Toward Highly Reliable Programmable Data Planes: Verification of P4 Code Generation

Csaba Györgyi
University of Vienna, Austria
ELTE Eötvös Loránd University, Hungary

Sándor Laki
ELTE Eötvös Loránd University, Hungary

Stefan Schmid
TU Berlin & Fraunhofer SIT, Germany
University of Vienna, Austria

2023 IEEE 9th International Conference on Network Softwarization (NetSoft) | 979-8-3503-9980-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/NetSoft57336.2023.10175397

***Abstract*—Data plane programming gained much attention in the past years, having a fast-growing community both in academia and industry. Many tools have emerged to simplify and/or help the development of reliable data plane programs, including fuzzing, formal verification, and different code generators. However, even the tools themselves must be verified to meet the most stringent dependability requirements. In this paper, we investigate various tools and methods to verify code generators leveraging P4 through the example of P4RROT (an open source code generator focusing on the application layer). We show that our approach is efficient and can indeed successfully find bugs. We identify two bugs and propose reusable ideas, such as the use of ghost code.**

***Index Terms*—code generation, in-network computing, data plane verification, P4**

I. Introduction

Many novel applications leveraging the high-throughput and ultra-low latency of programmable data planes have recently been proposed in the literature [1]. Certain applications — including complex event detection, caching, data aggregation, and in-network security, among many others — describe quite complex functionalities. One way to increase productivity and adaptability in software development is by using higher abstraction levels. Just like it is much faster to implement the same functionality in Python than in Assembly, the different levels of abstraction are also observable in the world of programmable data planes.

However, higher abstraction levels require dependable tools that accurately translate the specified high-level intentions to a lower-level code (e.g., P4). In this paper, we focus on the verification of the open source P4 code generator called P4RROT [2], [3] that compiles Python constructs to P4. Although P4RROT is a single tool, the challenges we encounter are common to other similar tools, and the methodology we propose can easily be adapted to other data plane code generators.

Our contributions are as follows:

- We analyze and decompose the verification task of P4RROT;
- We investigate different state-of-the-art and state-of-the-school tools and methods, and carry out an evaluation based on P4testgen [4] and Atheris;
- We derive reusable methodologies, such as the use of ghost code.

After developing and implementing our method, we found two real bugs in the P4RROT source code, showing the need for verification methodologies of data plane code generators. Our method can increase the trust in the code generator tools in an easy-to-integrate manner. Moreover, this case study led us to a better understanding of how to further develop P4RROT, and the observations we have taken could also be applied to other code generators.

II. Motivation

A. Software-Defined Networking and P4

Software-Defined Networking divides a programmable network device into two main components: the Data Plane describes the packet processing pipeline and the Control Plane that configures the first one. While control plane programmability has a long history (e.g.: OpenFlow), data plane programmability is relatively new. One of the most popular data plane programming languages is P4.

P4 is a domain-specific language tailored for packet processing. A P4 program always describes the processing of a single packet based on the architecture of the target device (e.g.: software switches, smartNICs, NetFPGAs or even programmable ASIC switches such as the Intel Tofino). Since P4 is a domain-specific language, it has different strengths and limitations compared to a general-purpose language. Different targets can pose additional constraints to enforce or even guarantee fast and efficiently executable solutions.

B. Code generators

Broadly speaking, we use different abstraction levels during the software development process to transform high-level requirements and designs into low-level deployable artefacts. Part of the process is manual (e.g., high-level UML diagram to a more detailed one), while others can be automated (e.g., C to Assembly). Since humans deal better with higher levels of abstraction, automating the majority of the development process offers benefits in development time and costs. Moreover, these tools often build on top of one another. A well-known example is the different database management libraries such as Hibernate for Java. They generate SQL tables and SQL queries based on code written in a general-purpose language. A more exotic example is AtelierB which provides a programming language suited for formal verification of the program that can be later exported into Ada or C.

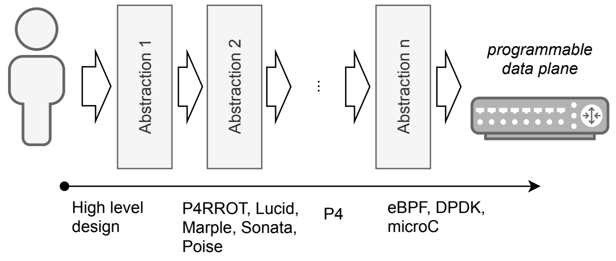


Fig. 1. Different tools offer different abstraction levels for programmable data planes.

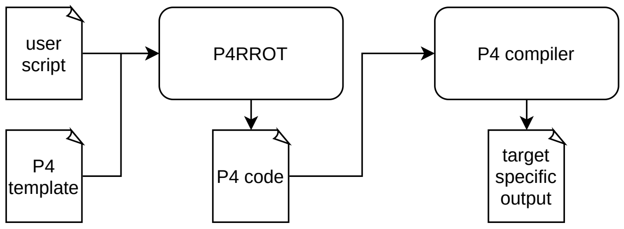


Fig. 2. The high-level overview of the use of P4RROT. [2]

The same holds for programmable data planes as well. Figure 1 depicts different data plane programming tools corresponding to different abstraction levels. Target devices usually require a deployable binary to execute the implemented behaviour which is not suited for human beings. DPDK, eBPF, and microC allow us to use C (or a restricted/slightly modified version of it) to describe our solution using a well-known language. P4 offers a higher level of abstraction tailored to packet processing. There are even compiler backends that transform P4 code into C using DPDK or eBPF. Thanks to P4's success, many tools started to build on top of P4. Lucid [5] is a programming language providing an event-driven abstraction for implementing control functionalities for the Intel Tofino switch. Sonata [6] offers a scalable and flexible query language to collect information about the network traffic. Poise [7] is able to define context-aware security policies. Marple [8] offers real-time performance monitoring using its Python-like query language. Finally, P4RROT [2], [3] aims to generate code for the application layer and general computing tasks. These exampples all emit P4 code among many others [9]–[11].

Verifying the correctness of automated translations is crucial in high-stake projects that require reliable tools.

C. P4RROT introduction

P4RROT [2] is a code generator library aiming to speed up the prototyping and development of data plane programming projects. It mainly focuses on the application layer and general computing tasks.

The use of P4RROT is depicted in Figure 2. The code generator takes 2 inputs: 1) a P4 template code, 2) the custom logic described in Python using the library. The result is a P4 program that can be compiled to the desired target using the respective compiler.

The templates are usual P4 programs describing often used functionalities and referencing generated files using the #include pragma.[1]

The Python script describes two kinds of main components: FlowProcessors and FlowSelectors. FlowProcessors describe the computation logic using so-called Commands that can hide very simple and very complex behaviours. Flow Selectors are responsible for picking the packets that are subject to the defined computation.

To promote early testing, P4RROT offers a mechanism to simulate the behaviour of the described computations. Besides the code generation process, commands also contain the expected behaviour described in Python.

P4RROT is a very thin code generator focusing on tricks and workarounds to overcome limitations and compiler bugs. A successful code generation does not guarantee that the P4 compiles to the target. Instead, it helps the programmer to detect bottlenecks as fast as possible and quickly alter the solution.

D. Testing and Validation of P4

To make P4 solutions reliable many methods and tools have been already proposed in the past. Some of them target the testing of individual programs while others hunt for bugs in the compilers.

For those looking for a simple unit-test-like experience, the STF and PTF frameworks offer a quick solution. FP4 offers line rate grey-box fuzzing for P4 programs using an extra switch generating packets and evaluating the results. P6 is a reinforcement learning guided fuzzer that can also automatically patch P4 programs in certain cases. P4testgen [4] uses Z3 semantics to generate input packets for maximum path coverage. Similarly, Gauntlet uses Z3 semantics to compare the meaning of the program between compiler steps. It also generates input packets that trigger the detected differences. P4Fuzz also targets the compiler. It generates random P4 programs and compares the yielded output using different targets (differential testing).

Besides the mentioned ones, many other projects investigated the subject [12]–[19].

III. Methods and Implementations

A. Problem analysis and challenges

A usual way of testing compilers and code generators is finding inputs that crash the compiler. P4RROT's method is to quickly identify what are the resource bottlenecks and architectural constraints through try-and-fail rounds and quickly find an alternative solution. In this context, the role of P4RROT is to quickly identify failing implementations. Based on this consideration, finding inputs that result in non-compilable P4 code is not helpful.

Moreover, P4RROT does not perform strict semantic validations making it very easy to generate P4 code, which will most certainly fail. Because of the lack of semantic checks, it

[1]Most P4 pragmas have the same meaning as in C or C++.

is challenging to create meaningful code without the original P4 compiler.

FlowSelectors are fairly simple, while the abstractions of FlowProcessors can hide very complex behaviour. Thus, we expect to find the majority of defects in the latter one and thus we focus on this in our study.

How can we ensure that when a meaningful FlowProcessor generates a compiling P4 code, the P4 code behaves as expected? FlowProcessors use so-called Commands and StatefulElements to describe their behaviour. One approach is to require and define a formal specification for these elements and compare it with the formal semantics of the generated P4 code. Although this seems possible since prior work achieved significant results in this area, providing a formal specification seems impractical for an average tool developer. A second approach is to leverage the built-in simulation capabilities of P4RROT. P4RROT's design allows programmers to express the Python equivalent of their generated P4 code, thus promoting the early testing of algorithms. We decided to compare the behaviour of the Python code and the P4 code.

Besides ensuring that the generated code does what it should, we must also check that generated code segments do not interfere with each other (e.g., defining 2 helper variables under the same name).

We decompose the testing task into 2 subtasks. First, we should show that the Commands indeed calculate what they are expected to do. Second, we must ensure that the internal workings of different code parts and representations are isolated.

B. Functional correctness

Our main goal is to find bugs by comparing the results of the P4-based calculations and the Python-based simulations. A simple state-of-the-school approach is differential testing.

To derive test cases, we can use both the Python simulation and the generated P4 code. Test cases based on the Python simulation could reveal bugs related to faulty code generation. On the other hand, the P4 code-based test generation could reveal too simplistic simulations, for example not taking into account the overflows and underflows of the P4 language.

We would like to automate the test case generation as much as possible. As mentioned before, generating meaningful FlowProcessors is challenging. Moreover, a Command or Stateful element is excerciseable by a single well-designed FlowProcessor. On the other hand, generating test inputs (test packets in this case) for a given program has a variety of tools available. Based on these considerations, we write FlowProcessors manually that exercise a specific abstraction, while the test inputs are generated automatically using available tooling.

Generating input packets based on the Python simulation we experimented with both Hypothesis and Atheris. Hypothesis is a property-based testing library, while Atheris is a coverage-guided fuzzer. Due to page limitations, we only show evaluation results for Atheris. Note that Atheris proved to be more powerful than the black-box testing approach of Hypothesis.

Generating test cases based on P4 programs is also possible thanks to a big variety of analyzers and fuzzers. Our final choice is P4testgen, a young project lately integrated into the P4 reference compiler suggesting serious long-term support. Moreover, it promises to support multiple target architectures. It performs symbolic execution after converting the P4 code into Z3 expressions. It can emit input packets that execute all reachable paths. After feeding the input packets to the Python simulator and a real P4 deployment, we can compare the results.

We further improve the effectiveness of the P4-based test case generation by introducing the concept of ghost code to P4RROT. Ghost code is a piece of program, that is available for the compiler but does not affect the generated executable. We implemented special Commands that generate P4 code guarded by #ifdef pragmas. With the appropriate compiler flag, this will be included in the p4testgen analysis but not in the compiled code that processes the input packets.

To illustrate the applicability of our ghost code extension we present two different use case scenarios. First, in the case of testing arithmetic operations (e.g., addition, subtraction), we produce a sequential pipeline that can be discovered with a single input packet containing only 0 bits. Using ghost Commands, the tester can give further guidance to the symbolic executor troughout artificially introduced execution paths (e.g., checking for boundary values). A second use case could be to force p4testgen to generate input packets for interesting value constellations in the middle of the pipeline (e.g., could this intermediate value be 0, could these 2 values be equal, etc.).

Ghost code can be introduced through not just dedicated Commands but also by the developer of the component (e.g.: Command, StatefulElement) to increase the testability of the solution.

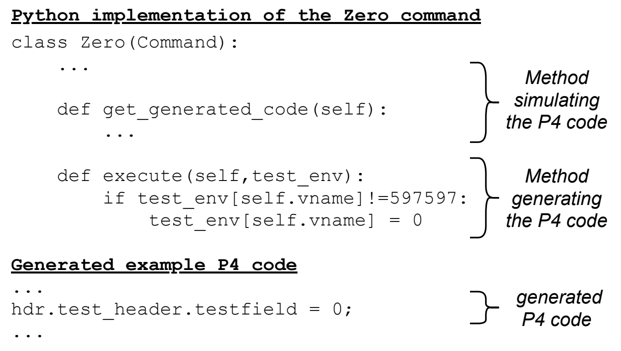
C. Isolation

Our second subtask is ensuring that Commands and StatefulElements do not interfere with each other's operation. The most probable way of this happening is by using the same variable, or the same name for different purposes.

P4RROT has a built-in method for generating unique identifiers that are used as a postfix in the generated name to help prevent this kind of issue. Static analysis of the generated P4 code can help in checking whether the code generation actually uses appropriate names or not. A big advantage of static analysis is that the generated P4 code does not have to comply with any resource constraints or other limitations since we do not have to execute the code on a real target.

Only considering names requires the partial parsing of the program's abstract syntax tree (AST), simplifying our work. Writing a fully functional parser for every P4 architecture would be quite time-consuming. Instead, we take the source code and feed each line to a simplified parser identifying the names. To this end, we use Lark, a parser generator library.

We should check every generated P4 identifier including but not limited to: variables, tables actions, action- and function parameters. If variable-, table and action names are used more



Python implementation of the Zero command

```
class Zero(Command):
    ...

    def get_generated_code(self):
        ...

    def execute(self,test_env):
        if test_env[self.vname]!=597597:
            test_env[self.vname] = 0
```

Method simulating the P4 code
Method generating the P4 code

Generated example P4 code

```
...
hdr.test_header.testfield = 0;
...
```

generated P4 code

Fig. 3. Zero command with an artificial bug in the Python simulation.

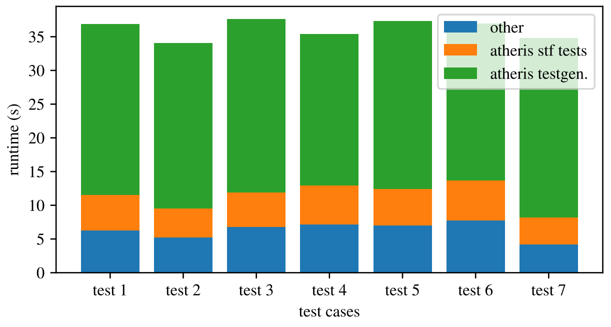
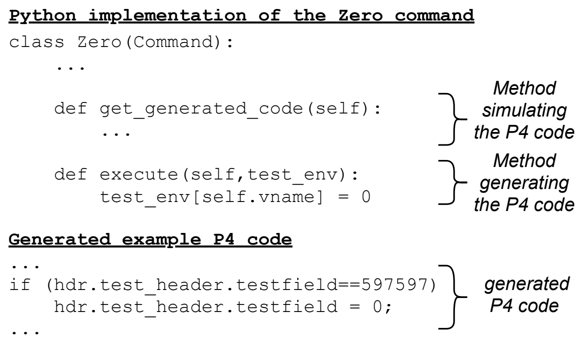


Fig. 4. Runtime of Atheris.



Python implementation of the Zero command

```
class Zero(Command):
    ...

    def get_generated_code(self):
        ...

    def execute(self,test_env):
        test_env[self.vname] = 0
```

Method simulating the P4 code
Method generating the P4 code

Generated example P4 code

```
...
if (hdr.test_header.testfield==597597)
    hdr.test_header.testfield = 0;
...
```

generated P4 code

Fig. 5. Zero command with an artificial bug in the P4 code generation.

than once, the P4 compiler should also catch them as double declarations making them less dangerous and easier to detect. However, shadowing of names can be more dangerous as we see through a real-world bug detected by our tool in the Subsection IV-C.

IV. Evaluation

A. *Effectiveness of Atheris-based testing*

To investigate the effectiveness of the Atheris-based testing (for Python), we extended P4RROT with a synthetic `Zero` command that sets the value of a given variable to 0. We intentionally introduced a bug to the python simulation: it only performs the required assignment when the variable's current value is not equal to a predefined constant (597597). Figure 3 highlights the relevant snippets of the implementation and the generated P4 code.

Our Python-based fuzzing using Atheris indeed discovers this bug. The fuzzer was started to run a maximum of 100000 calls. During the fuzzing, we save the inputs and outputs. Later, we convert these inputs into STF test cases and execute them to compare the output of the Python-based simulation against the output of the generated P4 data plane.

This example also demonstrates the usefulness of Python-based testing. Catching this bug based on the generated P4 code is almost impossible, the output of the P4 code (i.e., the effect of `Zero` command) is independent of the input.

Finally, we also investigated the time required to perform the Atheris-based testing. Figure 4 depicts the execution time for a representative set of test cases. The test was run using a single fuzzing process. The time spent on testing can be divided into 3 main parts: 1) generating the inputs and discovering the Python code using Atheris, 2) running the STF tests and 3) other setup and conversion logic. Generating the test cases accounts for the majority of the runtime. This can be further divided into instrumenting the functions and running the fuzzer itself. The instrumenting of every function takes 18-26 seconds.

These numbers can be further improved, by instrumenting only the simulation code instead of instrumenting every available function. However, to do it in an efficient and easy-to-automate way, we need to slightly alter P4RROT's design. By placing the functions responsible for the simulation in a different Python module, one can leverage the `instrument_imports` of Atheris to instrument only the necessary code parts. By separating the Command and its simulation we also open an easy way to use different simulations for the same command, which can be useful when working with approximations and probabilistic data structures.

B. *Effectiveness of P4Testgen-based testing*

Similarly to the evaluation of the Atheris-based testing, we implemented the `Zero` command with an artificial bug in the P4 code generation as depicted in Figure 5.

Since p4testgen discovers every possible path in the P4 code, it also discovers the hidden bug. Also, the P4-based testing can not be replaced by the Python-based fuzzing, since the Python simulation does not depend on the input at all.

Figure 6 presents the runtime of the same tests as presented for the Atheris-based testing. We make three key observations: 1) it usually requires much less time, 2) the generation of test cases is not the major part of the process in most cases, and 3) the exception is the first test case exercising the simple arithmetic operations. The first one can be explained by the different nature of the fuzzing and the constraint solvers. Fuzzing generates many inputs and tries to mutate them in a way to discover bigger new parts and paths of the code, while p4testgen intentionally creates a single packet for every possible path using the Z3 semantics of the code. The second one is explained by the fact, that although test generation takes less time, some aspects are independent of the number of test cases, e.g., compiling the P4 code, and starting a switch. The third one is caused by the excessive use of ghost code

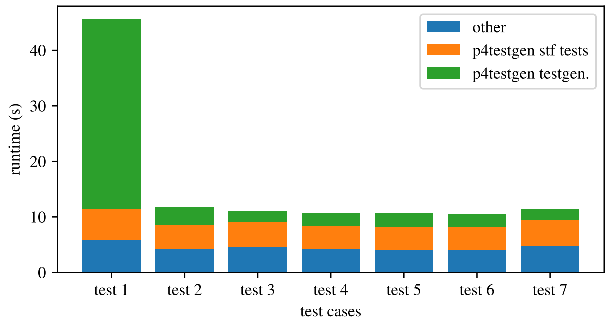


Fig. 6. Runtime of P4testgen.

in the first test case. Introducing new independent branches exponentially increases the number of paths the constraint solver has to account for. This also warns us, that solutions with too many branches can overwhelm p4testgen.

C. *Effectiveness of Static Analysis*

Each test finished within 0.07 seconds, showcasing the practicality of our solution. These results are not unexpected, since we parse every line separately not leaving room for very complex abstract syntax trees.

The analysis raised 24 warnings in total, 14 of which were false-positive. 9 out of the 10 remaining warnings were caused by the same bug, and there was another independent bug. Thus, we managed to discovered 2 real bugs in P4RROT with the use of SA.

The first bug is relatively simple and harmful. P4RROT allows the use of variables that are declared within a control block but outside of the `apply` block. In P4RROT, it was treated as a local variable, while in P4 it is more similar to a global one. Automatically appending unique IDs to the end of these variables can quickly fix the bug. Also, it is relatively harmless, since the P4 compiler would also detect if two variables have the same name.

The second bug is trickier and the P4 compiler would not detect it. The code generator uses Python f-strings to generate certain code parts, which allows the programmer to easily insert variables in the middle of strings while converting them into text. E.g., `f"calculation(int {vname}){ ..."` inserts the value of `vname` into the text. For simplicity reasons, we use this analogue example. In our particular case, `vname` is an optional parameter with a `None` default value. Since `None` can be converted to a string, the code produced the `f"calculation(int None){ ..."` output without any error. However, this `None` is a perfectly valid P4 identifier as well, that is able to shadow another None value thus potentially altering the meaning of the program without raising any errors during compilation.

V. Conclusion and Future work

We embarked on a journey of verifying the code generators through the P4RROT case study. After experimenting with different tools and methods, we found two bugs and proposed methods such as the use of ghost code that can help in verifying other P4 code generators. Though we have already increased our confidence in the code generator, certain aspects remain for future work like testing approximations, random behaviour, or target-specific state transitions.

Acknowledgment

Research supported by the Vienna Science and Technology Fund (WWTF), project WHATIF, grant 10.47379/ICT19045, 2020-2024, and the Austrian Science Fund (FWF), grant I 5025-N (DELTA), 2020-2024 S. Laki thanks the support of National Research, Development and Innovation Office - NKFIH, FK_21 138949.

References

[1] O. Michel *et al.*, "The programmable data plane: Abstractions, architectures, algorithms, and applications," in *Proc. ACM Computing Surveys (CSUR)*, 2021.
[2] C. Györgyi, S. Laki, and S. Schmid, "P4rrot: Generating p4 code for the application layer," *SIGCOMM Computer Communication Review (CCR)*, 2023.
[3] "P4rrot: Generating p4 code for the application layer," 2022. [Online]. Available: https://github.com/gycsaba96/P4RROT
[4] F. Ruffy *et al.*, "P4testgen: An extensible test oracle for p4," 2023.
[5] J. Sonchack *et al.*, "Lucid: a language for control in the data plane," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 731–747.
[6] A. Gupta *et al.*, "Sonata: Query-driven streaming network telemetry," in *Proceedings of the 2018 conference of the ACM special interest group on data communication*, 2018, pp. 357–371.
[7] Q. Kang *et al.*, "Programmable {In-Network} security for context-aware {BYOD} policies," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 595–612.
[8] V. Nathan *et al.*, "Demonstration of the marple system for network performance monitoring," in *Proceedings of the SIGCOMM Posters and Demos*, 2017, pp. 57–59.
[9] T. Jepsen *et al.*, "Forwarding and routing with packet subscriptions," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 282–294.
[10] E. O. Zaballa and Z. Zhou, "Graph-to-p4: A p4 boilerplate code generator for parse graphs," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–2.
[11] J. Vestin, A. Kassler, and J. Åkerberg, "Fastreact: In-network control and caching for industrial control networks using programmable data planes," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2018, pp. 219–226.
[12] A. Shukla *et al.*, "Toward consistent sdns: A case for network state fuzzing," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 668–681, 2019.
[13] Y. Zhou *et al.*, "P4tester: Efficient runtime rule fault detection for programmable data planes," in *Proceedings of the International Symposium on Quality of Service*, 2019, pp. 1–10.
[14] Y. Zhao *et al.*, "Pronto: Efficient test packet generation for dynamic network data planes," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 13–22.
[15] R. Doenges *et al.*, "Petr4: formal foundations for p4 data planes," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, 2021.
[16] Q. Kang *et al.*, "Probabilistic profiling of stateful data planes for adversarial testing," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 286–301.
[17] R. Stoenescu *et al.*, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 518–532.
[18] J. Liu *et al.*, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication*, 2018, pp. 490–503.
[19] A. Liatifis *et al.*, "Advancing sdn from openflow to p4: A survey," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–37, 2023.