that allows the expression of properties of interest. Our language uses the code annotation mechanism available in P4 through an *assert* annotation. Each assertion *a* is composed of an expression *e* or method *m* ∈ *{forward, traverse_path, constant, if, extract_header, emit_header}*. The set of methods was designed to facilitate the specification of security properties commonly required (e.g., network isolation, header integrity, information flow). Semantically, each assertion represents a boolean that should evaluate to a true or false logical value. In this sense, expressions have the same semantics as their equivalent in the P4 language, while methods have their own semantics. For example, *traverse_path* indicates if a given program section was executed, while *constant(f)* is true if the header field *f* is not changed during program execution.

Constructing C models. Once a P4 program is annotated, our mechanism generates an equivalent C model through a translation process. The process as a whole comprises two steps: (i) transforming the P4 program into a directed acyclic graph (DAG); and (ii) transforming the generated DAG into C code using the nodes of interest. We use C structs to model P4 headers and translate the remaining P4 structures (i.e., tables, actions, control blocks, and parsers) to multiple C functions. Each assertion type is modeled in C using a particular approach. In general, we use global boolean values initially set to false. They are assigned to true in different locations of the model depending on the semantics of the assertion used. P4 externs (i.e., device specific functionalities) can be integrated into the translator through libraries.

Symbolically executing program models. The generated C model is verified by a symbolic engine, which will traverse all possible paths in the program while treating the incoming packet headers as symbolic values. The number of execution paths is essentially given by the number of tables and actions used. Whenever a table can only be accessed under some condition (e.g., depending on some specific protocol), a new execution path is created. The same happens whenever multiple actions can be invoked by a given table.

3 EVALUATION

The main goals of our experiments are: (i) to answer if it is possible in fact to find vulnerabilities in P4 programs using the proposed mechanism, and (ii) to assess how long it takes to verify properties of interest according to different program characteristics. We have prototyped the proposed mechanism using KLEE (version 1.3.0) as the symbolic engine. To build C models, we first convert a P4 program to its JSON representation using the reference compiler provided by the *P4 Language Consortium*, then we translate the JSON model (represented as a DAG) to C code using a translator implemented specifically for this purpose. The translator source code, as well as the scripts and workloads used were made publicly available online[2]. All the experiments were performed using a linux virtual machine (kernel version 4.8.0) with one 3 GHz core and 16 GB of RAM.

3.1 Security analysis

To demonstrate the potential of the proposed verification mechanism as a defense tool against network attacks originated from vulnerabilities in P4 programs, we have selected the DC.p4 program as a case study. DC.p4 was proposed by [4], and captures the behavior of a data center switch. We have chosen this program due to its complexity (more than 2500 lines of P4 code) and representativeness of real scenarios. We have verified it with the goal of finding vulnerabilities that can be exploited by potential attacks.

Code circumvention. The first set of performed experiments is related to the verification of vulnerabilities in security functionalities implemented by the program (e.g., VLAN and ACL to provide network isolation). We verify if it is possible to circumvent any of

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

these functionalities in such a manner as to bypass the enforcement of the corresponding security policies. We used assertions using the format *if(bypass == 0, traverse_path)* to express this type of property, where *bypass == 0* indicates that the security functionality is active (i.e., it was configured by the administrator), and *traverse_path* verifies if the functionality is being executed for every possible input.

Given the program complexity, the coverage of all feasible paths takes in the order of days to finish. However, it was not necessary to traverse all paths to find assertion violations. Our verifier was capable of finding paths containing violations of the tested properties in less than 10 seconds. We have found that even though the program supports multiple options of access control lists (e.g., level 2 and 3 lists), IPv4 packets do not go through a level 2 list, which can be transformed in a vulnerability depending on the configuration adopted by the network administrator (e.g., if the administrator decides to only configure a level 2 ACL).

Traffic amplification. Some functionalities of the DC.p4 program (e.g., packet mirroring) involve replication of traffic among the device ports, being usually part of a network monitoring task. If wrongly executed, such procedures can become the source of attacks based on packet replication or traffic amplification [2]. In this regard, we verify if the replication procedures in the DC.p4 program are correctly executed. We have used assertions in the format ! *(outport == original_port && constant(outport))* inside actions that replicate the traffic to express these properties. The first part of the assertion above tests if the outgoing port of the replicated packet is the same as of the original packet, whereas the second part tests if such port is changed during the replicated packet processing.

As in the previous case, our mechanism was able to find property violations quickly (less than 13 minutes). Overall, the program is general enough to allow the control plane to configure the outgoing port of both the original packet and its replica, while not applying any modification to the replica headers. Thus, it is the control plane responsibility to ensure that the device table configuration does not cause a single packet to be forwarded multiple times to the same destination, which is difficult to guarantee.

3.2 Performance

In this section, we assess how our proposed verification mechanism scales according to P4 program characteristics. To this end, we used the Whippersnapper [1] programmable data plane benchmark to synthetically generate P4 programs with different pipeline sizes (i.e., number of tables). Additionally, we tested the impact of varying the quantity of actions associated with each table. In this case, we consider a program with only two tables (with the goal of minimizing this factor in the verification time), using scripts to insert new actions.

As expected, the verification time grows exponentially to variations applied to the number of tables. Pragmatically, however, it takes around a minute to check programs containing pipelines with up to 15 tables (Figure 3(a)), which includes various existing P4 programs. We can also observe that the execution time grows polynomially with relation to varying the number of actions (Figure 3(b)). In this case, the degree of the polynomial depends on the quantity of tables of the P4 program (e.g., two for the tested program). In practice, given a program containing p sequential tables, where each table can invoke one among q different actions, a total of q^p distinct paths should be symbolically executed. Pragmatically, P4 programs do not usually have more than a dozen actions in each table, which makes the proposed mechanism feasible for the majority of the existing instances.

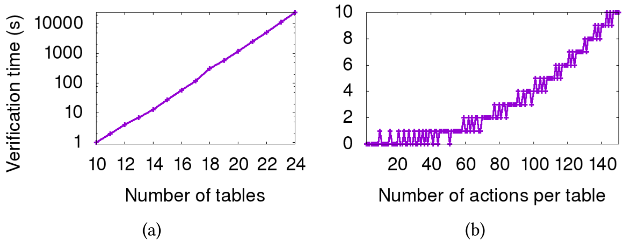


Figure 3: Performance analysis of the proposed mechanism.

4 CONCLUSION

We presented 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 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. We evaluated our approach by proving properties of traffic replication and circumvention of security functionalities in the DC.p4 program. Its performance analysis revealed that despite the efficiency in verifying small programs, the execution time grows rapidly with relation to the number of tables and actions. Therefore, we are working to extend SymNet, a high-performance network-oriented symbolic execution engine [6], to enable the scalable modeling of P4 programs.

REFERENCES

[1] 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

[2] Johannes Krupp, Michael Backes, and Christian Rossow. 2016. Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1426–1437. https://doi.org/10.1145/2976749.2978293

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

[4] 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

[5] 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.

[6] 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