A Simple Approach to Verify and Debug Data Plane Programs

Eduardo Castilho Rosa[1,3], **Italo Tiago da Cunha**[2,3], **Flávio de Oliveira Silva**[3]

[1]Department of Computer Science – Goiano Federal Institute (IFGoiano)
Catalão, GO – Brazil

[2]Faculty of Computing – Jatai Federal University (UFJ)
Jataí, GO – Brazil.

[3]Faculty of Computing – Federal University of Uberlândia (UFU)
Uberlândia, MG – Brazil.

eduardo.rosa@ifgoiano.edu.br, italo@ufj.br, flavio@ufu.br

Abstract. *The advances in data plane programmability through domain-specific languages such as P4 require the adoption of verification methods to ensure that a given code behaves appropriately. The standard approach in the literature is to use formal methods to verify a provided software. However, traditional techniques are often time-consuming and expensive. In this work, we propose a model to demonstrate the behavior of a P4 program that is simple and fast. To validate our model, we have used a name-based forwarding program and a control plane application that injects in a P4 table a total of 1 million random name prefixes. The results have shown that our model can quickly indicate whether or not a given program is behaving correctly.*

1. Introduction

Recent developments in the networking industry and academia has become possible the design and implementation of devices that is capable of forwarding packets at a high-speed while providing flexibility to network operators. The adoption of domain-specific languages such as P4 not only facilitate rapid innovation in this field but also opens up new opportunities for novel use-cases such as in-network caching [Jin et al. 2017] and in-band network telemetry, for example. In this regard, although data plane programmability offers a huge benefit to network operators as a whole, it also creates some challenges related to correctness.

To illustrate just how important is to make sure that a P4 code behaves properly, let's consider a firewall as an example. Suppose we have rules in this firewall to drop packets with destination IP address starting with 10. Once we have created a P4 code specifying such firewall and populating the corresponding tables with the rules, for a given packet P with IP address equals to $10.1.1.1$, we expect such packet being dropped. If the P4 code contains non-detected errors in such a way that it forwards the packet P anyway, the security of the firewall is compromised and sometimes such misbehaviour can not be detected right away. Therefore, it is really important to verify whether or not the P4 code is behaving as expected before we deploy it into a production network.

In literature, the most common way to verify a P4 code is by using formal methods. Generally, formal methods consists in formal specification by using mathematical models to specify the desired properties of a given system. The mathematical models are

usually expressed through a language whose syntax and semantics are formally defined. However, formal methods requires extensive training since only few network developers have the essential knowledge to implement it. Also, formal methods are time consuming and expensive. To address this problem, we propose in this paper a verification model to validate P4 programs that is simple and fast. The key idea of our method is to extract information such as input and output port from the log file generated by the BMv2 software switch, store them into a hash table and compare them with an expected set of input.

The remaining of this paper is organized as follows: Section 2 presents the related works. Section 3 presents the proposed model. Section 4 shows the experimental results and finally Section 5 provides the final considerations.

2. Related Works

Verification methods are common in software development. When it to comes to data plane programmability, such methods are also necessary to make sure the switches and routers process and forwards packets correctly. In [Stoenescu et al. 2018], the authors presents Vera, a verification tool that enables debugging of P4 programs both before deployment and at runtime. Vera verifies a P4 program exhaustively by using symbolic execution [Stoenescu et al. 2016]. Since verification is exhaustive, Vera can guarantee the P4 program is bug-free. Because Vera explore a large number of paths, one limitation is the verification process is time consuming and worst case time and space complexity is quadratic.

More recently, another P4 verification tool called p4v [Liu et al. 2018] is presented. Although p4v is based on classic verification methods, it adds a novel mechanism to make assumptions about the control plane. In contrast to Vera [Stoenescu et al. 2018], p4v avoids explicit run-time traversals of the P4 program, reducing the time consuming in the verification process. However, a limitation of p4v is that control-plane interfaces must be written by hand which impairs the scalability. Also, p4v can not run successfully some P4 programs such as HyperP4 [Hancock and van der Merwe 2016] because the Z3 theorem prover [de Moura and Bjørner 2008], used to solve the formulas that captures the execution of the program, ran out of memory.

3. Verification Model

Our model is represented by four modules shown in Figure 1. The control plane entity is responsible for generating the table entries and install them into the P4 tables by using the *runtime_CLI* or *P4Runtime API*. Such control plane APIs enables the sending of a batch of entries from a text file. Thus, in our model, we select a group of random prefixes from a public domain dataset and we set random output ports for each one of them.

As a case study, we have implemented a name-based forwarding algorithm proposed in [Rosa and Silva 2022]. In summary, [Rosa and Silva 2022] introduces an alternative way to perform the longest name prefix matching (LNPM) in Named-Data Networking (NDN). The key idea is to create multiple tables with different bit-width to accommodate name prefixes of different lengths. The LNPM is performed by using packet recirculations in the pipeline. Since the LNPM is a process that involves quite a number of operations in the data plane, to avoid misbehaviour in forwarding the packets, it is crucial to verify weather or not the P4 name forwarder is sending packets to the correct output

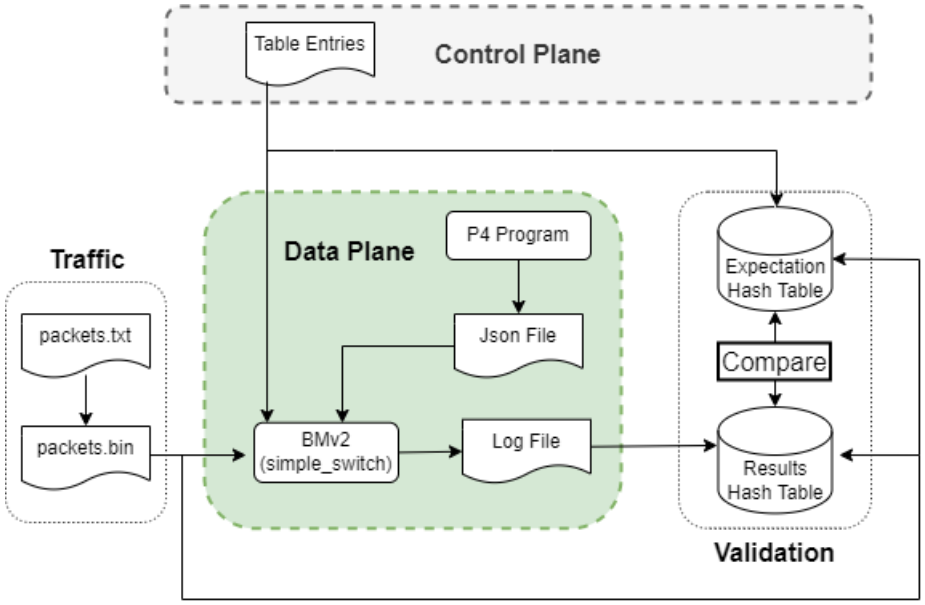

Figure 1. Operational flow in the verification model and its main components.

ports. As such, the table entries corresponds to name prefixes and the P4 code performs the longest name prefix matching for each packet that cross the pipeline.

The traffic generator is responsible for injecting on BMv2 NDN interest packets containing content names. The traffic is generated into two steps. First, a file called packets.txt is generated by selecting random domain names from the dataset [DomCop.com 2021]. Each line in the file packets.txt is a 4-tuple containing the following fields: <*id, prefix, inputPort, expectedOutputPort*>. Whenever a given packet is received from an input port or is sent to an output port, the BMv2 executable enables dumping the packet content in the log file. Then, we may use the packet content hash to identify each packet. However, the traffic generator can inject multiple copies of the same packet into the BMv2 and this approach impairs the uniqueness of identification. To solve this problem, each packet has an unique id controlled by the applications. The input port is random and the expected output port is obtained by performing a traversal in a trie data structure to determine the longest prefix matching. For simplicity, we assume such algorithm is correct. The second stage of the traffic generator is to convert the packets.txt file into a binary file called packets.bin to be used as input in *scapy* software. The structure of this file is a sequence of 3-tuple <*inputPort, packetSize, packet*>. The input port is generated randomly to avoid backpressure in one single port. The packet size is a 2-byte field indicating how many bytes the subsequent packet has. The data format of the packet is the same as proposed in [Rosa and Silva 2022].

In the data plane, one instance of BMv2 software switch called *simple_switch* is executed by using the *v1model* architecture. The *simple_switch* generates a log file that registers the main operations performed on each packet. Such operations includes receiving a packet, parsing the headers, performing the table lookup, and so on. The packet id is not registered in the log file by default. To do so, we uses the log_msg construct that is part of the P4 specification and allows us to log customized messages.

Algorithm 1 Data plane program validator

Input: H_1 = Expectation hash table and H_2 = Results hash table
Output: *yes* or *no*

```
validated ← true
for each k ∈ K and K = {keyset of H1} do
    expected_value ← H1(k)
    obtained_value ← H2(k)
    if obtained_value exists in H2 then
        id ← obtained_value.id
        ip ← obtained_value.input_port
        op ← obtained_value.output_port
        if expected_value.input_port ≠ obtained_value.input_port then
            print "Pck:"+id+" Expected input port: "+ip+ " Obtained input port: "+op
            validated ← false
        end if
        if expected_value.output_port ≠ obtained_value.output_port then
            print"Pck:"+id+"Expected output port:"+ip+"Obtained output port: "+op
            validated ← false
        end if
    else
        print"Pck "+id+" does not exist"
    end if
end for each
if validated then
    print "passed"
    return yes
else
    print "failed"
    return false
end if
```

The core of our model is the validation module. It takes the log file produced by the BMv2 as input and uses the *grep* tool to filter out the lines that contains the words "*packet_id*", "*Received*" and "*Sending*". The former identify the customized line that we set into the log file through the construct *log_msg* and it contains the following information: <*timestamp*, *switch_type*, *bmv2_thread*, *packet_id*, *pipeline_passes*>. The two latter identify the lines that corresponds to the events "*receiving a packet*" and "*sending a packet*", respectively. Besides the information aforementioned, these lines also includes information such as *packet_length*, *input_port*, *output_port* and the *packet_content* in hexadecimal.

The information extracted from the log file are stored into a hash table H_1 by using a hash function $h : K \rightarrow V$, where K is the key set represented by the packet ids and V is the associated value represented by the *7-tuple* <*receive_timestamp*, *send_timestamp*, *input_port*, *output_port*, *packet_length*, *pipeline_passes*, *latency*>. On the other hand, the name prefixes that we insert into the P4 tables are first stored into a trie data structure.

Then, we perform a search in the trie to determine the output port corresponding with the longest prefix. We uses such information to create a second hash table H_2 storing the names with its corresponding correct output ports. To verify whether or not a given P4 code produces the correct output, we compare the two hash tables element by element as we can see in Algorithm 1.

4. Evaluation and Preliminary Results

As a case study, we have use the name dataset [DomCop.com 2021] to implement the NDN forwarding algorithm in [Rosa and Silva 2022]. Such dataset contains 10 million domain names and we filter them out following the rules:

- We eliminate urls that contains duplicated name components.
- For urls with more than 8 name components, we take just the 8 first ones;
- We reverse the url to be NDN-friendly;
- For each url, we check with probability p whether or not the such url will contain all its sub-prefixes;
- We generate random output ports for each url;

The experiment was conducted on two different platforms. First, we have executed one instance of the BMv2 on Ubuntu 16.04 Virtual Machine. The BMv2 was configured with 8 data ports as well as 1 CPU port to send the packets to the control plane for additional processing. The table entries corresponds to 1 million name prefixes and all its sub-prefixes ($p = 1$) extracted randomly from the filtered dataset. Second, the traffic generator and the validation process is performed on Windows 10 with Intel(R) Core(TM) i7-4600M CPU @ 2.90GHz and 8GB of RAM. The traffic consists of ten thousand packets randomly picked from the dataset and it is injected on BMv2 by using *scapy* in intervals of 100ms to avoid packet queuing. Our algorithm was capable of performing the validation in only 32 seconds. Figure 2 shows the graphic interface of our validator.

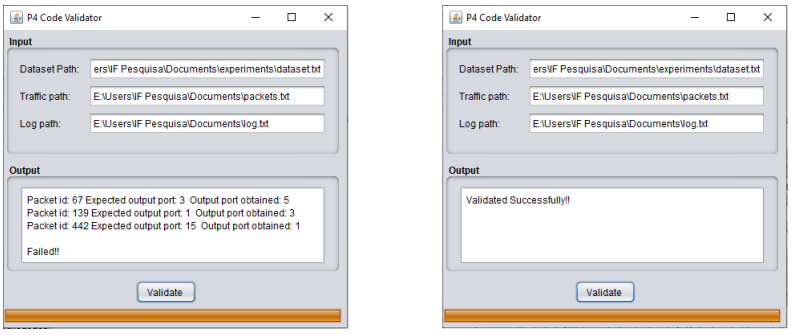

Figure 2. Graphic interface of our P4 code validator.

In contrast to verification methods that uses symbolic execution such as [Stoenescu et al. 2018], one limitation of our model is the need to execute a P4 code and to inject traffic to generate the log file. Thus, the validation time depends on how many packets the BMv2 process. As future works, we can extend our model to extract not only information such as input and output ports but also information related to parsing states, match action operations and conditional statements.

5. Considerations

Validation and verification techniques are required in software developments to ensure correctness. Data plane programs written in domain-specific languages such as P4 can be very complex and, therefore, the use of verification tools can be beneficial in networking. In this paper we have proposed a simple tool to verify a P4 code by using information extracted from the log file generated by the BMv2 software switch. We do not intend to provide a complete verification tool to prove the correctness of any P4 program. Instead, the idea is to provide a simple and quick alternative to make sure that a given P4 code, when deployed on a programmable switch, produces the correct output. As future work we are planning to extends our model to support a wide range of P4 programs.

References

de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

DomCop.com (2021). Top 10 million websites. https://www.domcop.com/top-10-million-website. last accessed: May 1, 2021.

Hancock, D. and van der Merwe, J. (2016). Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 35–49, New York, NY, USA. Association for Computing Machinery.

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA. Association for Computing Machinery.

Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., and Foster, N. (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, page 490–503, New York, NY, USA. Association for Computing Machinery.

Rosa, E. C. and Silva, F. d. O. (2022). A hash-free method for fib and lnpm in icn programmable data planes. In *2022 International Conference on Information Networking (ICOIN)*, pages 186–191.

Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. (2018). Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 518–532, New York, NY, USA. Association for Computing Machinery.

Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. (2016). Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 314–327, New York, NY, USA. Association for Computing Machinery.