In-Network Security Applications with P4RROT

Károly Kecskeméti, Csaba Györgyi, Péter Vörös, Sándor Laki
ELTE Eötvös Loránd University
{cgsmef,gycsaba96,vopraai,lakis}@inf.elte.hu

ABSTRACT

Computer networks have become a key infrastructure for many different business domains. Ensuring the security of such interoperable systems is essential in many areas. The emergence of in-network computing and data plane programmability has opened the door to novel security approaches. Although the logic behind most novel solutions is simple, their implementation in P4 is often complex for a non-domain expert or requires problem-specific languages and code generators. Existing in-network approaches only solve specific subproblems and are not general purpose. In this paper, we show how the open-source P4 code generator called P4RROT can simplify the implementation of various in-network security applications. To demonstrate its applicability, we reproduce three recent P4-based security methods. During the implementation, we extended P4RROT with new primitives needed for such applications and also added support for Intel Tofino ASICs. The complexity of the corresponding P4RROT codes is a magnitude lower than the investigated original P4 programs.

CCS CONCEPTS

• **Networks** → **Network security**; **Programmable networks**;

KEYWORDS

Network Security, Code Generation, P4

ACM Reference Format:
Károly Kecskeméti, Csaba Györgyi, Péter Vörös, Sándor Laki. 2023. In-Network Security Applications with P4RROT. In *The Twenty-fourth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc '23), October 23–26, 2023, Washington, DC, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3565287.3617612

MobiHoc '23, October 23–26, 2023, Washington, DC, USA

ACM ISBN 978-1-4503-9926-5/23/10.
https://doi.org/10.1145/3565287.3617612

1 INTRODUCTION

Quickly reacting to changes in a fast-paced, agile world is crucial. Building prototypes and speeding up the development process is beneficial for almost every IT project. One of these constantly changing areas is network security. Delaying a response to newly discovered security vulnerabilities can incur costs of up to 4.35 million USD (the average cost of a data breach), as reported in [11]. Software Defined Networking and programmable data planes have opened the door for in-network security solutions. By investigating security issues in the packet processing pipeline, these in-network methods have the advantage of detecting and even mitigating malicious activities at line rates. They also offer network-wide plug-and-play solutions without the need for modifying the end-hosts. P4 has proven useful for developing custom and standard security functions in the data plane. However, data plane program development in P4 still requires more or less domain-specific knowledge, esp. in cases of hardware targets (e.g., Tofino ASIC). To speed up the reconfiguration and the customization of their pipelines, P4Guard [3] and Poise [8] introduce an extra abstraction layer on top of P4. In addition to application-specific approaches, P4RROT [6] was recently proposed as a general purpose P4 code generator. P4RROT is a P4 code generator library written in Python, mainly focusing on offloading application-layer logic and general computing tasks to the data plane. The code generator takes two inputs: a P4 template code and the high-level description of the required processing using P4RROT's API. The template itself is a directory containing a skeleton of a general P4 program. It also needs to have certain hooks (e.g., define macros and include files) where the code generator can inject the custom logic. The custom processing is described using P4RROT's API. It heavily builds on method chaining to provide an intuitive interface similar to TensorFlow, Keras, and other popular libraries. P4RROT produces P4 code that can be later compiled to the target device.

In this work, we investigate whether the descriptive capability of P4RROT is sufficient to reproduce in-network security applications and what advantages we obtain compared to native P4 implementations. P4RROT is currently in a relatively early stage. During our study, we had to extend the library with additional functionalities needed to reproduce the selected network security methods. Moreover, it only supported the V1Model architecture that can be used on Netronome smartNICs and the BMv2 software switch.

We have also added Tofino support to the code generator, enabling its application in high-speed environments.

During our investigation, we first selected and then reimplemented three existing in-network security methods in P4RROT. We compared the P4RROT-based implementations to the original ones. More concretely, we have realised port knocking functionality similar to P4knocking [18]. Following P4Guard [3], we have created a simple firewall component that patches an NTP (Network Time Protocol) vulnerability [7]. Finally, we have reproduced the results of the in-network passive OS-fingerprinting method called P40f's [1]. We added missing features to P4RROT when they were required.

Our key contributions are as follows: 1) We investigated three in-network security applications, and identified the missing features of P4RROT needed for reproducing them. 2) We extended the P4RROT code generator with additional functionalities implementing the necessary features of security applications. 3) We added support to P4RROT for generating Tofino ASIC-compatible P4 code (TNA model), enabling the development of line-rate security solutions running on fast hardware data plane. 4) We showed that P4RROT has the potential to lower code complexity, and allow the developer to focus on the application logic instead of P4 and hardware specific details, while the generated codes have effectively the same behavior as the original P4 programs. 5) We made the source code added to P4RROT[1] and the evaluation artifacts[2] publicly available.

2 BACKGROUND AND RELATED WORK

Software Defined Networking and P4. To provide computer networks with a high degree of flexibility and scalability Software Defined Networking (SDN) introduced a new way of programming abstractions by decoupling the data and the control plane functionality. While the literature of control plane programmability has a rich past, difficulties of programmable and portable data planes are only recently gaining attention. To offer network developers the desired flexibility, specialized programming languages have been developed. These languages enable experts to describe the entire packet processing pipeline in a protocol-independent manner using high-level abstractions. P4 [2] is one of the most widely supported programming language propositions, with support from both industry and academia. It has numerous compilers for diverse software and hardware targets, ranging from general-purpose processors, FPGAs and SmartNICs, to custom-designed sets of ASICs such as Intel Tofino. The wide range of targets that can run P4 code offers great flexibility in choosing the appropriate device for a wide range of situations. However, it is important to keep in mind that different targets might have different restrictions. For example, in some devices we might find that the use of match-action tables is treated very liberally (in the sense of the number of entries, action complexity, etc.) while the use of registers is more restricted while in other cases we might find the opposite.

In-network security. The emergence of in-network computing offers new opportunities to protect networks more efficiently and effectively, resulting in numerous research works. H. Gondaliya et al. [4] investigate the feasibility of P4-based anti-spoofing mechanisms (ASM) for IP source addresses. M. Mönnich at al. [13] implemented a Router Advertisement Guard (RA-Guard) [10] using P4 to defend IPv6 networks against router spoofing attacks. D. Scholz et al. [16] offer a solution for protecting entire networks from SYN flood attacks using P4 and programmable data planes. K-meleon [12] is an in-network online change detection system that can be used as part of intrusion detection. It identifies heavy-changes instead of changes amongst heavy-hitters only. While the development of these security features in programmable data planes is a well-studied area, implementation usually requires knowledge of the P4 language. Further research can be found in the survey [9].

Code generators for P4. Numerous works leverage P4 by generating P4 source code to speed up the data plane development or to simplify the description of specific use case-dependent behaviour. Graph-To-P4 [19] generates boilerplate code for parser graphs. Lucid [17] is a programming language compiling to Intel Tofino compatible P4 code. It provides event-driven abstractions and a domain-specific type system. V. Nathan et al. [15] propose a network monitoring method involving the compilation of Marple [14] queries to P4 programs. Sonata [5] offers a declarative interface to express queries for a wide range of common telemetry tasks partially running on programmable switches. It also features method chaining and security use cases. Poise [8] compiles BYOD (Bring Your Own Device) policies to P4 switch data plane programs and device configuration files to enforce access control based on dynamic runtime context.

In contrast to these problem specific code generators, P4RROT aims to be a general purpose P4 code generator that can be used to reduce the implementation complexity of any in-network computing tasks including security solutions examined in this paper.

3 P4GUARD-LIKE HOTFIXES

In many cases, network exploits are simple enough so that they can be identified by comparing the values of possibly malicious packets to stored signatures. Creating and deploying an in-network hotfix for an exploit could be essential

[1]Source code: https://github.com/Team-P4RROT/P4RROT
[2]Eval. artifacts: https://github.com/kkaroly42/P4RROT_Security_Apps

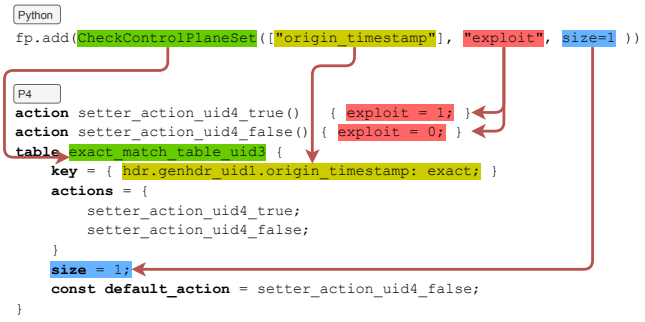

Figure 1: Example usage and P4 output of `CheckControlPlaneSet` command

for a lot of reasons. A trustworthy patch for the vulnerable software might not exist yet or first we might need to make sure that installing it wouldn't create problems on its own in our system. P4Guard is a software-based firewall that can tackle these security challenges. It is protocol-independent and platform-agnostic, making it a versatile solution.

Network Time Protocol (NTP) in its current (at the time of writing this paper) is quite safe against attacks outside the network when it is configured well, but the same thing can't be said about its outdated versions that are still in use on a surprisingly high number of servers.

3.1 Our implementation with P4RROT

As an example we created NTPGuard, a really simple application written with the use of P4RROT, which is meant to block packets carrying malicious payloads aiming to exploit one of NTP's well-known vulnerabilities (detailed in CVE-2015-8138). This application is comparable to the ones created with P4Guard.

To this end we created a `CheckControlPlaneSet` P4RROT command that creates a match-action table that compares the fields of interest to the values stored as table entries and returns a boolean value representing whether the observed packet carries data stored on the switch or not. Note that the permanent addition of this command to P4RROT could also be useful for other use cases since the task of looking up whether a value (or tuple of values) is already stored on the device is quite common in in-network applications. Using match-action tables with exact matching to perform this task is also quotidian, since these use the relatively 'cheap' SRAM and lead to readable and easily extendable code.

4 P4KNOCKING

Port knocking is a well-established security solution that is commonly deployed to prevent attackers from port scanning a server and mapping out its available services. In its simplest form, the user must send one or more UDP or TCP packets to the designated port(s) in order to gain access to the internal network. The "knocking" part can range from sending a single packet to a designated port to a complex sequence of packets sent to static or dynamically calculated ports, even taking into consideration the time gap between consecutive packets.

With the use of P4RROT, our main focus is to create flexible building blocks that allow network administrators to be creative and easily deploy port-knocking solutions that meet their network's requirements. We believe that, based on our work, anyone with a sufficient oversight of the technologies involved can create their own knocking mechanism with minimal effort.

Port knocking among other uses can be a last resort against reverse shells and malware 'talking home'. This, of course, stipulates that the malicious code running in our network was written without exact knowledge of the security measures, or we can create dynamic sequences looking random enough to minimize the likelihood of giving away a complete solution even if a malicious actor can observe the entire network traffic.

A bit more complicated and less-traditional scenario is deploying port-knocking solutions in data centers for accessing network resources. The knocking mechanism can be used to gatekeep access, e.g., to higher-than-average speed links without requiring a central solution, letting ToR or other switches manage this low level of access control.

All of these use cases are different from the traditional use of port knocking in terms of their orientation and placement but the core technical implementation (the knocking mechanism) remains the same.

4.1 Our implementation with P4RROT

Simple implementation. We have created a lightweight version of the 'Main control plane and minimal data plane offloading solution for port 'knocking' presented in [18] for Tofino ASICs. To implement this solution we needed to extend P4RROT with a `Digest` command in order to register the new clients identified by their source IP address. The `CheckControlPlaneSet` command introduced in Section 3 can then be used to decide whether a client has already been granted access or not. It also reinforces our presumption about the importance of this command in in-network applications.

Abstraction for complex knocking tasks. In addition to the simple case, we have also created a more complex replica of P4Knocking, requesting a knocking sequence from users. In the original P4Knocking implementation, registers indexed by the source IP address are used to store counter values indicating how long the user completed the knocking sequence, while we use a state machine-based approach in P4RROT instead.

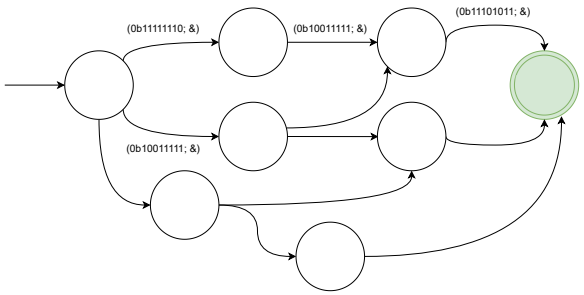

Figure 2: An example for Single Touch State Machine: 255 is the starting state (signified with the arrow without a starting point) and 138 is the single accepting state, signified by the double outline and green color.

In this extended application, the central question was how to represent and update the sequence of states in a way that ensures consistency and allows a large number of sessions to exist simultaneously.

We propose to use a state machine as an abstraction, where transitions are possible actions of the stored state. Each transition consists of a read operation, performs a simple logical operation (and, or, xor) on the bit string, and stores the result. All these steps must be implemented as a single atomic command to ensure consistency.

Note that abstracting the problem into a state machine operating on bit strings could be beneficial for other use cases, perhaps beyond the scope of this paper, since maintaining state while ensuring consistency in networked applications is often a difficult task.

Complex sequence. In this specific task, we define an arbitrary start and accept state represented by bit strings stored in registers. The transitions between the states are defined as a mapping between the destination ports of the incoming TCP packets and the bit string, the user combines (the exact action used in the transition is predefined, but can be easily changed at the code level) with the current state. Mapping between target port and bit string is done with a new `ReadFromControlPlaneSet` P4RROT command, which targets the common problem of mapping values to each other. To do so it generates a match-action table using exact matching (in this particular case, otherwise the command is capable of handling ternary matching as well) in which the pairs are stored as entries, this way making the mapping completely modifiable from the control plane.

To ensure state consistency, we needed to make sure that reading the current state and calculating the next state is done in a single atomic step. For this purpose, we created the P4RROT commands `XORSharedArray`, `ANDSharedArray` and `ORSharedArray`, which create register operations that can perform a logical exclusive or, logical and, and logical or operations respectively on the bit string stored in the register, and the bit string requested as part of the transition. The register actions then store it in the register and return the result.

Fig. 2 depicts a special Single Touch State Machine where we require each transition to be available from every state and all states must have a path leading to the accepting state or states. For a P4 application, such a state machine can be realized with the use of registers storing the current state and register actions carrying out the state transition safely.

5 OS FINGERPRINTING AND P40F

Operating System fingerprinting can be an extremely useful tool for network administrators. The knowledge on the precise distribution of applied operating systems in a network can enables network administrators to deploy complex network policies and gain deeper insight into problems that may lead to user complaints (e.g., some software that is part of an OS might conflict with a service causing some hard to trace-back behaviour).

P0f monitors the network's TCP/IP header information and assigns a prediction on the operating system to each stream according to the stored signatures. The method is based on pure passive network measurements, without sending probe packets and without consuming network bandwidth.

P40f presented in [1] is a P4 implementation of P0f. P40f fingerprints the traffic passing through the P4 data plane by examining TCP/IP headers, mostly relying on TCP options. The design allows specific and generic rules to exist as signatures, while still providing a degree of flexibility in configuration. This flexibility is enhanced by the idea of fuzzy matching, which means that if some key fields match, P40f (and P0f too) can loosen the examination on other fields.

P40f randomly selects a number of HTTP GET packets that the control plane analyzes. The reason for this is the fact that these are too complex to be processed on hardware data planes (e.g., Tofino ASIC).

5.1 Our implementation with P4RROT

To implement P40f, we only had to extend P4RROT slightly. First, we generalized P4RROT's Bloom filter construct to handle tuples, as required to recognize HTTP GET request-response sessions. Mapping the operating system label to the TCP/IP header signatures is done by the `ReadFromControlPlaneSet` command described in Section 3. Finally, we created a binary `LeftShift` and a binary `RightShift` command to implement the binary search used to determine the size of the TCP window. These are necessary because there is no built-in operation for division in P4.

Observed code	P4Knock	P4Guard	P40f
P4RROT code	35	38	187
Generated P4	93	56	356
Complete generated	325	283	768
Original implementation	321	-	884

Table 1: Comparison of source code lengths

6 EVALUATION

First, we evaluate the effectiveness of P4RROT as a code generator along various metrics. We then measure the accuracy of the OS fingerprinting and compare it to the original solution. Finally, we examine the functional correctness of NTPGuard and P4Knocking implementations. Note that all the source code used in our study and additional evaluation artifacts are publicly available (see the links in Section 1).

6.1 Code generation

Table 1 compares the length of source code files for different solutions of the investigated use cases. The rows of the table include: 1) P4RROT code: the Python code using P4RROT; 2) Generated P4: the P4 code generated directly by P4RROT; 3) Complete generated: The P4 code generated by P4RROT and the template created by hand; 4) Original implementation: The code published by the original authors of the reproduced application; Blank lines are excluded from the count, while comments are included.

We observed that in cases where a comparison can be made, both the Python code generating the solution and the generated P4 files are shorter than the original implementation or approximately the same length. Naturally, we don't want to claim that code length is a good measure of any quality of an implementation. We believe this comparison shows that using the highly reusable template provided in P4RROT and a well-thought-out design in mind, it is faster to write an order of magnitude less Python code than to write P4 code directly.

Of course, there may be cases where the exact operation or construct required to solve a particular problem is not available in P4RROT and would be too much work to implement. Even in these cases we can presumably produce P4 code base quickly, which can be modified manually. We believe this mindset aligns with P4RROT's goal of being a support tool rather than a replacement for P4 programming.

Table 2 summarises the resource usage of the applications implemented by us using the Tofino Native Architecture. We used SDE 9.4.0 to compile the P4 code.

We observed that using P4RROT to create our implementations didn't come at an unreasonably high cost, as we used well under 50% of all types of resources, leaving space for potential further functions. Unfortunately, we were unable to make a direct comparison with the original implementations due to several reasons. For P4Guard we couldn't find a publicly available code base and the other applications weren't implemented with the Tofino Native Architecture. While P40f does have an implementation for Tofino, we used the original version designed for the BMv2 software switch as a basis for our work.

Resource	P4Knock	Extended P4Knock	NTPGuard
Stages	25%	25%	16.7%
SRAM	1.3%	2.0%	0.4%
TCAM	0%	0%	0%
ALU	0%	2.1%	0%

Table 2: Used resources on Tofino

6.2 Accuracy of OS fingerprinting

To test our p40f implementation we compiled our solution and the original one with the p4c compiler and ran them on the BMv2 software switch. Since the signatures and packets used in the original P40f paper weren't at our disposal we couldn't exactly reproduce the evaluation steps described there, but we did compare the results given by the original and our implementation on part of the CIC-IDS2017 dataset. In this comparison the labels generated by the two implementations were exactly the same. Based on this, we claim that we managed to reproduce P40f for BMv2 with the use of P4RROT quite adequately.

6.3 Functional evaluation of NTPGuard and P4Knocking

To test whether our implementation of NTPGuard and P4Knocking really perform the tasks they are supposed to we sent some sample traffic through them. In both cases, we monitored the packets entering and exiting the Tofino model (ASIC emulator). We made the resulting files and scripts publicly available in order to make our claims easily verifiable (see the link in Section 1).

To evaluate the efficiency of our P4Knocking implementations we conducted a test by attempting to establish an SSH connection through the model. The test consisted of two attempts: one in which the required port knocking sequence was not performed, and another in which the sequence was performed. The results demonstrated the correctness of our implementation, as the first attempt failed while the second attempt was successful.

In the case of NTPGuard we sent structurally correct but randomly generated NTP packets into the model. Roughly

half of the packets constituted an exploit. The results indicated the correct functioning of the application, as all packets constituting an exploit were successfully filtered.

7 DISCUSSION

We found P4RROT well-suited to support network-security use cases, and we successfully reproduced the solutions of three prior works. While admitting that comparing number of lines is not a good measurement of quality, we believe that our results (summarized in Table 1) strengthen the point that using P4RROT's highly reusable commands can speed up the implementation of P4 applications.

Its higher abstraction-level enables to quickly experiment with new ideas like the Single Touch State Machine (STSM) construction described in Section 4. With the use of P4RROT we were able to build it quickly into our solution. Further exploration of STSMs seems very promising, as they provide a general solution to the problem of keeping track of state while maintaining consistency. In the future we would like to evaluate the prospect of creating disaggregated STSMs, employing STSMs in other use cases and the benefits of further mathematical constraints on the abstract model.

Although P4RROT's design is not created for non-whole byte length special fields and flags, it is able to accommodate them. During our work, we contributed multiple missing features and support for the Tofino Native Architecture (TNA).

In this work, we focused on security use cases, and it will probably remain a constantly changing area requiring fast patches and quick adaptation providing ground for numerous similar projects. However, P4RROT is not exclusively focused on security but instead aims to facilitate high-level application layer logic and general computing. One of the other potential areas is natural science projects dealing with high-throughput network traffic.

ACKNOWLEDGMENT

Cs. Györgyi thanks the support of project Strengthening the EIT Digital KIC in Hungary (2021-1.2.1-EIT-KIC-2021-00006), implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the 2021-1.2.1-EIT-KIC funding scheme. S. Laki thanks the support of National Research, Development and Innovation Office - NKFIH, FK_21 138949.

REFERENCES

[1] Sherry Bai et al. 2022. Passive OS Fingerprinting on Commodity Switches. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 264–268. https://doi.org/10.1109/NetSoft54395.2022.9844109

[2] Pat Bosshart et al. 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] Rakesh Datta et al. 2018. P4Guard: Designing P4 Based Firewall. In *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*. 1–6. https://doi.org/10.1109/MILCOM.2018.8599726

[4] Harsh Gondaliya et al. 2020. Comparative Evaluation of IP Address Anti-Spoofing Mechanisms Using a P4/NetFPGA-Based Switch. In *Proceedings of the 3rd P4 Workshop in Europe (EuroP4'20)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3426744.3431320

[5] Arpit Gupta et al. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.

[6] Csaba Györgyi et al. 2023. P4RROT: Generating P4 Code for the Application Layer. *SIGCOMM Comput. Commun. Rev.* 53, 1 (apr 2023), 30–37. https://doi.org/10.1145/3594255.3594258

[7] Philipp Jeitner et al. 2020. The Impact of DNS Insecurity on Time. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 266–277. https://doi.org/10.1109/DSN48063.2020.00043

[8] Qiao Kang et al. 2020. Programmable {In-Network} Security for Context-aware {BYOD} Policies. In *29th USENIX Security Symposium (USENIX Security 20)*. 595–612.

[9] Elie F. Kfoury et al. 2021. An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *CoRR* abs/2102.00643 (2021). arXiv:2102.00643 https://arxiv.org/abs/2102.00643

[10] E Levy-Abegnoli et al. 2011. RFC 6105: IPv6 Router Advertisement Guard. *Interet Engineering Task Force, Request for Comments* (2011).

[11] IBM LLC. 2022. How much does a data breach cost in 2022. (2022). https://www.ibm.com/security/data-breach

[12] Gonçalo Matos et al. 2021. Generic Change Detection (Almost Entirely) in the Dataplane. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS '21)*. Association for Computing Machinery, New York, NY, USA, 113–120. https://doi.org/10.1145/3493425.3502767

[13] Moritz Mönnich et al. 2021. Mitigation of IPv6 Router Spoofing Attacks with P4. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. 144–150.

[14] Srinivas Narayana et al. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3098822.3098829

[15] Vikram Nathan et al. 2017. Demonstration of the marple system for network performance monitoring. In *Proceedings of the SIGCOMM Posters and Demos*. 57–59.

[16] Dominik Scholz et al. 2020. SYN Flood Defense in Programmable Data Planes. In *Proceedings of the 3rd P4 Workshop in Europe (EuroP4'20)*. Association for Computing Machinery, New York, NY, USA, 13–20. https://doi.org/10.1145/3426744.3431323

[17] John Sonchack et al. 2021. Lucid: a language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 731–747.

[18] Eder Ollora Zaballa et al. 2020. P4Knocking: Offloading host-based firewall functionalities to the network. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 7–12. https://doi.org/10.1109/ICIN48450.2020.9059298

[19] Eder Ollora Zaballa and Zifan Zhou. 2019. 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, 1–2.