Technische Universität München
Fakultät für Informatik

Performance Prediction of Programmable Data Planes through Control Flow Modeling

Dominik Scholz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Die Dissertation wurde am 27.09.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 04.02.2022 angenommen.

Acknowledgments

This thesis would not have been possible without the support I have received from many individual people surrounding me.

First of all, I want to thank Prof. Dr.-Ing. Georg Carle for offering me the chance to join the Chair for Network Architectures and Services and for guiding and supervising my dissertation. Throughout the past years you have supported me and provided me with invaluable ideas and critical feedback. I would also like to extend my gratitude to Ass.-Prof. Dr. Sándor Laki for taking your time and offering to be my second assessor and Prof. Dr. Florian Matthes for chairing the examination committee.

I want to thank my fellow colleagues and co-authors that I had the pleasure to spend time with in and out of work over the past years:Henning, Stephan, Johannes, Marcel, Daniel, Paul, Manuel, Benedikt, Max, Alexander, Holger, Fabien, Andreas, Krzysztof, Hendrik, Thomas, Andreas, and Wolfgang. I am particularly greatful for everyone that I had the pleasure to work and teach with in the iLab1 team. Thank you Lars, Minoo, Florian, Lukas, Marc-Oliver, Stefan, and Florian for great times and an uncountable amount of oral attestations. I am also grateful for my longest-serving office colleague, Sebastian. Thank you for long evenings developing different testbeds and bouncing ideas off of each other, night-shifts of doing testbed maintenance and cable management in the router lab, and proof-reading many of my texts, including the following *p*ile *o*f *s*heets.

I specifically want to mention the people that are responsible for me ending up in computer science, my math and informatics teachers in gymnasium. Thank you Hermann and Peter. I would never have ended up where I am today without your steady support and letting me program “advanced” little programs during your school lessons. The bread never stopped walking up the stairs. This joke really does not translate well into English.

Lastly, I want to thank my wonderful parents Angela and Heiko. None of this would have been possible without your support, whenever and for whatever I needed it. You let me explore this path, for which I will be forever grateful. Danke für alles.

Abstract

Entirely programmable network devices are a dream-come-true for network operators. Finally, they can have complete control over packets within their networks, independent of the functionality provided by switch vendors. Network operators can use custom protocols for specialized processing or gather detailed telemetry data from within data planes, significantly improving efficiency within their network. Thereby, intellectual property is kept internal, instead of sharing it with switching fabric and operating system designers. Vice-versa, open-source solutions, with all their benefits and drawbacks, can be used. This newfound freedom in network data planes also sparks and revitalizes a plethora of new and old research domains. In this thesis, we want to tackle three different domains.

First, we propose a novel methodology for reproducible experimental evaluation of programmable data plane targets. All experiments presented in this thesis focus on producing reproducible results. We reduce the network setup to the bare minimum of two nodes, a load generator, and the device under test. Using a high degree of automation and gathering all experiment artifacts allows to run measurements and publish their results efficiently. We achieve this through our purpose-built testbed orchestrating software, which enforces reproducibility by design. Although we only apply this approach to analyzing data planes in this work, we designed our testbed setup and management software to enable generic network experiments.

Second, due to the increasing landscape of data plane applications and data plane targets that can be programmed using domain-specific languages, modeling and predicting the performance for each possible combination becomes infeasible. However, this ability is essential to understand how and why an application behaves in a certain way on a data plane target. Network operators may use this information to determine bottlenecks that might arise before buying hardware, indicating a necessity to switch to another network device. To combat this complexity, we instead opt to model components of data plane processing pipelines individually. Based on our experiment methodology and testbed environment, we designed a framework for the automated evaluation and modeling of parameter changes for data plane program components. As control-flow graphs can be derived from such programs, as a second step, we determine the cost of each path through the data plane. We argue to focus modeling efforts on key performance impacting components. Based on the worst-case path, i.e., the path with the highest cost, we derive traffic that hits this path, resulting in a worst-case evaluation of the program.

Lastly, we analyze how cryptographic hash functions can be integrated into programmable data planes. Using our evaluation framework, we evaluate the performance of such a component for various software and hardware targets. We argue that cryptographic hashes are an essential component for many applications that could be moved to the data plane. This includes low-level protocols that require built-in data integrity or the creation of cryptographic challenges for the mitigation of attacks. We use our prototypes to successfully implement SYN flood mitigation approaches for a selection of programmable data planes. Furthermore, we implement the same strategies using a traditional framework for high-performance packet processing. Based on implementation effort and performance figures, we discuss the advantages and disadvantages of both approaches.

Zusammenfassung

Vollständig programmierbare Netzwerkgeräte sind für die Betreiber von Netzwerken ein wahr gewordener Traum. Die Betreiber haben die vollständige Kontrolle über Pakete in ihren Netzwerken, unabhängig von der Funktionalität, die von Switch-Anbietern bereitgestellt wird. Netzbetreiber können benutzerdefinierte Protokolle für spezielle Verarbeitungen verwenden oder detaillierte Telemetriedaten innerhalb der Data-Planes sammeln, wodurch die Effizienz in ihrem Netzwerk erheblich verbessert wird. Zusätzlich wird geistiges Eigentum geheim gehalten, anstatt es mit den Entwicklern von Switch-Hardware und Betriebssytemen zu teilen. Umgekehrt können Open-Source-Lösungen mit all ihren Vor- und Nachteilen eingesetzt werden. Diese neu gewonnene Freiheit in Netzwerk-Data-Planes führt auch zu einer Vielzahl neuer und alter Forschungsbereiche und belebt diese neu. In dieser Arbeit wollen wir uns mit drei verschiedenen Bereichen befassen.

Zunächst schlagen wir eine neuartige Methodik für die reproduzierbare experimentelle Bewertung programmierbarer Data-Planes vor. Alle in dieser Arbeit vorgestellten Experimente konzentrieren sich auf die Erzeugung reproduzierbarer Ergebnisse. Wir reduzieren den Netzwerkaufbau auf ein Minimalsetup bestehend aus zwei Geräten, einem Lastgenerator und einem Device-under-Test. Durch einen hohen Automatisierungsgrad und die Aggregation aller Experimentartefakte können wir Messungen effizient durchführen und ihre Ergebnisse veröffentlichen. Wir erreichen dies durch unsere eigens entwickelte Testbed-Orchestrierungssoftware, die die Reproduzierbarkeit gewährleistet. Obwohl wir diesen Ansatz in dieser Arbeit nur auf die Analyse von Data-Planes anwenden, haben wir unsere Testbed-Setup- und Management-Software so konzipiert, dass sie auch generische Netzwerkexperimente ermöglicht.

Zweitens ist es aufgrund der wachsenden Anzahl von Data-Plane Anwendungen und Plattformen, die mittels domänenspezifischer Sprachen programmiert werden, nicht mehr möglich, die Leistung für jede mögliche Kombination zu modellieren und vorherzusagen. Diese Fähigkeit ist jedoch unerlässlich, um zu verstehen, wie und warum sich eine Anwendung auf einer Zielplattform auf eine bestimmte Weise verhält. Netzbetreiber können diese Informationen nutzen, um mögliche Engpässe bereits vor dem Kauf von Hardware zu ermitteln. Entsprechend kann dann ein passenderes Gerät gekauft werden. Um dieser Komplexität entgegenzuwirken, modellieren wir stattdessen die Komponenten der Data-Plane Verarbeitungspipelines einzeln. Auf der Grundlage unserer Experimentmethodik und Testbed-Umgebung haben wir ein Framework für die automatisierte Auswertung und Modellierung von Parameteränderungen für Programmkomponenten der Data-Plane entwickelt. Auf Grundlage der Kontrollflussgraphen dieser Programme, bestimmen wir in einem zweiten Schritt die Kosten für jeden Pfad durch die Data-Plane. Wir plädieren dafür, sich bei der Modellierung auf die wichtigsten Komponenten zu konzentrieren, die sich auf die Leistung auswirken. Anhand des Worst-Case-Pfads, d.h. des Pfads mit den höchsten Kosten, leiten wir Netzwerkpakete ab, die durch diesen Pfad verarbeitet werden, was zu einer Worst-Case-Bewertung des Programms führt.

Schließlich analysieren wir, wie kryptografische Hash-Funktionen in programmierbare Data-Planes integriert werden können. Unter Verwendung unseres Evaluationsframeworks bewerten wir die Leistung einer solchen Komponente für verschiedene Soft- und Hardwareziele. Wir ar-

gumentieren, dass kryptografische Hashes eine wesentliche Komponente für viele Anwendungen sind, die in die Data-Plane verlagert werden könnten. Dazu gehören Low-Level-Protokolle, die Datenintegrität oder die Anwendung kryptografischer Verfahren zur Abwehr von Angriffen erfordern. Wir verwenden unsere Prototypen, um erfolgreich SYN-Flood-Abwehrmechanismen für eine Auswahl von programmierbaren Data-Planes zu implementieren. Darüber hinaus implementieren wir dieselben Strategien unter Verwendung eines traditionellen Frameworks für die Hochleistungs-Paketverarbeitung. Anhand des Implementierungsaufwands und der Leistungsdaten diskutieren wir die Vor- und Nachteile beider Ansätze.

Contents

1 Introduction **1**
1.1 Research Questions . . . 2
1.2 Outline . . . 6

2 Programmable Network Devices **9**
2.1 Historic & Related Approaches . . . 9
2.1.1 Software Devices . . . 9
2.1.2 Programmable Hardware . . . 15
2.1.3 OpenFlow . . . 15
2.2 Domain-specific Languages for Programming Data Planes . . . 16
2.2.1 Extended Berkeley Packet Filter . . . 17
2.2.2 Programming Protocol-independent Packet Processors . . . 17
2.2.3 Future Trends . . . 20
2.3 Target Platforms . . . 20
2.3.1 Software on CPU . . . 20
2.3.2 Hardware . . . 21
2.3.3 Other Targets . . . 22
2.3.4 Comparison . . . 23
2.4 Data Plane Programming Taxonomy . . . 24
2.4.1 P4 Application and Research Domains . . . 25
2.4.2 Discussion . . . 25
2.5 Key Results . . . 26
2.6 Statement on Author's Contributions . . . 27

3 Measurement Methodology **29**
3.1 Terminology and Experiment Metrics . . . 29
3.2 Orchestrating Reproducible Network Device Measurements . . . 31
3.2.1 Types of Testbeds and Testbed Controllers . . . 31
3.2.2 Testbed Life Cycles . . . 32
3.2.3 Requirements Analysis for Testbed Controller . . . 33
3.2.4 Plain Orchestrating Service . . . 34
3.2.5 Case Study: Orchestrating a Combined Teaching and Research Testbed . 43
3.2.6 Conclusion . . . 53

3.3 MoonGen as Dynamic Load Generator 53
3.3.1 Motivation 53
3.3.2 Survey of Protocol Stacks 54
3.3.3 libmoon Framework 57
3.3.4 libmoon's Dynamic Protocol Stack 58
3.3.5 Performance Evaluation 64
3.3.6 Conclusion 67
3.4 Key Results 68
3.5 Statement on Author's Contributions 69

4 Performance Modeling of Programmable Network Devices 71
4.1 Modeling Approach 72
4.1.1 Components of P4 Data Planes 72
4.1.2 Deriving Parameterized Component Models 73
4.1.3 Path-specific Performance Model 74
4.1.4 Per-packet Performance 75
4.1.5 Model-first Approach 76
4.2 Related Work 76
4.2.1 P4 Control Flow Graphs 76
4.2.2 Performance Analysis & Modeling of P4 Devices 77
4.3 Measurement Framework for Automated Component Modeling 82
4.3.1 Component Experiment Specification 82
4.3.2 Measurement Execution 83
4.3.3 Artifact Evaluation 84
4.4 Modeling a Software Data Plane Target 90
4.4.1 Framework Integration 91
4.4.2 Resource Utilization 91
4.4.3 Baseline Model 91
4.4.4 Component Models 94
4.4.5 Modeling of Program Paths 107
4.5 Model Limitations 109
4.6 Key Results 110
4.7 Statement on Author's Contributions 112

5 Cryptographic Hashing in Programmable Data Planes 115
5.1 Motivation 116
5.1.1 The Need for Cryptographic Hashing 116
5.1.2 Usage in Networking Protocols 117
5.2 Feasibility and Integration of Hashing with Cryptographic Properties 118
5.2.1 Related Work 118
5.2.2 Interface for Hashing in P4 120
5.2.3 Choice of Hash Function 121
5.2.4 Integration Strategies 125

5.2.5 Prototype Implementations . . . 127
5.2.6 Evaluation . . . 128
5.2.7 Limitations . . . 133
5.2.8 Conclusion . . . 133
5.3 Case Study: SYN Flood Defense in Programmable Data Planes . . . 135
5.3.1 SYN Flood Mitigation . . . 135
5.3.2 Deployment Scenario . . . 142
5.3.3 Related Work . . . 146
5.3.4 Requirements for SYN Proxy Implementation . . . 148
5.3.5 Software Packet Processing Prototype using libmoon . . . 149
5.3.6 Programmable Data Plane Prototype in P4 . . . 152
5.3.7 Evaluation . . . 154
5.3.8 Conclusion . . . 161
5.4 Key Results . . . 162
5.5 Statement on Author's Contributions . . . 163

6 Conclusion **165**
6.1 Contributions . . . 165
6.2 Future Work . . . 168

A Appendix **171**
A.1 List of Acronyms . . . 171
A.2 List of Figures . . . 175
A.3 List of Tables . . . 177

Bibliography **179**

Chapter 1

Introduction

Researchers and network operators have been chasing the dream of fully programmable networks and network operating systems (NOSes) over the past decades. During this endeavor, step by step, they have gotten closer and closer to achieving these goals. OpenFlow was a first step that implemented software-defined networking (SDN) concepts, allowing to make decisions in the centralized control plane, which can then be propagated to individual data planes. However, the number of possible fields that can be matched in the data plane on a packet is limited to selected fields of existing protocols. Similarly, possible actions, how the packet should be processed, cannot be programmed freely. The complexity and capabilities of networks and the requirements for networked applications are steadily increasing. This added complexity also raises the requirements for achieving a programmable network. Thus, fully programmable data planes are the next step in this evolution. With the advent of languages like Programming Protocol-independent Packet Processors (P4)[1] [23], the behavior of the network device's data plane can be—almost—freely defined using higher-layer, abstract programming languages. Instead of only allowing a limited set of actions for a limited set of fixed bytes that can be matched in a packet, P4 allows for arbitrary operations triggered by matching any sequence of bits in the packet—independent of existing or future protocols. Such new paradigms for programming high-performance packet processing platforms enable a shift of network applications, previously located in the control plane or an end host, to the data plane—located anywhere in the network.

The first aspect discussed in this thesis considers the ever-increasing number of programmable network targets and their performance characteristics. Several broad categories of different technologies as the basis for such devices exist; containers, using namespaces or cgroups; virtualization based on Xen or KVM; high-performance software packet processing platforms like the Data Plane Development Kit (DPDK), netmap, or Snabb; and, finally, actual hardware devices based on smart network interface cards (NICs), field-programmable gate arrays (FPGAs),

[1] Not to be confused with the P4 (Programmable Protocol Processing Pipeline) platform for protocol boosters in FPGAs proposed by Hadzic et al. in 1997 [22]

or application-specific integrated circuits (ASICs). In addition, several features, like multi- or many-core CPUs or multi-queue NICs, further increase the complexity of the packet processing landscape. It is clear that this list of possible targets, architectures, and approaches will only get longer in the future. Combining this aspect with the shift of entirely new applications to such programmable devices poses the challenge of understanding the performance of a particular program on a specific target device. In other words, which aspects of a program, e.g., an longest prefix matching (LPM) table lookup, cause performance degradations on a target device, for instance, the used algorithm. To improve the understanding, models are required that predict the performance of packets through data plane programs. In particular, the model for the actual program behavior on a given target can be compared to the expected theoretical performance. We present a modeling framework that uses reproducible measurements to model the behavior of individual P4 program components. We then combine these models to provide performance estimates for the processing of the worst-case packet, i.e., a packet that takes the path with the highest predicted cost through the plane.

The possibility to move network applications to data planes anywhere in the network also sparks new research questions: which applications *could* and, more importantly, which applications *should* be moved to the data plane in the first place? While the current set of P4 instructions allows for manifold network applications in the data plane, additional functionality might be desired. For instance, cryptographic functions for encryption or hashing, required in many widely used network protocols and applications, are currently not part of the core language specification. Therefore, this functionality might only be available through external, target-dependent application programming interfaces (APIs). However, providing this set of cryptographic functions would enable the use of P4 in entirely new domains like industrial networks. In this thesis, we discuss how cryptographic hashing can be integrated into programmable data planes. We implement different approaches for a selection of target devices and discuss limitations. Whether such an integration is worth the effort in terms of resource requirements, throughput, latency, and other performance criteria is analyzed using a case study: moving TCP SYN flood defense to the data plane by implementing several promising strategies as prototypes. Further, we compare the resulting implementations with a classic approach using a high-performance packet processing framework.

1.1 Research Questions

We want to answer the following research questions in the context of this thesis:

- RQ1: How can we perform reproducible measurements for a heterogeneous set of software and hardware data planes?
- RQ2: How can we operate multi-user testbeds for reproducible experiments using heterogeneous hardware?
- RQ3: How can we model the performance of individual data plane components in an automated fashion?

- RQ4: How can we determine the packet that takes the worst case path through the data plane?
- RQ5: How can we extend programmable data planes with cryptographic hashing functionality?
- RQ6: How can we defend against TCP SYN floods from within the data plane?

The context and content covered by each research question are further detailed in the following. Each research question focuses on a networking domain, for which we want to improve the state-of-the-art through our contributions.

RQ1: How can we perform reproducible measurements for a heterogeneous set of software and hardware data planes?

Producing reproducible network experiments is essential to present and publish results that others can verify and understand. Typically, three stages are distinguished: repeating the same experiment using the same setup; reproducing the experiments of others using the same setup; and, finally, replicating experiment results of others using a different setup. To obtain the highest degree of reproducibility—replicability,— several aspects, like documenting and collecting the entire configuration and setup, and automation to reduce human error, are vital.

To achieve reproducibility in this thesis, we reduce our measurement setup to the bare minimum whenever possible: a load generating node, directly connected to the device under test (DuT). As we are only interested in the behavior of the DuT, this approach reduces, for instance, external influences caused by other network devices like switches or middleboxes on the path. Furthermore, independent individuals can easily set up this simple setup, enabling them to reproduce the experiment.

As a second step, we specifically designed our experiments and the management software for our testbed, coined the plain orchestrating service (pos), via which experiments are executed, to not only enable but enforce reproducibility. This includes a high degree of automation and the central collection of all experiment artifacts, for instance, configuration and setup of experiment nodes, executed scripts, and all generated results. Typically, the same measurement is repeated several times for the same DuT, only varying one parameter to analyze its impact on the performance. To reflect this, experiments via pos are executed using so-called *loops*. As input, the testbed controller expects all parameters of the experiment that may vary and the respective parameter space. The measurement is then automatically executed for every possible combination of these parameters, drastically simplifying the process and, therefore, reducing potential causes for human error, which directly contradict reproducibility.

We define terms and important performance metrics used throughout this thesis in Chapter 3. Then, we introduce our methodology to achieve reproducible network experiments using our testbed orchestrating service. Thereby, the enforced structure of experiments and usage of pos are key components.

RQ2: How can we operate multi-user testbeds for reproducible experiments using heterogeneous hardware?

The presented testbed and pos are not solely implemented for the reproducible measurements discussed in this thesis. Instead, we designed pos to allow multiple users with different backgrounds, like students and researchers, for a wide range of network experiments. While this includes the two-node setup used throughout this thesis, the testbed may also be used, for instance, for pure computation or large-scale distributed experiments with several dozen connected nodes. This changes the requirements towards the testbed and respective management software: testbed nodes need to be efficiently shared between users; multiple users may simultaneously use different nodes; and testbed nodes themselves are based on a wide variety of heterogeneous hardware, providing different management APIs.

In Chapter 3, we present how our testbed management software pos helps to reduce and streamline administrative efforts and allows efficient usage of testbed resources. We discuss requirements and present its core components. In a case study, we highlight how our testbed controller can be used to manage a variety of testbeds for different purposes, including a hybrid teaching and research testbed consisting of 48 nodes for distributed experiments.

RQ3: How can we model the performance of individual data plane components in an automated fashion?

Domain-specific languages (DSLs) like P4 allow programming the packet processing behavior of the entire data plane. This processing pipeline can be split into individual components, including parser, match-action tables, or deparser stages. Each component has different parameters that describe their complexity, e.g., the number of parser states. This clear modularization of processing allows investigating individual features of each component, in theory.

Understanding the performance of a P4 program on a given target platform is crucial to know whether the target fulfills the requirements towards the expected performance. However, the landscape of both P4 applications and target platforms is steadily increasing, such that modeling the performance of every complete application for every target becomes infeasible. Tehrefore, modeling the performance of individual components of the data plane processing pipeline can help to predict the performance of programs using the modeled components. This approach reduces the overall effort for performance measurements, limiting experiments to components of different targets.

As the search space is still large we present a framework for automated measurements and modeling of P4 program components to achieve reproducible performance models. The auto-generated models can be used to compare the behavior of a component to the theoretically expected performance. Potential deviations can be further investigated to find the root cause, e.g., inefficient or incorrectly implemented data structures or limitations of the target's architecture.

Chapter 4 details our framework for the automated generation of models for P4 data plane components. It is based on the pos structure for experiments, wherefore, all measurements and created models are reproducible. The framework is specifically designed to be data plane target-independent, and testbed-independent, further increasing the reproducibility. We present a case study for a software and a hardware target, inspecting the core component of P4-based data planes: match-action tables.

RQ4: How can we determine the packet that takes the worst case path through the data plane?

Using a DSL to program the data plane of network devices allows using tools for the analysis of control-flow graphs (CFGs). In other words, based on the program, the path of a packet through the data plane can be determined. This opens up new possibilities to verify correct program behavior but also to predict the performance of the program for different packets. Using the CFG, i.e., which processing component exists how many times for a certain path, in combination with the performance models for individual program components, results in the total cost for each path.

Not all components will significantly impact the overall program performance similarly. We assume that key components will dominate, for instance, match-action tables, while other actions, like simple calculations, will only marginally impact the performance. Therefore, modeling the entire path can be reduced to modeling the key components while using fixed offsets for other, minor operations.

After modeling all paths, the worst-case path through the data plane program can be determined and, with it, the packet that uses this path. Therefore, generating this specific sort of traffic triggers the worst-case behavior of the program on the target device. A similar approach is possible for the average case.

In Chapter 4 we propose our methodology to determine such worst-case traffic for P4 programs. We discuss different tools to generate CFGs and how this information can be combined with our presented models for individual data plane components. After parameterizing the cost of each path we deduce the worst-case traffic. This is then used to verify the quality of our model.

RQ5: How can we extend programmable data planes with cryptographic hashing functionality?

Cryptographic functionality is detrimental for modern networks. Whether these means are used directly between network nodes or on application-level between two or more communicating parties, with their usage, communication channels can be authenticated, confidential, and integral. Although cryptographic properties are included in many essential protocols, e.g., for tunneling applications, cryptographic functionality is not yet available in the core of DSLs like P4. Instead, such functions can only be used through external libraries or interfaces provided by the concrete target. However, programmable hardware targets often do not provide cryptographic functionality or require complex integration efforts.

Given all different classes of cryptographic functions, we have a closer look at cryptographic hash functions. These hashes are primarily used to provide authentication and data integrity in network protocols. Furthermore, specialized use cases exist, e.g., client puzzles to protect against certain network attacks.

Chapter 5 outlines how cryptographic hash functions can be integrated into P4-programmable data planes. We showcase prototype implementations for a variety of software and hardware targets, using different integration approaches. Further, we discuss limitations encountered due to hardware-specifics and quantify the achievable performance regarding throughput, latency, and resources using measurements. We argue that the importance of cryptographic functionality for network devices and applications merits a more thorough discussion about integrating them closer into programmable data plane targets.

RQ6: How can we defend against TCP SYN floods from within the data plane?

Distributed denial-of-service (DDoS) attacks are a threat to the Internet and all its stakeholders, including content providers and network operators. If successful, i.e., attack traffic is not mitigated, such attacks can cause financial losses by overwhelming network devices. With ever-increasing bandwidths, DDoS attacks will likely become even more devastating in the future. Consequently, the cat and mouse game, finding solutions capable of defending against such attacks, will—and has to—continue.

One popular attack vector among more significant denial-of-service (DoS) attacks are SYN floods, exploiting a flaw in the TCP handshake that causes asymmetric usage of resources for the attacked server compared to the attacking client(s). Typically, every network stack of modern operating systems provides countermeasures, however, with limited performance. Commercial closed-source solutions can be deployed within the network instead of mitigating the flood at the end host. While a plethora of potential defense approaches has been proposed, solutions that cannot be circumvented by an intelligent attacker typically require a client puzzle, which uses a cryptographic hash function for its calculation. We use this popular attack vector to demonstrate how cryptographic hash functions can be used to mitigate SYN floods in programmable data plane targets anywhere in the network.

Chapter 5 includes a critical discussion of general SYN flood mitigation mechanisms and in which setups they can be deployed. We investigate state-of-the-art implementations using the Linux TCP/IP stack as a case study. Based on our findings, we narrow down potential candidates for porting them to the data plane. We implement and compare prototypes using two approaches: using a traditional framework for high-performance packet processing in software; and using the P4 language to create programs for a variety of software and hardware targets. We use the same underlying framework for both approaches using software targets, allowing a detailed comparison of implementation effort, limitations, and performance.

1.2 Outline

Chapter 2 covers the history and road towards programmable network devices and compares general properties of such target devices and data plane applications. We introduce our measurement methodology based on our testbed and tools for reproducible network experiments in Chapter 3. Our approach for automatically modeling the components of P4 language constructs on different target devices is presented in Chapter 4. Furthermore, this chapter discusses the application and interesting findings of our modeling framework to a software-based target platform. In Chapter 5, different strategies to integrate cryptographic hashing capabilities into programmable data planes are presented. These approaches are used to implement multiple high-performance TCP SYN flood defense prototypes directly in the data plane. The contributions of this thesis and open topics for future work, are summarized in Chapter 6.

Chapter 2

Programmable Network Devices

The landscape of packet processing applications has always been broad. They range from software devices, for instance, allowing to implement custom networking capabilities on top of operating system (OS) network stacks or dedicated frameworks, to purpose-built hardware devices. How to define the networking behavior, is a similarly complex domain: standard programming languages, switch OSes, or limited configuration options for fixed-function ASICs, are only a few prominent possibilities. Therefore, the urge to develop a DSL, a high-level language to define data plane packet processing, has obvious advantages.

This chapter introduces important terms, concepts, DSLs, and investigated software and hardware platforms. Furthermore, we present a survey of the current landscape of research and applications for programmable data planes.

2.1 Historic & Related Approaches

Programming custom network functions instead of buying fixed-function, ASIC-based appliances is possible for both software- and hardware-based platforms. These have been developed and improved over time to achieve higher throughputs, lower latency, and allow for more complex functionality. In this section, we focus on and discuss properties, advantages, and disadvantages of approaches *not* using standardized or high-level DSLs for data plane programming.

2.1.1 Software Devices

Section 2.1.1 is based on two publications by the author [1], [2]; and a collaboration between Daniel Raumer, Florian Wohlfart, Dominik Scholz, and Georg Carle [3].

The virtualization of network functions, moving networking functionality onto software-based systems, has increased the importance of CPU-based packet processing. Compared to dedicated networking hardware, a CPU-based system has several advantages, incentivizing the shift from hardware to software: commercial off-the-shelf (COTS) hardware is cheaper compared to dedicated hardware; a wide range of different software OSes and libraries, as well as COTS hardware to choose from exists; and development cycles are quicker compared to custom-made hardware that cannot be changed after production. We distinguish two major approaches to networking in software: performing the networking tasks as part of the OS's network stack or using an optimized packet processing framework.

In-kernel Network Stack

Common OSes like Windows and Linux provide a network stack that is located within the OS's kernel. The network stack is typically located within the kernel to protect the system from accidental or intentional misuse, which could lead to system crashes. The goal is a simple networking API for applications, without the need for low-level networking tasks. As such, the network stack supports a vast range of protocols and features. Performance, while important for modern networks with bandwidths potentially beyond 10 Gbit/s, is often only a secondary goal. In the following, we use Linux' network stack as an example for in-kernel networking.

Processing Steps: With the reception of the packet by the NIC the packet data is stored in main memory using direct memory access (DMA). As part of the OS's internal representation of the packet, metadata required for several protocols is added. Using a hardware interrupt, the kernel is informed that the packet is ready for processing. Depending on the mode, for instance, the New API [24] is used to poll all available devices and receive their available packets. The processing within the network stack happens layer by layer: on the network layer, basic checks are performed before a routing lookup is executed to determine the destination of the packet. If the result is a local application, the packet is passed to the next layers, for instance, UDP or TCP. After this, the packet is handed over to the application via the socket API, resulting in a copy operation of the data to user space.

For transmitting packets, either after a forwarding decision was made for a received packet or for packets from user space, processing happens top-down. Eventually, the next hop and outgoing interface are determined, and the packet is handed to the outbound NIC.

While passing the network stack, further processing of the packet can happen. For instance, packet filtering can be applied using hooks of the netfilter module.

Performance Limiting Factors: The Linux network stack, as an example for network stacks in mainstream OSes, is designed for general purpose networking including routing and higher layer protocols. The goal is to provide an easy to use interface for applications and high robustness, without the user having to care about the underlying networking subtasks. However, this results in a trade-off between usability and performance, creating performance bottlenecks when surpassing 1 Gbit/s Ethernet [25]–[30]. Bolla and Bruschi formulate bottlenecks that limit the performance of software packet processing platforms as follows: the "CPU limits the number of packet headers that can be processed, while bandwidth and latency of the I/O buses limit the total throughput" [31].

The complexity of possible per-packet processing is defined by the available CPU cycles—the most common bottleneck in CPU-based packet processing systems. If a processor operates with a maximum frequency f and requires $\widetilde{C}$ cycles to process a single packet on average, the resulting maximum packet rate $\widetilde{p}$ can be computed with Equation 2.1:

$$\widetilde{p} = \frac{f}{\widetilde{C}} \tag{2.1}$$

Vice-versa, for a packet processing system equipped with a 3 GHz CPU to achieve line-rate of 14.88 Mpps for 10 Gbit/s Ethernet, less than 207 CPU cycles must be spent per packet on average. We use Equation 2.1 throughout this thesis to determine the metric of CPU cycles per packet from the measured packet rate.

Another limiting factor is the system's memory. Per-packet allocation and deallocation of memory for received or sent packets requires a significant number of CPU cycles spent using the memory BUS. According to Dorado et al., this can be up to "63 % of the CPU usage in the reception process of a single 64 B sized packet" [25]. Further, packet data is copied multiple times. When receiving a packet, data is copied from special DMA memory of the NIC to the kernel's internal buffer. From this buffer, the packet data may be copied to user space applications. Doing these operations at high speeds causes the CPU to idle waiting for data loaded via the memory BUS, consequently increasing the CPU cycles required per packet.

Due to the generality of the network stack, the internal data structure is bloated, containing metadata for several protocols. This can also cause bottlenecks due to the memory hierarchy. If packet data including metadata, or other data structures like the routing table, surpasses the size of the cache, data has to be fetched from slower memory. This can even lead to cache thrashing effects, requiring constant load operations for cache misses, leading to idle processing cycles.

Another performance limiting factor are context switches between the user space application and the network stack residing in the kernel. While this increases the CPU cycles, the separation between user space and kernel is intentional as it increases the system's robustness. Another mechanism to improve system reliability are locking mechanisms. Linux uses active waiting locks to protect access to several data structures. This can be a potential bottleneck when multiple CPU cores are used for packet processing in parallel.

Packet Processing Frameworks

As general purpose network stacks are not able to perform even simple processing tasks at high speed, dedicated software packet processing frameworks have been developed. To circumvent or improve on the identified bottlenecks of in-kernel network stacks, they implement different techniques [28]: bypassing the kernel's network stack; using polling instead of interrupt-driven notification of new packets; pre-allocation of data structures, including packet buffers; no additional copy operations for packet data; and processing of packets in batches.

Over the years, several frameworks have been developed, using different architectures and programming languages. The following lists a selection of different approaches and discusses their advantages and disadvantages.

Data Plane Development Kit: The DPDK [32] is a framework consisting of optimized packet processing libraries and drivers for fast packet I/O on multi-core architectures. It completely bypasses the Linux kernel, allowing for high-performance packet processing applications in user space. All required processing steps, including low-level operations, are implemented in user space from scratch. While this increases the effort for developing and implementing the application, it also increases performance as only necessary processing is performed. Further, by bypassing the kernel the applications do not benefit from the guarantees in terms of robustness provided by the kernel's network stack. To avoid interrupts entirely, devices are accessed using active polling. Further, packets are processed in batches.

The libraries are optimized for common tasks, such as memory allocation of network packets, buffering, routing decisions, or hashing. DPDK uses a run-to-completion model, i.e., a single CPU core is responsible for the entire processing of a packet. All resources required during runtime are allocated during the startup of the application. This reduces CPU cycles spent per packet to create necessary data structures. During startup the application can allocate pools of memory for fixed-sized objects using a lock-less ring data structure. During runtime, an object can be retrieved from such a pool, circumventing the costs for allocating its memory. A sample use case are network packets. For this purpose, DPDK uses a special data structure, which is in turn stored in a memory pool. This packet structure is optimized to fit inside a cache line, while larger packets can be created by chaining multiple buffers together. It also contains metadata, which can be application-specific. DPDK further optimizes processing by utilizing hardware features of modern NICs, offloading functionality like the calculation of checksums to save CPU cycles.

DPDK does not provide out-of-the-box support for the IP protocol or higher layer protocols like UDP or TCP. All functionality has to be programmed for the specific application using DPDK's C libraries. The libmoon framework improves on this by providing an easy to use Lua scripting layer on top of DPDK. Using a scripting language to wrap typically verbose tasks simplifies the creation of applications using DPDK. libmoon maintains the performance of the underlying framework written in C through just-in-time (JIT) compilation and Lua's foreign function interface (FFI). Furthermore, libmoon extends the functionality of DPDK by providing a powerful library for working with protocol stacks, which is introduced in Section 3.3. This allows the creation of high-performance packet generators like MoonGen or routers [4], [33], [34].

Another framework that takes advantage of DPDK is Vector Packet Processing (VPP) [35]. Instead of using scalar processing, i.e., processing each packet individually, packets are processed as a vector. For all packets within this vector, one processing step is performed before moving on to the next step. As this approach makes better use of the caches, this reduces the overall processing latency.

Snabb: Snabb is a high-performance framework for traffic processing beyond 10 Gbit/s using open-source software. It is intended as alternative to existing hardware appliances used by Internet service providers or general network operators. All processing happens in user space bypassing the kernel, however, Snabb leverages features of modern commodity x86 architectures and NICs. Thus, the raw performance of the Ethernet device can be reached, while abandoning all security features of the network stack in the kernel. Using memory mapped I/O, the PCI device is directly bound to the Snabb application in user space. Read and write access of the NIC for packets is allowed via DMA. Similar to libmoon, Snabb uses a JIT-compiled language, in this case Lua. This enables writing applications using a high-level programming language, while the profiling of the JIT compiler allows for high-performance. For the same reasons, even low-level modules like the NIC drivers are written in Lua.

Computationally expensive tasks are offloaded to the hardware whenever possible. However, compared to DPDK, Snabb prefers offloading to the CPU over offloading to the NIC. For instance, checksums as part of headers like IPv4 or TCP can be computed using CPU-specific Single Instruction Multiple Data (SIMD) instructions. Advanced Vector Extension (AVX) instructions are typically highly parallelized and can be computed in fewer CPU cycles compared to using traditional operations. Snabb offloads these tasks to the CPU, using, for instance, Data Direct I/O, instead of offloading it to the NIC due to a different design philosophy. Offloading to the CPU using SIMD instructions can be performed for workloads, independent of the used NIC. Thus, the resulting code is simplified and not NIC-specific. [36]

Snabb puts emphasis on virtualization to further optimize the usage of hardware resources. It uses single root I/O virtualization, vhost-user, Unix domain sockets, and QEMU/KVM to circumvent the kernel of the host system and to allow virtual switches to access shared data structures in the memory space of the guest machine.

In Snabb, each networking functionality is represented as small module called app. Within each app, representing a specific functionality usually implemented in hardware, packets are pulled from an input, processed, and finally send to the output. Several apps can be combined to form more powerful functionality, similar to actual networks. This approach, comparable to network function chaining, fosters reuse and simplifies the development of new applications.

netmap: Instead of bypassing the kernel, netmap provides high-speed packet I/O as kernel module [37]. The framework tries to solve the identified problems of the network stack: per-packet allocation of memory, overhead produced by system calls, and multiple memory copies of the same data. Similar to DPDK, netmap uses batch processing and an optimized, lightweight, and pre-allocated data structure to store packet data. Further, netmap uses memory that is shared between all processes, requiring no copy operations when forwarding a packet from one interface to another. While an error in the user-written application cannot cause the kernel to crash, other netmap-specific data structures can be corrupted.

netmap has been used to implement the Click modular software router [38], [39] and the switched Ethernet VALE [40].

PF_RING: PF_RING [41] is a kernel module that provides an API for high-performance capturing, filtering and analyzing of packets. It uses polling to avoid the impact of interrupts, pre-allocation of packet buffers, multi-core processing, and memory mapping techniques. Using the zero copy module, no copy operations are required to access the packet data from user space.

	DPDK	Snabb	netmap	PF_RING
Kernel bypass	+	+	−	+/−
Polling mechanisms	+	+/−	+	+
Memory pre-allocation	+	+	+	+
Shared mapped memory	+	+	+	+
Batch processing	+	+/−	+	−
Multi-core support	+	+	+	+
Simplified packet buffer	+	+	+	+
Standardized DSL	−	−	−	−

Table 2.1: Comparison of techniques used by different high-performance software packet processing frameworks

The downsides are that only one application can access the data in parallel and misbehavior can lead to system crashes.

Comparison: Table 2.1 showcases different features the presented frameworks use to achieve high-performance packet processing. They either completely bypass the existing network stack of the kernel and also lose all guarantees provided by it, or extend and improve selected aspects of the kernel's network stack. Processing can be further sped up by using batch processing of packets. All frameworks use variations of the same conceptual ideas: they use polling mechanisms instead of costly interrupts to start processing received packets; memory allocation during the startup of the application to reuse buffers during the actual runtime of the application, reducing CPU cycles spent on memory allocation; memory regions shared between NIC and application to remove or reduce copy operations; multi-core support by using, for instance, receive-side scaling; and a simplified data structure representing network packets, only containing critical metadata. Further, the packet buffer does not differentiate headers or payload of the packet as this is entirely up to the processing of the application.

None of the introduced packet processing frameworks uses a standardized DSL for creating networking applications. Therefore, development is different for each platform, limiting the portability of the solutions. Even basic processing steps, for instance, network and transport layer processing, have to be implemented from scratch, although selected optimized algorithms, like LPM, may exist. Further, the frameworks are limited to software platforms, i.e., porting a solution to dedicated hardware is not trivially possible. Depending on the framework's architecture, programming language, and API the development effort can vary. Using a scripting language like Lua can simplify the development process. As these frameworks allow low-level packet processing, however, the applications can be optimized, resulting in potentially higher performance. In general, the introduced packet processing frameworks are able to reach 10 Gbit/s line-rate [28], [42], however, results can vary depending on the used hardware and complexity of the application-specific processing.

The discussed frameworks are only a selection of existing high-performance packet processing platforms. A notable mention is PFQ as it provides its own functional language for defining packet processing tasks, called pfq-lang [43]. While this DSL is similar to approaches like Extended Berkeley Packet Filter (eBPF), PFQ-lang is not as widely used.

2.1.2 Programmable Hardware

Programmable hardware platforms are systems of which the core processing functionality can be changed after manufacturing. This is in contrast to fixed-function hardware like ASICs. This definition includes that the device behaviour cannot only be changed by setting different options, but instead fundamentals like parsed headers, bytes modified or lookups performed in data structures based on the packets data can be programmed. Examples for such hardware platforms are network processing units (NPUs) and FPGAs. The advantage of these platforms is the high performance, extending even beyond 100 Gbit/s [44].

Smart NICs using, e.g., the Netronome NFP-4000 NPU [45] chip, are specialized NICs using network flow processors. These are many-core processing units that allow highly parallelized processing and multithreading. Each flow processing cores can process a single packet. The NFP-4000 can be programmed, for instance, using a variation of C, called micro-C.

FPGAs can be programmed using low-level hardware description languages like VHDL. Before the existence of higher level compilers that allow to compile, for instance, C [44] or P4 to VHDL, domain-specific knowledge was required to create basic packet processing logic. Early approaches, for instance, tried to increase the performance of regular network stacks by "boosting" selected processing steps dynamically [22]. The authors called their FPGA-based prototype architecture the Programmable Protocol Processing Pipeline, which shares the same abbreviation as the primary subject of this thesis, P4.

Designing FPGA-based packet parsers that can cope with ever increasing speeds is ongoing research [44], [46]. While most of the design principles are specific for FPGA targets, the authors also propose high-level description languages to simplify the creation of packet parsing programs on FPGA targets [44]. According to Gibb et al., adding programmable packet parsers doubles the required area for parsers, however, this only increases resource consumption from 1 to 2 % [46].

Compared to the discussed software processing platforms, these hardware platforms excel in performance, both throughput and latency. The downsides are that the available resources are limited and that the devices are even more difficult to program, requiring domain-specific knowledge. Although the targets might offer higher-layer languages like C or DSLs for parts of the packet processing like packet parsing, developed applications are still not portable. Creating the same application on two different targets, e.g., NPU and FPGA, requires completely different languages, design principles, and program architectures.

2.1.3 OpenFlow

OpenFlow [47] is a network protocol that allows to modify the data plane of network devices, enabling SDNs by decoupling the control plane from the data plane. This was one of the first successful attempts to standardize a cross-vendor interface for remotely managing network switches. The protocol allows to add, modify, or delete rules composed at the control plane for the forwarding table of a switch. Packets that do not match any rule can be forward to the control plane to create new or updated rules. The configurability of OpenFlow-enabled devices is limited: both, the possible match fields and actions performed after a match, are defined by protocol and cannot be arbitrary. Exceptions are target-specific extensions that allow to match additional fields or perform additional actions.

The downside with such a limited number of match fields is the continuous development and evolution of new and existing protocols. For instance, if a data center operator wants to use a new or custom tunneling protocol within the data center, OpenFlow and OpenFlow-enabled devices will likely not or not yet support the protocol. Consequently, no rules that match the custom tunneling protocol can be inserted and matched against as the hardware needs to be updated. This requires both monetary and temporal resources.

Compared to both the introduced software and hardware platforms, OpenFlow provides a standardized interface for changing the forwarding behavior of network devices. While this is a step towards SDNs, the approach also limits its usability: Outside of the forwarding behavior, customizing general device behavior is not or barely possible. Furthermore, the standardized match fields and actions are limited as the data plane is still coupled to protocol formats.

2.2 Domain-specific Languages for Programming Data Planes

In the previous section, we surveyed packet processing frameworks that can be used to create networking applications for a specific target, typically CPU, NPU, or FPGA. Every target has a different architecture, uses different features and approaches, and comes with its own advantages and disadvantages. However, they also all use different programming languages and offer different APIs. Hence, an application programmed for one device is typically not portable to another platform, especially not from one hardware platform to another. For these platforms, DSLs like Click [38] for creating routers on software-based platforms or G [48] for FPGAs exist.

To solve these problems, DSLs have been proposed to program the data planes of heterogeneous devices and target platforms. Compared to what is possible using, for instance, OpenFlow, their goal is to further decouple the data and control plane and extend the programmability of the data plane. However, they are not intended to replace OpenFlow as they are two different use cases: OpenFlow is a protocol that manages the communication between control and data plane, while these new DSLs focus on the programmability of the data plane. Even fully programmable data planes still require means to communicate with the control plane to retrieve new or update existing configuration.

Several DSLs have been proposed, including, for instance, packetc [49], baker [50], or NPL [51]. However, these either lack features or were not adopted by a wide range of vendors. Another example that entirely decouples the data plane and protocols is Protocol-oblivious Forwarding (POF) [52]. Instead of defining actions that perform on protocol fields, POF defines match keys based on arbitrary bit ranges of the packet. This key is then used to perform actions, which are also protocol agnostic. An initial hardware prototype implementation by Song is based on the Huawei NE5000 platform [52]. Ongoing research on POF is primarily presented by Chinese research groups [53].

In the following we introduce more prominent approaches that are adopted by different hardware vendors and are the focus of this thesis.

2.2.1 Extended Berkeley Packet Filter

Section 2.2.1 is based on a collaboration between Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurt, Krzysztof Lesiak, and Georg Carle [5].

Berkeley Packet Filter (BPF) exist since 1992 in UNIX OSes [54]. They were developed to improve the performance of packet filtering, for instance, using tcpdump. In Linux, the BPF VM was extended with a JIT compiler, which enabled the Linux kernel to translate the VM instructions to assembly code on the x86_64 architecture in real time [55]. Further improvements and generalizations led to eBPF. Instead of sole packet filtering, eBPF can be used for general purpose filtering systems, allowing the creation of traffic control, shaping, and tracing applications.

The 64 bit instructions of the eBPF VM are mapped to assembly instructions of the underlying hardware architecture using an interpreter or JIT compilation. The memory that an eBPF program can access is set up by the kernel on startup depending on the type of the program. Further, the kernel allows calling a predefined and type-dependent set of functions. [56], [57]

eBPF programs are like regular programs running on hardware with two exceptions: all eBPF programs are subject to static verification before being loaded into the kernel for security reasons. This step ensures that the program contains no backward jumps, which are required for loops. Since Linux 5.3, bounded loops are allowed [58]. Further, the program must not exceed a maximum number of 4096 instructions, which was extended with Linux 5.1 to one million instructions [59]. The goal of the static verifier is to protect the kernel from DoS attacks by making sure that the eBPF program terminates. Thus, a malicious eBPF program cannot compromise or block the kernel, allowing unprivileged users to use eBPF in the kernel [60]. The second difference are special key-value stores, coined maps, used to maintain program state or communicate with other applications. In contrast to using dynamically allocated memory, files, or sockets, eBPF programs only have read and write access to maps. Maps are memory regions set up before the program is loaded into the kernel. Thereby, key size and value type, as well as the maximum size are determined at instantiation. Via a file descriptor, the kernel provides secure access to the map from user space. [56], [57]

Netronome has adopted eBPF for their SmartNICs. eBPF programs for eXpress Data Path (XDP), which are usually executed within a kernel VM, can be offloaded to the Agilio CX SmartNIC, which is based on the NFP-4000 NPU platform [61]. Furthermore, XDP programs developed for Linux can be compiled to and run on FPGAs using the hXDP framework [62].

2.2.2 Programming Protocol-independent Packet Processors

P4 [23] is a standardized DSL for programming software and hardware data planes. As part of the Open Networking Foundation, it is managed by the P4 language consortium p4.org in several working groups for language, application, and architecture design. Unless mentioned otherwise, we use P4 to refer to the 2016 version of the language.

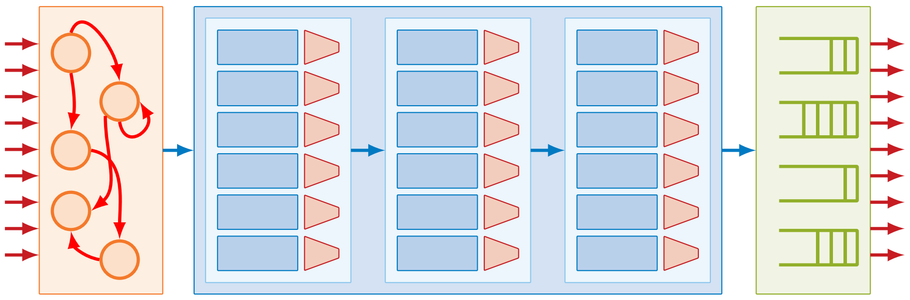


Figure 2.1: Protocol-Independent Switch Architecture model

Goals: Bosshart et al. describe the three goals of reconfigurability, protocol independence, and target independence. The processing of the device should be reconfigurable by the control plane in the field. Thereby, the program freely defines the parser of the device independent of existing protocols, i.e., moves the definition of protocol headers from protocol specifications to the programming language. Based on this definition, fields are named and extracted from the packet. These can than be used for further processing. Lastly, the programming should be agnostic regarding the underlying switch hardware architecture. The target-independent program should be transformed by a respective compiler to target-specific instructions. [23]

With these goals the P4 language achieves portability. The program is developed independent of the target. Depending on the desired usage scenario, the device can be chosen based on different key performance indicators (KPIs) like speed, reliability or cost. Further, it allows later upgrades or replacements with equal or completely different devices, while retaining the same software.

Using a standardized DSL and developing target-independent also enables rapid development cycles. As long as the functionality can be expressed by the language and the target fully implements the feature set, new functionality, program variations, or bug fixes can be developed using common software practices. These can then be tested on real hardware, without the need for a lengthy production of a new, purpose-built device.

Keeping software development independent of the hardware manufacturing can also be an important property for companies in regards to intellectual property. The device behavior can be developed in-house and closed source without the need to share critical know-how with other parties, e.g., competitors. In contrast, P4 programs, or snippets thereof, can be made open-source and incorporated into different programs, enabling reusability of code.

Implementing only required functionality increases the robustness and security of the application. In fixed-function devices, any functionality that exists, but is not required still increases the risk that it is faulty and can be exploited.

Language Components: A basic P4 program consists of three different stages as arranged in the Protocol-Independent Switch Architecture (PISA) architecture model[1] shown in Figure 2.1: parser, pipeline with match-action tables, and deparser. The parser is a finite state machine that defines the parsing of the packet, ending in accepting or rejecting the packet. For the current state, the next following packet data is parsed into a header data structure. Based on this data, a transition to the next state can be performed.

The centerpiece of P4 is the match-action programming paradigm. In the match-action pipeline, matches can be performed on the extracted header data or metadata. Thereby, P4 defines different match types: exact, ternary, and LPM. If a matching entry is found, the corresponding action is performed. An action may perform various processing, e.g., manipulating header or metadata fields. Table entries, defining the match key and action to be performed, are not defined by the P4 program. Instead, tables are populated by the control plane, while the P4 program solely defines the properties of the table.

Finally, the deparser constructs the outgoing packet. It defines which headers are applied in which order. Packet payload, i.e., data that was not parsed, is appended.

Other constructs, like registers or hash functions, are not part of the P4 specification as they would be too restrictive for different devices. Instead, P4 defines externs, i.e., functionality that may be provided by a concrete target device.

Architecture Models: Network devices are heterogeneous. They differ in various aspects, including price and throughput, but also in processing complexity. Therefore, P4 does not assume every network device to have the same architecture. Instead, a P4 program can be written for a specific target architecture, i.e., the programming model of the target device. Each architecture provides a logical view of the data plane processing pipeline: arrangement of different stages like parser and match-action pipelines, additional functionality like traffic manager, and extern function capabilities. For a given architecture it is the job of the respective compiler to generate target-specific code from the P4 program. For example, the PISA model is the primary architecture for the $P4_{14}$ version of the language and consists of only a single match-action pipeline stage. In contrast, the Portable Switch Architecture (PSA) model targets a more general switching device. Its complete pipeline consists of multiple match-action stages and defines a set of externs, e.g., for hashing.

For the sake of simplicity we will use the PISA architecture throughput this thesis for explanations unless mentioned otherwise. However, explanations can be applied to other architecture models. [63]

Control Plane Interface: The P4 language consortium also proposed an API for communication between control plane and P4-enabled data planes, coined P4runtime. The goal of this interface

[1] The PISA architecture model was designed for $P4_{14}$ and is not used by modern targets. However, we use it for explanations throughout this thesis due to its simplicity.

is to allow remote management of such devices, including loading and starting new P4 programs, as well as managing the program state by inserting, updating, or deleting match-action table entries. In this work, we focus on the capabilities and performance of the data plane by itself, excluding the P4runtime interface.

2.2.3 Future Trends

The introduced languages are only a first step towards fully programmable networks and NOSes. They still lack in several aspects, in particular portability. Although P4 tries to overcome this issue by providing explicit architecture models, a program still needs to be developed for a certain architecture, using the respective number of, for instance, match-action pipelines. To combat this, Soni et al. [64] propose a higher-level abstraction, using libraries of composed P4 code snippets. This approach further increases modularity, composability, and portability of P4 programs.

To add an additional layer of abstraction, higher-layer languages like Lyra [65] have been proposed. Its goals are to allow cross-platform compilation, overcoming the mentioned portability issues, allow for program composition, i.e., running multiple programs in the same data plane, and being extensibility. Compared to P4, Lyra uses a simplified programming abstraction, coined one-big-pipeline. Within the pipeline, a packet is processed by different algorithms, composed of simple if-else, read, and write statements. The Lyra compiler suite is responsible for creating target-specific code, e.g., translating if-else statements to P4 tables. Using a simpler and higher level programming abstraction also has the additional advantage that the P4 programs generated by the Lyra compiler may even use less hardware resources and lines of code compared to the manually developed counterparts.

2.3 Target Platforms

The introduced data plane programming languages target different software or hardware platforms using varying architecture models. Furthermore, they come with different toolchains, e.g., for P4 program compilation, control plane interfaces, or device debugging. In this section, we introduce and discuss the properties of the targets and toolchains important for the context of this thesis. While other architectures exist, we chose these targets as they are commonly used by related work and are easily available commercially.

2.3.1 Software on CPU

Software-programmable CPU-based systems run on general-purpose COTS hardware. In general, they are flexible, i.e., it is simple to fit a program to the target, and extensible, i.e., new functionality can be added easily. While a CPU-based system can provide almost arbitrary and complex functionality, this comes at the cost of performance. Typically, the system's throughput is limited by the processing power of the CPU. Further, latency and jitter may be impacted by interrupts and cache effects.

p4c

p4c is the primary frontend compiler for P4 programs. It generates a platform-independent intermediate representation in JSON format that can be used by backend compilers to generate target-specific code. As part of the p4c suite, compilers for the behavioral model version 2 (bmv2), eBPF, and Userspace eBPF (uBPF) targets, as well as for the generation of program graph representation and testing exist.

Emulation

The behavioral model version 2 (bmv2) is the P4 reference software switch implementation targeting the Mininet emulation framework [66]. While it is not intended for high-performance applications, it represents a close-to-specification implementation of the standardized P4 language. Therefore, bmv2 is best suited for prototyping applications before testing them on a target intended for production deployment.

BPF

The eXpress Data Path (XDP) [67] is available in the kernel since Linux 4.8 and provides a stateless and high-performance programmable packet processing path. To achieve this, network drivers of various NIC vendors were extended to expose a common API, allowing processing at the lowest level of the entire software stack. As a result, the eBPF programs, which are configurable from user space, have read and write access to the packets stored in DMA buffers of the NIC's driver. Due to its simplicity and location before the kernel's network stack, XDP is ideally suited for discarding packets early in the processing path. Use cases include the mitigation of DoS attacks, as well as quick forwarding of packets to egress buffers without the interception of the network stack. [5]

P4 can be used to program XDP using the p4c-XDP [68] backend for p4c.

DPDK

The t4p4s [69] P4 compiler generates platform-independent C code. Based on the JSON intermediate representation, t4p4s generates its own high-level intermediate representation. This can be used with target-specific libraries, for instance, for the DPDK backend, to generate high-performance, P4-programmable switching devices on COTS hardware. The t4p4s target is actively developed and primarily serves academic purposes, allowing rapid prototyping of new high-performance applications and P4 mechanisms like asynchronous processing of externs [70]. P4 programs use the v1model architecture model. We use the name t4p4s synonymously for both the P4 compiler and the DPDK-based P4 target throughout this thesis.

2.3.2 Hardware

Compared to software targets, hardware devices are limited in resources and flexibility for programming and extending them. However, they excel in throughput and latency metrics.

We compare three different hardware-based targets: NPUs are many-core architectures, whereby the cores are optimized for packet processing; FPGAs can be programmed using a hardware description language (HDL) to provide almost arbitrary functionality based on the chips available logic and memory blocks; and purpose-built ASICs, which have a dedicated, but limited instruction set.

Smart NIC

The Netronome Flow Processor (NFP)-4000 chips from Netronome [45] are available for Agilio SmartNIC NPUs equipped with 10, 40, or 100 Gbit/s ports. This platform relies on a 32 bit many-core architecture, equipped with 48 packet processing cores and 60 freely programmable flow processing cores. Netronome offers support for various programming languages, including backend compilers for C, eBPF, and P4. The NFP-4000 uses the v1model architecture model for P4 programs.

FPGA

P4 programs can be synthesized for various FPGA-based hardware platforms [71], [72]. The academic P4→NetFPGA project by Ibanez et al. [72] provides an open-source framework to compile P4 programs using the Xilinx SDNet, generating a hardware design for the NetFPGA SUME. This board is based on the Xilinx Virtex-7 XC7VX690T FPGA and es equipped with four 10 Gbit/s Ethernet ports, as well as a total of 216 Mbit static random-access memory (SRAM) and 8 GB of DDR3 RAM.

Ibanez et al. also propose an event-driven P4 architecture model and discuss their prototype implementation for the NetFPGA SUME [73]. Compared to standard P4, this model also generates events, for instance, for the expiration of timers, packet generation, or en- or dequeuing packets in buffers, which can then be processed by the P4 pipeline. The authors argue that these events allow for even more complex P4 applications, while only requiring an additional amount of the FPGA's resources of up to 2 %.

ASIC

The Intel Tofino is a fully P4-programmable switching ASIC [74]. The first generation chip is equipped with up to 65 100 Gbit/s Ethernet ports, while the ports of Tofino 2 are capable of up to 400 Gbit/s. Always 16 ports are bundled together and served by one P4 pipeline, of which each individual pipeline can be configured with a separate program or connected for programs requiring more complex processing pipelines. The Tofino uses the Tofino Native Architecture model, which is an extended version of the PSA. The list of additional functionality comprises a traffic manager, various externs including meters, counters, and registers, and a traffic generator. The Tofino ASIC guarantees processing of packets at line-rate, independent of program complexity. [74]

2.3.3 Other Targets

Other prominent targets, compilers, and transpiler not used in this thesis include various P4 backends for BPF [75]–[78]; the discontinued PISCES framework [79], a P4-enabled software switch based on Open vSwitch (OvS) on top of DPDK; VPP with pvpp [80]; the P4-enabled Xilinx Raymax SmartNIC for cloud acceleration [81]; SmartNICs by Pensando; P4FPGA [71]

	CPU	NPU	FPGA	ASIC
Throughput	+	++	+++	++++
Latency	$> 10\,\mu$s	$5\,\mu$s to $10\,\mu$s	$< 2\,\mu$s	$< 2\,\mu$s
Jitter	−−−−	−−−	−−	−
Resources	++++	+++	++	+
Flexibility	++++	+++	++	+

Table 2.2: Comparison of different P4 target architectures. Performance categorizations are estimates for available products based on own measurements and related work [7], [83], [84].

to generate Bluespec System Verilog code targeting Xilinx and Altera FPGAs; and P4-to-VHDL [82] for 100 Gbit/s FPGAs. Further targets including their supported P4 language versions and architecture models are listed in related work [74, Table 2] or online[1].

2.3.4 Comparison

Section 2.3.4 is based on a collaboration between Dominik Scholz, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle [6].

Generally speaking, CPU, NPU, FPGA, and ASIC-based devices can be categorized using performance KPIs like shown in Table 2.2. In addition to runtime characteristics like throughput, latency, and jitter, we compare the amount of resources a device can offer, as well as, its flexibility, i.e., how easily the target can be extended with custom functionality.

The advantages of CPU-based systems are the virtually unlimited amount of resources, as additional hardware can be added or existing hardware can be upgrade easily, and high flexibility, as arbitrary functionality can be implemented. Consequently, complex programs with custom functionality, including externs or even customized architecture models for P4 programs, can be developed and executed. However, the downside of CPU-based systems is typically the performance in form of throughput, latency, and jitter: throughput is limited by the processing power of the CPU, i.e., each packet requires a certain number of CPU cycles; latency and jitter is influenced by interrupts and cache effects [28].

NPU architectures and their processing cores are optimized for packet processing and provide high throughput and consistently low latency. However, due to the specialized hardware architecture, fewer commercial products are available compared to, for instance, fixed-function NICs. Compared to a pure software target, the flexibility is also reduced, however, common programming languages like C are often available to program an NPU. Similar, resources are more limited, in particular, available on-board memory and its architecture will restrict program complexity and performance.

FPGAs often surpass NPU and CPU targets in throughput, latency, and jitter metrics. As arbitrary functionality can be implemented, FPGAs are highly flexible. The challenge is for

[1] https://github.com/hesingh/p4-info

	bmv2	t4p4s	NFP-4000	NetFPGA SUME	Tofino
Use Case	Reference Model	Academic	Commercial	Academic & Commercial	Commercial
Backend	multiple	DPDK	SmartNIC	FPGA	ASIC
Toolchain	CLI	CLI	CLI/GUI	CLI	CLI/GUI
Architecture	v1model & PSA	v1model	v1model	SimpleSume-Switch	TNA & v1model
Throughput	<< 1 Gbit/s	< 10 Gbit/s	< 100 Gbit/s	10 − 100 Gbit/s	> 100 Gbit/s
Resources	++++	++++	+++	+	++
Flexibility	++++	++++	+++	++	+
Maturity	+++	+	++	+	+++

Table 2.3: Overview of different P4 targets investigated in this thesis. Performance categorizations are subjective estimates for available products.

the toolchain to synthesize a working hardware design utilizing the limited resources of the hardware architecture without, for instance, timing issues. A downside is that programming and debugging FPGAs requires domain- or hardware-specific knowledge. Therefore, for the untrained, developing network functionality using an HDL is a time-consuming task.

ASICs are highly optimized, for instance, through a high degree of parallelism. Consequently, they can achieve higher throughput with lower latency and jitter compared to the other target platforms. This comes at the cost of flexibility: an ASIC is a fixed-function chip, i.e., the expressiveness of the instruction set is limited. This makes it impossible to add new functionality after manufacturing. Similar to FPGAs, the total amount of resources is limited.

Summarizing, a clear trade-off for the introduced CPU and hardware-based targets can be identified: either a target excels for runtime properties like throughput, latency, and jitter, or it offers high flexibility and resources to develop complex networking functionality.

For the target platforms relevant for this thesis, a summary of the intended use case, platform-specific properties, and performance values is shown in Table 2.3.

2.4 Data Plane Programming Taxonomy

Cross-platform DSLs like P4 enable functionality, that was previously only available on end hosts, to shift to data planes within the network. This also opens up the opportunity for completely new applications. Consequently, plenty of research has emerged, discussing which applications can and should be shifted to the data plane. Figure 2.2 shows the number of publications in recent years including the keywords *P4*, *Programmable Data Plane*, and *Programmable Switch* in the publication title when searching only on dblp[1].

[1] `https://dblp.uni-trier.de/`

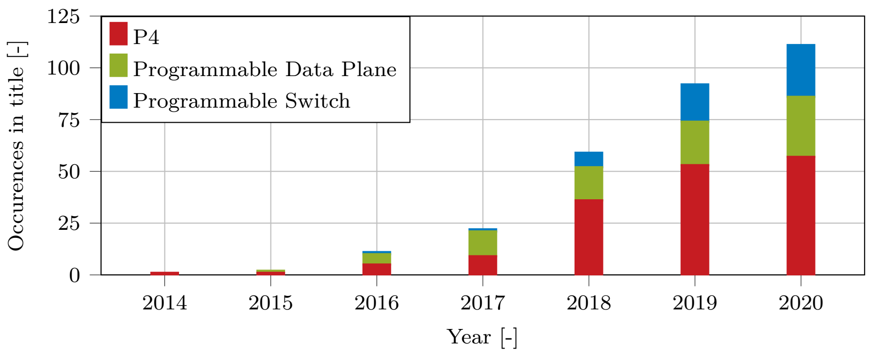


Figure 2.2: Popularity of P4 and programmable data planes in related work. Search for keywords in paper titles using dblp.

This section provides an overview of different applications and research domains for P4-programmable data planes. We discuss which areas are still lacking research, and for which areas the remainder of this thesis contributes to fill discovered holes.

2.4.1 P4 Application and Research Domains

Kfoury et al. [85] and Hauser et al. [74] surveyed the landscape of P4, including approximately 300 and 500 references, respectively. The primary focus of both publications are research domains and applications. Kfoury et al. split their findings into seven categories: in-band network telemetry, network performance, middlebox functions, accelerated computations, Internet of Things, security, and testing. In comparison, Hauser et al. list six different P4 domains: monitoring, traffic management and congestion control, routing and forwarding, advanced networking, network security, and miscellaneous research domains. Compared to Kfoury et al., Hauser et al. specifically list related work on applications for industrial networks and applications that require or benefit from cryptographic functionality in programmable data planes. Hauser et al. not only discuss a broad survey of applications, but provide an overview of P4 language concepts, architectures, targets, and control planes. In contrast, Kfoury et al. further discuss and compare the surveyed applications and provide an outlook on future research trends. [74], [85]

2.4.2 Discussion

Due to the overwhelming number of existing and newly published papers regarding P4 and related topics it is difficult to maintain an overview of all application and research domains. The presented taxonomies show that P4 is applied to multiple, vastly different areas: basic switching and routing, gathering of network statistics like in-band network telemetry [86]–[88] or on a per-flow basis [89]–[91], DoS and other attack mitigation [92], [8], in-network computation [93], [94], and other applications commonly used in data center environments. However, P4 is also applied to other network domains with sometimes drastically different requirements, including, for instance, wireless communication [95], mobile communication [96]–[98], Internet of Things [99], or industrial networks [100]. Research also extends to the language itself as P4 enables program optimization [101], [102], as well as program verification for bug-mitigation approaches [103]–

[106]. Lastly, a related area of research is the integration of control plane interfaces and support for network controllers for NOSes [107]–[109].

The variability also extends to the different layers of the ISO/OSI model that a P4-enabled device operates at: ranging from L2 and L3 switching and routing, to middleboxes [110], and even higher layer protocols or applications like DNS [111] or consensus [112], [113], seemingly all network layers are represented. While many applications require no persistent state across individual packets (e.g., [114]), stateful processing is becoming a topic of interest [9], [115].

We notice two shortcomings among these existing taxonomies. Neither of the presented taxonomies includes an extended survey of P4 performance aspects, for both, targets and applications. We provide a survey of P4 target performance, approaches for performance improvements, and modeling of programmable data planes in Chapter 4. Furthermore, we extend the discussion on the benefit of cryptographic functions in programmable data planes, in particular for industrial networks, in Chapter 5.

2.5 Key Results

This chapter demonstrates the recent history and efforts towards fully programmable networks. While highly potent network stacks exist for all major OSes, they lack performance due to their complexity. For software-based systems, high-performance packet processing frameworks have been developed to create arbitrary networking functionality on COTS devices. Through various techniques like kernel-bypass or memory pre-allocation for packet buffers, these frameworks exceed in performance compared to common network stacks, however, at the cost of having to implement all functionality from scratch. For hardware platforms, programmable network devices based on NPUs and FPGAs exist. They exceed in performance metrics like throughput and latency, but the downside is domain-specific knowledge required to program the devices.

The commonality of these approaches is the lack of a standardized and unified language to program network devices, independent of the concrete underlying architecture or hardware. The advantages of such an abstract programming language are similar to other high-level programming languages: quick development cycles; increased portability as the implemented program may run on several different target platforms; and decreased entrance hurdle for programming network devices. These advantages open up the process of creating new network functions on high-performance devices to a wider audience. Consequently, new functionality is moved to the data plane.

Domain-specific languages like P4 have been proposed exactly for this purpose. Using standardized and abstract architecture models, as well as a specified set of instructions, data planes of network devices can be—almost—freely programmed. Although the P4 instruction set is fixed, additional functionality may be offered by a target and be used within the P4 program through externs. In addition to the already mentioned advantages, P4 is independent of existing protocol definitions. Thus, the programmer has full reign over how the data plane interprets data in existing, custom or future protocols. For the operators of network infrastructure, the intellectual property of the implemented network functionality remains within the company, instead of contracting the hardware manufacturer. Alternatively, functionality can be open-sourced and reused by many.

The survey of current ongoing research and application domains for programmable data planes shows that P4 is applied to a plethora of areas, not only limited to data center applications. In parallel, over a dozen of software- and hardware-based targets have been developed and are either available open-source or commercially with throughputs extending to the terabit range. A property that is not thoroughly discussed in related work are performance aspects, in particular, how program complexity influences the performance when run on different target platforms. Depending on the concrete target or application, the importance of different performance metrics like throughput, latency, or resources, might differ. Therefore, we propose a modeling framework in Chapter 4 that analyzes the influence of individual P4 program components.

Another area for languages like P4 are industrial networks. However, these require increased security measures, already on the data link layer. In Chapter 5, we discuss how the functionality to calculate cryptographic hashes in programmable data planes can further enhance the applicability of P4. Further, we will show how this functionality can be used to protect, for instance, against DoS attacks.

2.6 Statement on Author's Contributions

Section 2.1.1 is based on two publications by the author [1], [2]; and a collaboration between Daniel Raumer, Florian Wohlfart, Dominik Scholz, and Georg Carle [3]. The discussion of network stacks and software packet processing systems (cf. Section 2.1.1) was extended and modified to fit the context and research questions of this thesis. Similar, the comparison of different packet processing frameworks and techniques presented in Section 2.1.1, including Table 2.1, was enhanced for this work. The notation presented in Equation 2.1 was unified to match the notation used throughout the remainder of this thesis.

Section 2.3.4 is based on a collaboration between Dominik Scholz, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle [6]. The entire section was significantly extended for this thesis. Table 2.2 is a modified version based on [6, Table 1], while Table 2.3 was created for this work. Due to these changes, Section 2.3.4 goes significantly beyond the scope of the original work.

The taxonomy presented in Section 2.4 was created by the author for this thesis and extends the related work.

Chapter 3

Measurement Methodology

In this chapter we introduce the underlying terminology, methodology, and tools used throughout this thesis. We want to analyze the performance of networking devices. For this, we formulate a measurement methodology to perform measurements and obtain result data. We define performance metrics to assess and quantify the performance of the analyzed networking device based on the gathered results of the experiment. We created a testbed environment that implements this methodology. Thereby, we enforce a specific experiment structure and automate many steps of the methodology, ensuring that the results are reproducible. Within this framework, we use different tools that help executing the experiment in a reproducible fashion.

3.1 Terminology and Experiment Metrics

We define an **experiment**, in our case a network experiment, as a set of **measurements** involving one or more interconnected networking nodes, called DuTs. The goal is to solve a certain research question and creating, refining, or confirming a formulated **model** as shown in Figure 3.1. Thereby, the model is a mathematical representation of the system's behavior. During an experiment, one or more individual measurements are performed. A networked system, and therefore the experiment, typically has several input parameters that influence the system's behavior. A measurement run investigates the outcome for one possible combination of these input parameters. To accurately model the system and achieve high coverage, the experiment consists of many measurements with different combinations of input parameters. The experiment creates **artifacts** that help to answer the research question. Vice-versa, the set of artifacts fully describes the experiment. Artifacts are data that describe the setup, e.g., tools, measurement scripts, topology information or device configuration, and measurements results, e.g., raw data or plots.

Measurement results are gathered and used to generate **KPIs**, describing the performance of the DuT. RFC 2544 [116] defines four KPIs for evaluating network devices: throughput, latency, frame loss rate, and back-to-back frames. In addition to these runtime properties, other metrics, like overall functionality or setup time, might be of interest. In this thesis, however, we focus on

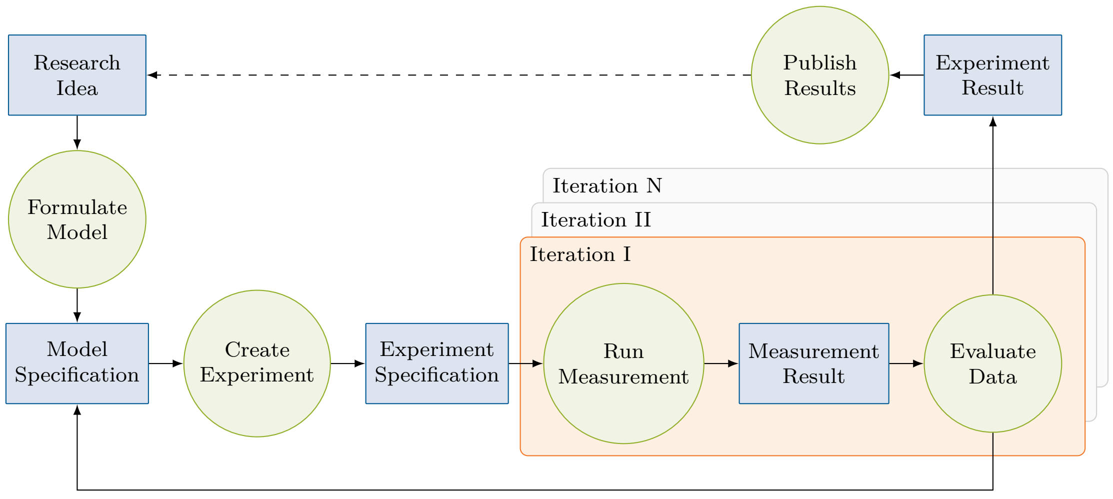


Figure 3.1: Abstract experimental process

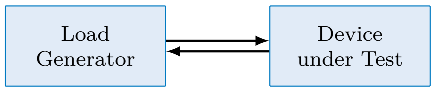


Figure 3.2: Measurement setup used throughout this thesis

a slightly adjusted set of KPIs: **throughput**, often in the form of packet rate, as measure for a DuT's maximum capable load; **latency**, as measure for the processing time of an individual packet from ingress to egress of the DuT; and **resource usage**, as measure for the percentage usage of the DuT's available hardware resources when performing a certain network task or program.

Experiment artifacts help in creating reproducible results. In this thesis, we follow the ACM's definition for three levels of reproducible research [117]:

- **Repeatability**: using the same experimental setup, the same research group can reliably repeat the experiment outcome.
- **Reproducibility**: using the same experimental setup, a different team can reliably reproduce the experiment outcome.
- **Replicability**: using a different experimental setup, a different group can reliably replicate the experiment outcome.

Throughout this work, experiments will be performed using the two-node setup shown in Figure 3.2 unless mentioned otherwise. In this setup, one node acts as the DuT, the target of the experimental evaluation. A second node, the load generator, will be used to generate traffic that the DuT will be subjected to. The DuT will process the traffic and send back the result to the load generator, which also serves as sink. For all experiments, the nodes are connected via 10 Gigabit Ethernet (GbE) cables.

3.2 Orchestrating Reproducible Network Device Measurements

Sections 3.2.1, 3.2.3, and 3.2.4 are based on a collaboration between Sebastian Gallenmüller, Dominik Scholz, Henning Stubbe, and Georg Carle [10]; and a joint work between Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [4].

Testbed environments are essential for network research experiments. In the context of this thesis, we distinguish two types of testbed designs: purpose-made one-off testbeds, typically only used to answer one research question by one research group; and multi-purpose testbeds, used over a long time period by multiple users in parallel. While the former sort of testbeds are quick to set up and use for measurements, their temporary existence clearly limits reusability. We focus on the latter: long-term, multi-user, multi-purpose testbeds, intended for reproducible network measurements. This sort of testbeds requires sophisticated orchestration software. A testbed controller manages the testbed and its usage, i.e., users, testbed resources, and ongoing measurements. This section introduces our approach for orchestrating such a multi-user, heterogeneous testbed with a focus on reproducible results.

3.2.1 Types of Testbeds and Testbed Controllers

Bajpai et al. [118] formulate best practices for performing reproducible research experiments, and, therefore, also indirectly for testbeds. They emphasize the need for repetition, performing multiple iterations of an experiment with slightly adjusted parameters. To solve this, they recommend a high degree of automation. This also helps with their observation that documentation of the whole experimental workflow is key. This does not only include a description of the setup and all metadata, but also the collection of all artifacts. [118]

There exist a wide variety of scientific testbeds, characterized by their purpose, size, and governance. Wired networks, cloud computing, Internet of Things, mobile networks and 5G, or Internet-wide measurements are typical areas for testbed deployments. They can be locally operated, for instance, by a chair of an university, or by national or even international co-operations. Predominantly found in literature are domain-specific testbeds on national or international level [119]: the french Grid5000 [120] for grid computing with 5000 nodes; European initiatives like Fed4Fire [121], a collection of several, primarily wireless testbeds [122], and OneLab [123], a federation for testbeds for the Internet of the future, including embedded, IoT, wireless, and Internet overlay testbeds [124]; the Chameleon [125] and GENI [126] cloud testbeds (USA); or PlanetLab [127], a world-wide testbed for "broad-coverage network services" [127]. In particular, testbeds for distributed computing research can scale in size to international co-operations.

Nussbaum [119] investigated the three testbeds Chameleon, CloudLab, and Grid'5000. He focused on their ability to execute reproducible network experiments and concluded that the investigated testbeds are indeed capable of performing such experiments. However, the testbeds neither guarantee nor enforce the creation of reproducible experiments. Zhuang et al. [128] evaluated the testbeds Emulab, PlanetLab, Seattle, and GENI for teaching. They found several problems, such as involuntary configuration changes during the running experiments and fluctuating bandwidths that may impact the repeatability of the experiments running on these testbeds. Although focusing their analysis on teaching aspects, these findings also hold true for research experiments.

Locally operated testbeds are often reused between multiple publications of the same research group with slight alterations to the hardware setup. Thereby, a specification of the complete setup is not always provided, limiting the reproducibility of the results.

Testbeds based on simulation and emulation frameworks exist [11], [12], however, in this work we focus on testbeds built from physical hardware.

A testbed controller manages the resources of the testbed and supports the experimental procedure. OMF [129] is a testbed controller which has its own DSL to program network experiments. OMF enables the creation of reproducible experiments relying on automation. The DSL of OMF allows the specification of complex experimental workflows as Petri nets.

The OneLab federation solves the issues of managing distributed heterogeneous resources that are controlled by multiple authorities. The focus is on providing easy and uniform access to these testbed resources. OneLab does not further enforce a certain experimental workflow or how reproducibility can be achieved. [124]

We focus on small to mid scale testbeds for network devices and assume a scaling of multiple racks or even multiple sites. Furthermore, our testbed and controller focus on reproducible experiments: how they can be created, supported, and enforced by design.

3.2.2 Testbed Life Cycles

Testbeds typically involve two different stakeholders that use the nodes of the testbed: users, e.g., researchers, and administrative staff, operating and maintaining the testbed. Based on these entities, users, administrators, and nodes, testbed processes can be modeled as three separate life cycles as shown in Figure 3.3. The common denominator where all life cycles intersect, is the testbed's management node and controller software.

Users want to answer a research question and, therefore, will follow the experimental workflow. During the **user life cycle**, users will create a model based on which they will create an experiment using the testbed. This experiment will consist of one or more measurement runs performed using the testbed controller and its nodes. Each run will create measurement results, which are then evaluated. The circle continues: based on the evaluation, the model and respective experiment can be refined, and new measurements can be executed.

The **node life cycle** is triggered by the testbed controller during a measurement run. Nodes taking part in the measurement typically have to be booted first. Before the device can perform the actual measurement, all required configuration has to be made. At the end of the measurement, the node can be shut down to reset all configuration. This allows a clean slate for the next iteration.

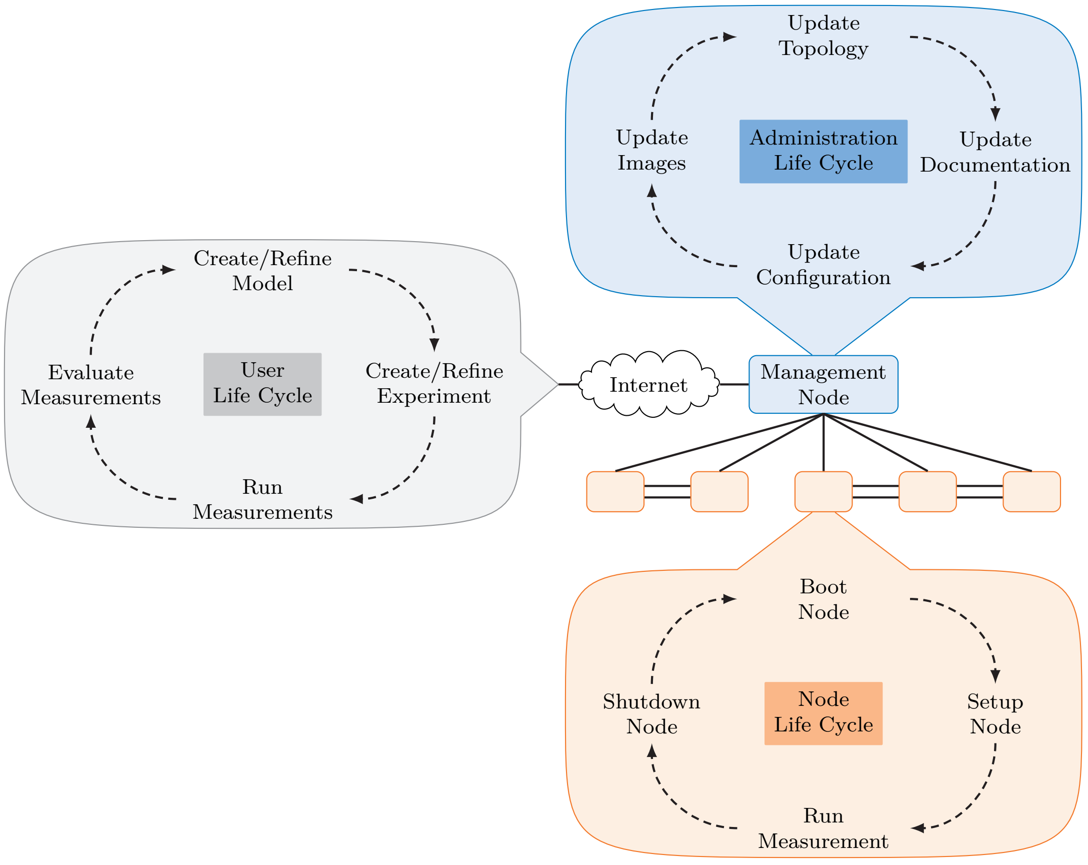


Figure 3.3: Common administration, user and node life cycles of experimental testbeds

Lastly, the **administrative life cycle** is for maintaining or improving the state of the testbed. This includes, but is not limited to creating or updating the configuration of the management node and controller, images used by the network devices, the network topology, and user documentation about the experimental process.

3.2.3 Requirements Analysis for Testbed Controller

A testbed controller is required to offer, orchestrate and manage the entities and actions of the different life cycles. Although the execution of reproducible experiments is of highest priority, other requirements regarding the usage of the testbed need to be observed. The following analyzes the requirements for a testbed controller based on the introduced life cycles with increasing importance.

Maintainability (RQ1): Although not of scientific value, the maintainability of the testbed and its controller from an administrator's perspective is crucial to allow continuous operation of the testbed. This includes a first setup of the management node and controller software, as well as continuous maintenance. During the life time of a testbed, hardware can be replaced or added or configurations can change. This must be reflected by the controller to reduce the administrative overhead.

Usability (RQ2): A testbed is commonly used by many users in parallel. These users might have a different background and familiarity with networking devices, or measurement and evaluation software. Consequently, the controller must be easy and intuitive to use and offer an abstract interface for testbed nodes. To prevent accidental or intentional interference of an experiment by one user with the experiment of another, the controller must ensure, that a resource is used exclusively by one user. However, a particular node might be required by multiple users, facilitating the need for temporal sharing and allocating the node resources of the testbed.

Heterogeneity of Network Devices (RQ3): The Internet is a collection of heterogeneous devices communicating with each other using a common protocol. Consequently, networks involve a wide variety of drastically different participants. This includes, but is not limited to, smart NICs built to accelerate specific network tasks, embedded, resource-constrained devices, packet processing software on off-the-shelf hardware, and high-performance switches with dedicated ASICs. To allow effective experiments, a testbed controller must be able to incorporate a wide variety of heterogeneous devices.

Experiment Automation (RQ4): Network devices often require complex configuration, consisting of several steps and configuration files, before the desired behavior is obtained. Providing an accurate setup is required not only for performance reasons, but also to allow recreation of the experiment at a later point. The best way to gain a reproducible setup is by fully automating the process. Specifying all configuration steps and files can mitigate potential sources of errors and reduce invested time.

Experiment Isolation (RQ5): Due to the distributed nature of network experiments, different devices may influence the behavior of other devices. As this would influence the experimental outcome, the testbed and its controller needs to isolate experimental devices from other devices that are not part of the investigated network. Only then, the influence of unwanted effects on the experimental result can be diminished.

Experiment Recoverability (RQ6): The trial-and-error approach to answer a research question is often used for experiment-driven science. Applying this methodology to network experiments causes frequent configuration changes for the participating devices and the experimental network. In the worst-case, these modifications might result in an unresponsive device or network outage. Independent of the state of the network or individual devices, the testbed controller must be able to recover from faults and reset the experiment to a well-defined state. Commonly, the controller restarts the whole experiment, starting from an initial state again.

Publishability of Artifacts (RQ7): Only a full specification of experimental artifacts allows reproducibility. This includes scripts, results, evaluation tools, and other necessary information that needs to be well-documented and accessible for others. The testbed controller should aid the researcher with the final goal of publishing all experimental artifacts.

3.2.4 Plain Orchestrating Service

We have developed the plain orchestrating service (pos) testbed controller to model the life cycles and fulfill the requirements of experimental testbeds. pos enables operating local or

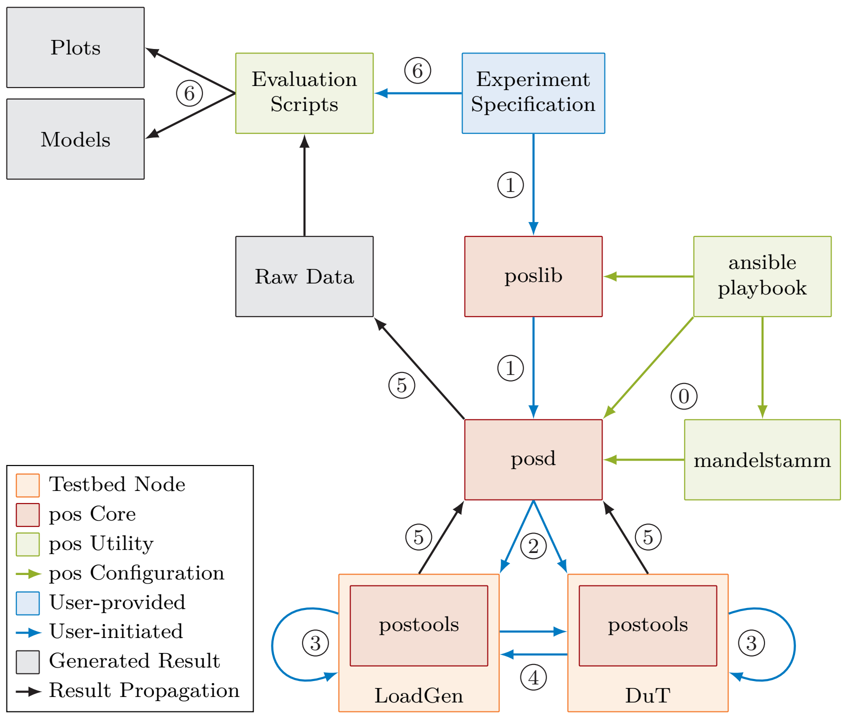


Figure 3.4: Steps performed during an experiment using the pos testbed controller. The sample experiment uses two nodes, a load generator (LoadGen) and the DuT. The ansible playbook is used to set up ⓪ the testbed and components of pos, whereby mandelstamm generates images for the testbed nodes. The experiment specification is passed ① via the poslib to posd. posd initializes ② and configures ③ the participating testbed nodes according to the experiment specification. The measurement is executed ④ and raw data is gathered ⑤ via posd. Lastly, the raw data is evaluated and results are generated ⑥.

remote, heterogeneous, multi-user testbeds, and enforces an experiment structure that supports reproducibility. The domain-independent testbed controller can be used to manage a large number of network devices in an experiment.

Components of pos

pos consists of the core controller software, utility software and scripts provided by the testbed user shown in Figure 3.4.

Controller Core: The heart of pos is the daemon **posd**, running the controller software. It consists of a postgresql database that stores the configuration of the testbed and its nodes. Further, posd exposes a RESTful API that allows to interface with the testbed nodes and all steps of the experimental procedure. To manage the temporal allocation of testbed resources, the daemon can optionally host a web interface with access to a testbed calendar, documentation about the testbed topology and hardware, and remote out-of-band capabilities.

Testbed users can use the pos library **poslib** to interact with the daemon's API. This python library consists of wrappers for all RESTful API calls, which are also available as command-line interface (CLI).

The third part of the core pos software are the **postools**. This is a collection of utility tools available on the testbed nodes. Their purpose is to provide an easy method for the nodes to interact with posd or with other testbed nodes.

Utilities: **mandelstamm** is a tool to automatically and reproducibly build Linux live images for network devices of the testbed. Its modularized architecture allows to easily add configurations for new images. In contrast, the core modules insert mandatory software into every image: configuration required by pos to control the node and programs commonly used by testbed users. Providing the configuration for an image once allows to re-build the image on a regular basis for regression testing. Further, the specification can be published for reproduction by others.

A collection of **evaluation scripts** can be used to analyze commonly produced raw data, for instance, for throughput or latency measurements using MoonGen or profiling using perf. For these data formats, several different representations, including cumulative distribution function (CDF) plots or regular and high dynamic range (HDR) histograms, are generated automatically. This serves testbed users as starting point and allows to quickly bootstrap their experiment evaluation. The evaluation scripts also consist of a module for the automated generation of models based on the measurement data. This module uses heuristics and linear regression to determine one or more partial functions matching the data. The modeling module will be further explained in Section 4.3.

The complete configuration of the management node, the controller software pos, and the testbed nodes is centrally maintained in an **ansible playbook**. The configuration of the management node includes the network setup, e.g., interfaces and IP addresses, and services like DHCP or DNS. User management, i.e., users allowed to login to the management node and/or use pos, can be done manually using ansible or using external authentication like LDAP. The playbook also includes installation and configuration instructions for the latest stable versions of pos software. Lastly, a list of testbed network devices and their network and pos-related configuration is maintained. Using software like ansible increases the maintainability for the administrators of the testbed (RQ1). New testbeds can be set up by only providing a new configuration file for the management node. Changes, e.g., new pos versions, can be deployed remotely, fully automatically and centrally to one or more testbeds. This includes adding new, updating existing, or removing testbed nodes.

We have also developed a fully virtualized version of an exemplary testbed, including management node and pos, called **vpos**. Typically a testbed is bound to the hardware and, in particular, the topology of testbed devices. A virtualized version not only allows simple topology changes, but also permits sharing the testbed including its controller. Consequently, other researchers can use vpos to create and test experiments in the testbed. Once the experiment works as expected, the experiment specification can be used to perform the experiment in a physical testbed. This allows to more efficiently use the hardware resources as these are typically used by multiple users and, therefore, are time-constrained.

Provided by the Testbed User: Software not included by pos is the **experiment specification**, which has to be provided by the testbed user. The script contains successive calls using the poslib to interact with testbed devices via posd. Details of of the experiment specification using pos are provided in the following sections.

Experimental Structure

pos enforces a specific structure to program and execute network experiments. The reason is so that all experimental artifacts can easily be gathered and processed for publication. This is vital for other researchers to achieve reproducibility.

Experiment scripts created by the testbed user distinguish two different types of files: script files, containing an individual sequence of commands that should be executed on the experiment node to perform setup or experiment steps; and parameter files, used to parameterize the script files according to the concrete instance of the measurement run. This approach in splitting the experiment specification is similar to HTML—defining the syntactical structure of the content—and CSS—defining the design. For instance, a script file defines the insertion of a routing table entry for the DuT with the name `$TABLE_ENTRY`, the variable file assigns `$TABLE_ENTRY` the value `10.0.0.10/25`. This separation allows different iterations, referred to as runs, of the experiment to use different variables without having to duplicate the entire script. Further, this mechanism can be used to run the experiment in a different (hardware) setup, for instance, using a variable to specify the port of the NIC that should be used.

Experiment scripts, which can be any executable running on the target node, are further split per experiment node to simplify the experimental structure. Every experiment node requires two separate scripts for the different phases of the experiment: in the **setup phase** all necessary configuration of the node is performed. Consequently, the setup script contains the complete configuration of the experiment node. In the **measurement phase** a multiple runs of the experiment are performed, yielding measurement results. The measurement script therefore contains the sequence of commands executed to obtain the measurement results of an individual run. Combined, the per-node setup and measurement scripts define the complete experiment. Enforcing the user to provide them per node enables reproducibility of the experiment.

pos differentiates between three kinds of variables: *local variables* are defined for each experiment node and only accessible on this particular node; *global variables* are defined for and accessible from all experiment nodes; and *loop variables* are like global variables, but change for every run of the experiment.

Experimental Workflow

Figure 3.5 describes the high-level workflow of an experiment controlled using pos. The example assumes the experiment setup presented in Figure 3.4, consisting of two nodes, load generator and DuT. In general, the workflow is independent of the number of nodes involved. Compared to, for instance, the OMF testbed controller, pos assumes a simpler experimental workflow. It distinguishes three separate phases that are traversed by all experiment nodes: setup, measurement, and evaluation.

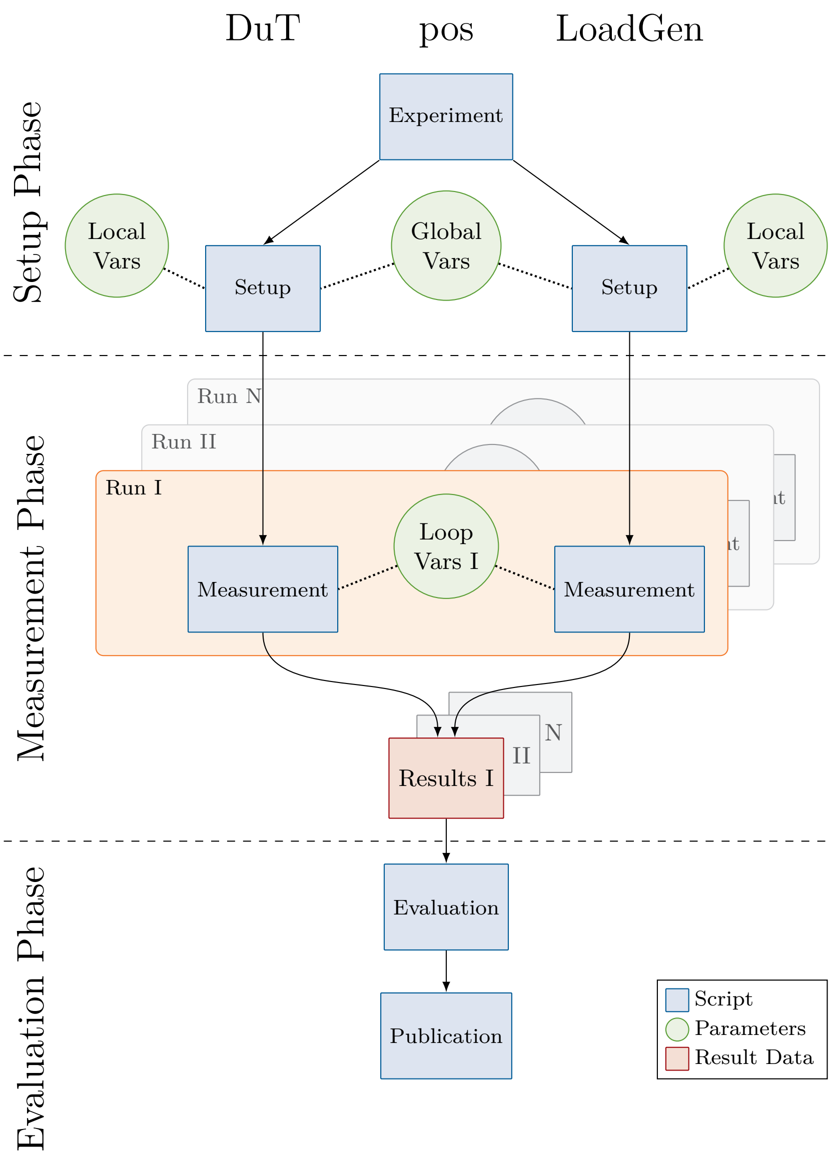


Figure 3.5: pos experimental workflow

The *experiment script* provided by the user is the central part of the experiment. This script defines all actions for every node of the experiment for each experimental phase. Every action performed is thereby an interaction with the posd API using the poslib.

Setup Phase: First, the experiment script allocates the desired devices, in our example the DuT and LoadGen. As we operate a multi-user testbed, we use an integrated calendar to temporally separate the experimental devices between users (RQ2). Allocations via the calendar are enforced, i.e., only if the calendar indicates that the devices are free for the planned duration of the experiment, the allocation can be created.

Afterwards, the devices are configured by loading the global and loop variables. The local variables are loaded for each experiment host individually. Moreover, an image is selected for every device, if the device supports images like Linux live images for COTS x86 devices. This image is being used by the device for booting. pos relies on live-boot images to avoid any shared state between the different executions of the experiment. Such images enforce repeatability, as the OS repeatedly starts from a well-defined state, and the researcher must automate and thereby document the device configuration (RQ4 and RQ6). Optionally, boot parameters for the image can be set.

Finally, the experiment script instructs pos to start the devices. Thereby, the boot is internally executed by posd using the respective initialization interface. This abstraction improves the usability for users, as they do not need to take care of boot specifics (RQ3). Details about this interface are explained in Section 3.2.4.

Once the experiment hosts have finished booting, pos deploys a set of utility tools, the postools. These can be used during measurement runs to read or communicate variables, or synchronize testbed nodes using barriers. Furthermore, all the output of the executed commands can be captured and automatically uploaded to the management server as a result.

Lastly, the per-node setup scripts can be loaded and executed to complete the setup phase. The entire initialization process and configuration of a network device is automated via user-defined scripts (RQ4).

Measurement Phase: During the measurement runs, pos executes each node's measurement script. The number of executions depends on the number of individual parameters contained in the loop variables file. Each of these loop parameters can represent either a single value or a list of values. A pos experiment performs measurements for each possible combination of loop parameters. If lists are used as parameters, pos automatically generates the cross product over all the available loop parameter values to ensure full coverage. Then, for every set of parameters contained in the calculated cross product, it executes the measurement script of every node once. Parameters must be carefully chosen, as the exponential growth in the measurement runs may cause infeasibly long experiment completion times.

pos' loop variables reflect possible input values of a packet processing system presented in Figure 3.4. A single loop variable represents an input, a list of concrete values for a loop variable represents the scaled input values. Calculating the cross product of all loop variables, therefore, represents all possible combinations of inputs for the DuT. Performing a measurement run for every input combination results in all possible output values for the investigated system.

pos automatically queues one run after another, starting the next run only after the current run has been completed. The complete output of the experiment script is captured and stored in the result folder of the experiment. This enforced central collection of artifacts, including the output of the utility tools and all executed scripts, variables, and device hardware and topology information, guarantees publishability (RQ7).

Evaluation Phase: During the evaluation phase the user can either use a custom-made script or a modified version of the pos evaluation scripts. The evaluation script processes the result files either after all runs have been completed or asynchronously during their runtime. Due to the enforced structure of pos experiments, each run of the experiment is associated with a specific instance of loop parameters and result files. This information can be used to evaluate loop experiments, which is supported by the included evaluation scripts. The evaluation script can filter or aggregate specific parameters and values, thus enabling automated evaluation. Using a set of different representations (histogram, CDF, and violin plot) the scripts generate default plots for standard parameters. Researchers can adapt or extend the evaluation script to reflect the concrete experiment setting and to create custom graphs. The generated plots are exported to multiple formats, e.g., tex, svg, and pdf.

Further, the modeling component of the evaluation scripts can be used to deduce models from the measurement data.

Our structured experimental workflow allows all artifacts linked to an experiment, i.e., executed scripts, used variables, generated results, and created plots, to be gathered. The *publication* script bundles these artifacts into a release format, e.g., an archive or a repository. In addition, it generates a website and inserts all the collected artifacts documenting the experimental structure in a format that can be easily read by researchers.

Controller Features

The complete core of pos, consisting of posd, poslib, and the postools, is written in python to enable portability of the software across different OSes. This allows to run the daemon and CLI on a wide range of management nodes, while the postools are not limited to a certain OS distribution for testbed nodes.

posd is implemented using python's modern asynchronous libraries asyncio and aiohttp. This allows for lightweight parallelism, processing dozens of API calls, queued commands, or scheduled jobs virtually in parallel using only a single CPU core. Documentation for the posd, poslib, and postools APIs is automatically generated (RQ2). Persistent data, including the testbed configuration, node configuration, allocations and commands are stored in a postgresql database.

Modularized Interfaces to Access Heterogeneous Devices: From a user's perspective, the concrete method to, for instance, start up a network device is not of interest. In other words, the user wants to just start the device, but not care about the steps that have to be performed to actually start it. In general, the user wants to perform several different actions with a network node that can be reduced to the following: **starting** a network node that is currently offline or even without power, also referred to as booting a node; **stopping** a network node to shut it off and reset all configuration, independent of the current state the device is in; and **executing a command** on a running network node. This can be a script or configuration file that should be performed.

The CLI of pos exposes these functions to the user. The daemon, however, has to map the abstract command to the steps required by the concrete device. Therefore, posd uses interfaces

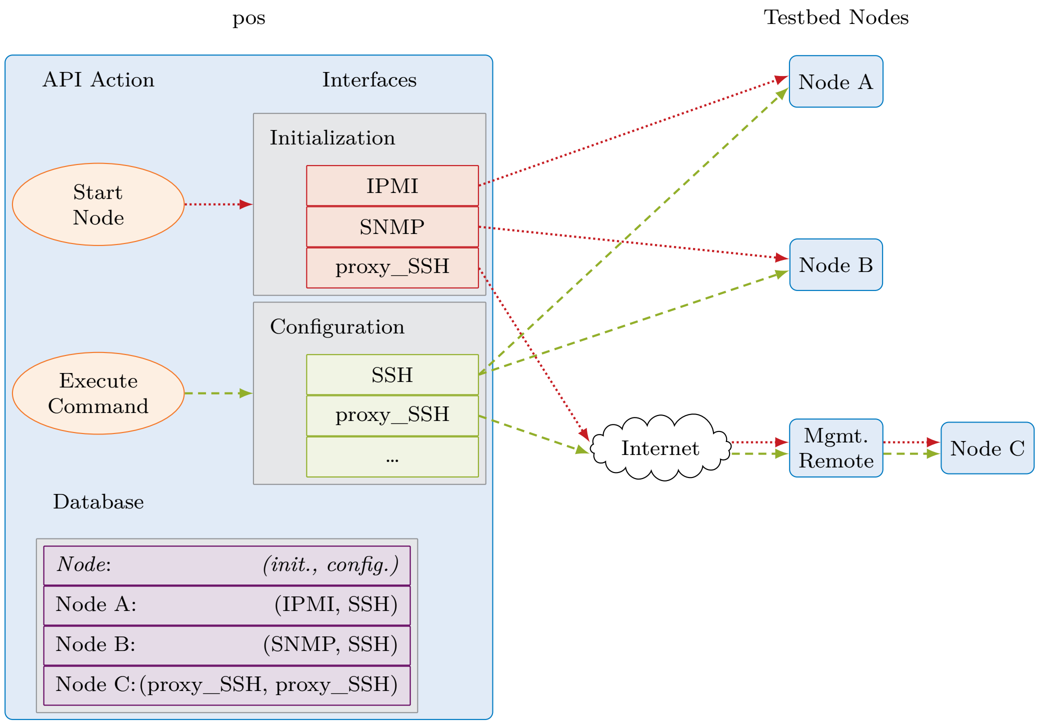


Figure 3.6: pos configuration and initialization interfaces

to map the API action to the correct implementation for the category of hardware as shown in Figure 3.6. Based on the information in the database, the type of the target node is resolved and respective interfaces are used.

pos differentiates two mandatory interfaces per node. The **initialization interface** defines functions for remotely changing the state of the device. The primary functions to reset or boot a node are *power on* and *power off*, i.e., turning the node on or off. This may, for instance, include providing or cutting the power for the device. A typical representative for this interface is the Intelligent Platform Management Interface (IPMI). However, pos also supports other initialization interfaces, including Intel's vPro or AMD's Pro features, or a remotely switchable power plug that triggers a device reboot. Via this interface the initialization of a device can be triggered out-of-band (RQ6), i.e., the device can be reinitialized in the case of configuration errors.

After the experimental node has been initialized, the device can be configured via the **configuration interface**. This includes the general execution of commands on the device, including setup and experiment steps. A typical configuration interface is ssh, for instance, to access Linux servers.

Interfaces can be combined to form new interfaces. This can be used to create, for instance, proxy interfaces, accessing a node using ssh via a proxy management node in another network (see *proxy_ssh* in Figure 3.6).

There exist two additional interfaces that are optional: an energy interface in case the node supports energy measurements; and a switch interface in case operating the node also requires to make modifications to a switch in the network. An example use case for this interface is provided in Section 3.2.5.

The advantage of pos interfaces is the reusability of interface implementations. For a certain type of node an interface has to be implemented only once. Detailed per-node configuration of an interface is provided via the database, e.g., the location of the ssh private key when using the ssh interface. Further, adding a new type of device to the testbed requires to only implement the interfaces for this type. This allows the testbed controller to manage numerous heterogeneous resources (RQ3). The abstraction of the CLI provides ease of usability for researchers (RQ2).

Executing Commands on Testbed Nodes: The pos API allows to perform arbitrary commands on the testbed nodes. This can be a shell command or any executable script like the setup or measurement scripts. Commands can be executed in one of three ways: Synchronous execution blocks until the result of the command is returned. Asynchronous execution enqueues the command into the command queue of the target node and returns immediately. The command is executed once the previous command in the queue finished successfully. The result of the command can be fetched at a later stage using the command's ID.

Finally, pos allows the scheduled execution of commands as job. This can either be at a fixed date in the future, or as soon as possible after a given date. Scheduled execution has several advantages. First, it further improves the temporal usage of testbed nodes as an experiment can be executed as soon as the nodes are available. Second, experiments can be scheduled to run at times where the user is not available, e.g., during the night. Lastly, the scheduler also supports repeated execution of commands. This can be used for regression testing, for instance, performing an experiment monthly to detect performance regressions, or to perform testbed maintenance. Administrative duties include automatically updating testbed images or the documentation about the topology on a regular basis (RQ1 and RQ2).

Evaluation

We have shown that the experimental workflow of pos models the general workflow of network experiments. The design of pos experiments enforces reproducibility, while allowing to be used for heterogeneous multi-user testbeds. Utilities provided by pos guide the user through every step of the experiment.

Deployments: Since the end of 2019, pos has been used to orchestrate four testbeds. These independent instances of pos manage testbeds for different research domains, with different number of nodes and testbed users. An overview of these testbeds and relevant usage statistics is given in Table 3.1.

In total, over 240 unique users have used the testbeds since their establishment. Of these, more than 130 have performed experiments using pos within the span of 18 months. For various reasons, the testbeds can be used without being forced to use pos. This explains the discrepancy between users with access to a testbed, and users that use pos.

Domain	Wired Networks	Distributed Computing	TSN	Wireless Networks
Established	2012	2019-10	2020-11	2019-12
Uses pos since	2019-10	2019-10	2020-11	2019-12
Nodes	27	80	15	8
Users (Access)	128	113	18	11
Users (pos)	40	78	13	5
Calendar Events	3423	8736	1422	493

Table 3.1: Statistics about testbeds orchestrated using pos (as of 2021-08-16)

Performance: Due to the daemon's lightweight parallelization using an asynchronous REST API, pos can be used for mid- to large-scale testbeds using only a single CPU core. We tested booting up to 48 nodes in parallel. Thereby, the utilization of posd was not significantly noticeable. In fact, the bottleneck is the management network—in our case 1 Gbit/s—as every node will individually fetch an image using unicast. Using multicast for this purpose, booting all nodes in parallel required the same amount of the time as booting just one node individually.

Limitations of pos

Changing the network topology of the testbed or specific experimental setups is not possible using pos and requires manually and physically changing the hardware configuration. Using switches to create a highly interconnected L2 topology is possible, however, all network devices connecting load generator and DuT influence the measurement. For instance, a common forwarding delay of cut-through switches is approximately 300 ns [130]. L1 switches, which provide remotely controllable switching using optical fiber links between ports add less than 15 ns [131] latency. While these L1 switches would be a valid option to achieve reconfigurability of the network topology, the price tag is often six figures and does not merit the benefits.

pos only provides limited access for device configuration outside of the OS. While, for instance, BIOS settings or the firmware of the NIC may influence the device behavior, their interfaces are highly varying and manufacturer-dependent.

3.2.5 Case Study: Orchestrating a Combined Teaching and Research Testbed

The iLab teaching testbed is used to teach students in practical hands-on exercises about networking as part of the iLab teaching concept [132]. While it is primarily intended for teaching, its architecture and availability of hardware resources makes the iLab laboratory an ideal testbed for distributed computing experiments. However, this dual usage further increases the requirements for the testbed orchestration software: it has to combine the requirements for the teaching and research testbeds, in particular, research experiments must not interfere with ongoing teaching exercises. This section introduces the iLab teaching testbed and details how we use pos to also use the resources as measurement testbed for distributed computing.

Related Work on Teaching Testbeds

Searching for the keywords “teaching testbed” on dblp returns only a few results that are typically domain-specific: Silva et al. [133] propose a low-cost 4G testbed, consisting of two COTS devices, which can be optionally virtualized, and open-source software. Their aim is to improve 4G cellular teaching capabilities with additional teaching units. KYPO4INDUSTRY is a cybersecurity lab proposed by Celeda et al. [134]. They combine COTS devices and domain-specific hardware to teach about security threats in industrial control systems. Hanna et al. [135] developed a combined research and teaching testbed for wireless communications and networks. The testbed consists of COTS devices with wireless interfaces running more than 100 VMs. They use a testbed manager to enable multiple simultaneous users, while enforcing strict node isolation. However, common problems of VMs persist, namely longer execution times and VMs influencing other VMs running on the same parent node. While our testbed also consists of COTS devices, it can be used to teach a broad spectrum of networking domains.

Zhuang et al. [128] analyze the usage of large-scale public testbeds for teaching purposes. Investigating Emulab, PlanetLab, Seattle, and GENICloud they discover multiple problems when exposing student teams to those infrastructures, e.g., the lack of higher level debugging functions. Approaches based on simulation or emulation for teaching exist [136], [137], however, our goal is to teach students using different hardware platforms.

To verify the validity of results obtained using measurements, reproducibility of artifacts is an essential property for experiments [117], [138]. While we deem it an essential requirement towards a networking testbed, the related works shows that it should also be a requirement towards teaching testbeds: reproducibility is essential for teaching. Our proposed infrastructure combines both, a teaching and measurement testbed using shared COTS devices, managed using pos as testbed controller. While the design is motivated by the teaching concept, it converges with the requirements for a research testbed without having to repurpose parts of the infrastructure. The dedicated testbed controller simplifies the complete measurement procedure for users of the measurement testbed, while collecting all artifacts, enabling reproducible research. pos also guarantees that measurement experiments do not conflict with teaching experiments.

Extension from Teaching to Research Testbed

Using the teaching infrastructure consisting of 48 powerful and expensive devices for research is an obvious idea. However, previous hardware generations lacked critical remote control capabilities. Therefore, many tasks had to be done manually: the infrastructure could only be used for research during lecture-free periods, where all nodes had to be temporarily wired and manually started once. For the duration of the experiment, all nodes were continuously running and there was no remote recovery from error states.

With deprecation of the previous hardware generation, the new hardware was primarily selected with teaching requirements in mind. In addition, the problems encountered when being used for research were tackled. To allow integration of the hardware with pos, remote control capabilities and a permanent and dedicated measurement network were required.

Lab Setup Goals & Requirements

The lab room infrastructure should be designed to support the iLab didactic concept [132]. This requires supervisors to make frequent changes to the infrastructure, while allowing reproducible hands-on networking exercises for students. Further, to optimize the usage of costly hardware resources, the infrastructure should be available for research experiments performed by researchers. However, teaching has precedence, i.e., research experiments can only be performed when the nodes are not being used for teaching. To facilitate more students or experiments in the future or permit upgrades to the hardware, the design should be scalable.

Due to the dual usage as teaching and research infrastructure the requirements are two-fold. The didactic concept of the iLab course has the following requirements per isle used by a student team of two:

$\mathbf{RQ_T1}$ six Linux nodes to provide sufficient capabilities for creating network topologies and performing experiments;

$\mathbf{RQ_T2}$ full access to the hardware and OS to both, change physical topologies and configure software used;

$\mathbf{RQ_T3}$ Internet access via the browser to access instructions via the e-learning platform;

$\mathbf{RQ_T4}$ and easy access between nodes of the isle to simplify working on six nodes using a set of two peripheral devices.

Thereby, it must be guaranteed that

$\mathbf{RQ_T5}$ no additional services or background tasks are running on the nodes to give students the chance to understand all processes by themselves;

$\mathbf{RQ_T6}$ students are not able to actively or passively tamper with other teams' experiments (interference);

$\mathbf{RQ_T7}$ and students are not able to actively or passively tamper with the nodes (obstruction).

To be used as testbed, it is also required that

$\mathbf{RQ_R8}$ nodes can be accessed and controlled, i.e., power on and off, remotely without physical presence;

$\mathbf{RQ_R9}$ no manual cabling has to be performed for research experiments, permitting 24-hour usage;

$\mathbf{RQ_R10}$ and teaching operations have higher priority than testbed experiments: while a node is being used by a student for a teaching exercise, this node must not be used as testbed experiment node. However, if a student is supposed to work on a given node, although this node is currently being used as part of the testbed, the student is allowed to simply take the node.

We show that our life cycle-based tooling fulfills all requirements to allow the symbiosis of teaching and measurement testbed.

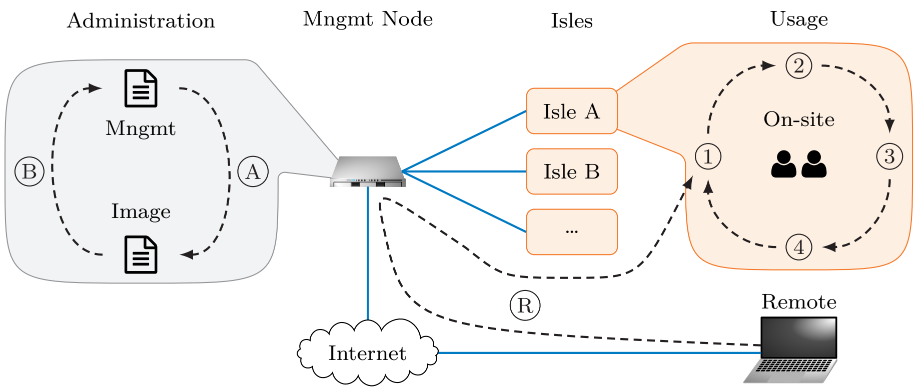


Figure 3.7: Administration (left) and usage (right) life cycles. Supervisors can continuously deploy and update the management and image configuration. Students can boot, use, reboot (optional) and shutdown/hand-off isles to the next team. Remote access is possible to access the administration life cycle as supervisor and the usage life cycle as student or researcher.

Life Cycle-based Tooling

Based on the goals, roles, and requirements introduced in Section 3.2.5 we have designed our tooling for the life cycles of our lab room.

Lab Room Life Cycles: Figure 3.7 visualizes the different components of the lab room's two asynchronous life cycles: The first life cycle is for supervisors (administrators) to create and update the configuration of Ⓐ the management node and Ⓑ the image used by the nodes.

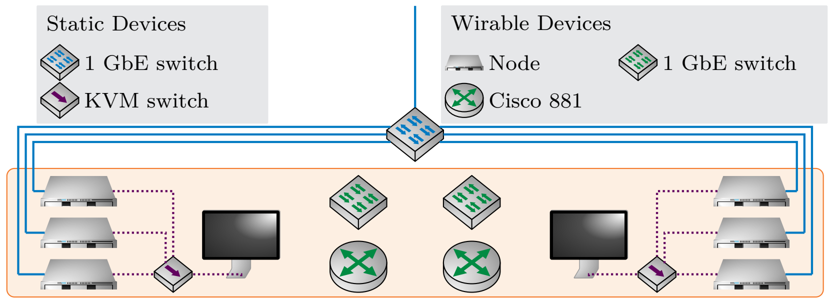
Second, the usage life cycle for teaching experiments performed by a student team consists of four steps: ① physically booting the nodes of an isle, ② performing experiments, ③ optionally rebooting the isle to reset the configuration, and ④ shutdown of all nodes to handover the isle to the next student team.

Ⓡ Remote access and control of testbed nodes is possible via the management server for both students and researchers. Thereby, the steps of the usage life cycle can be performed remotely, while only supervisors can update the configuration.

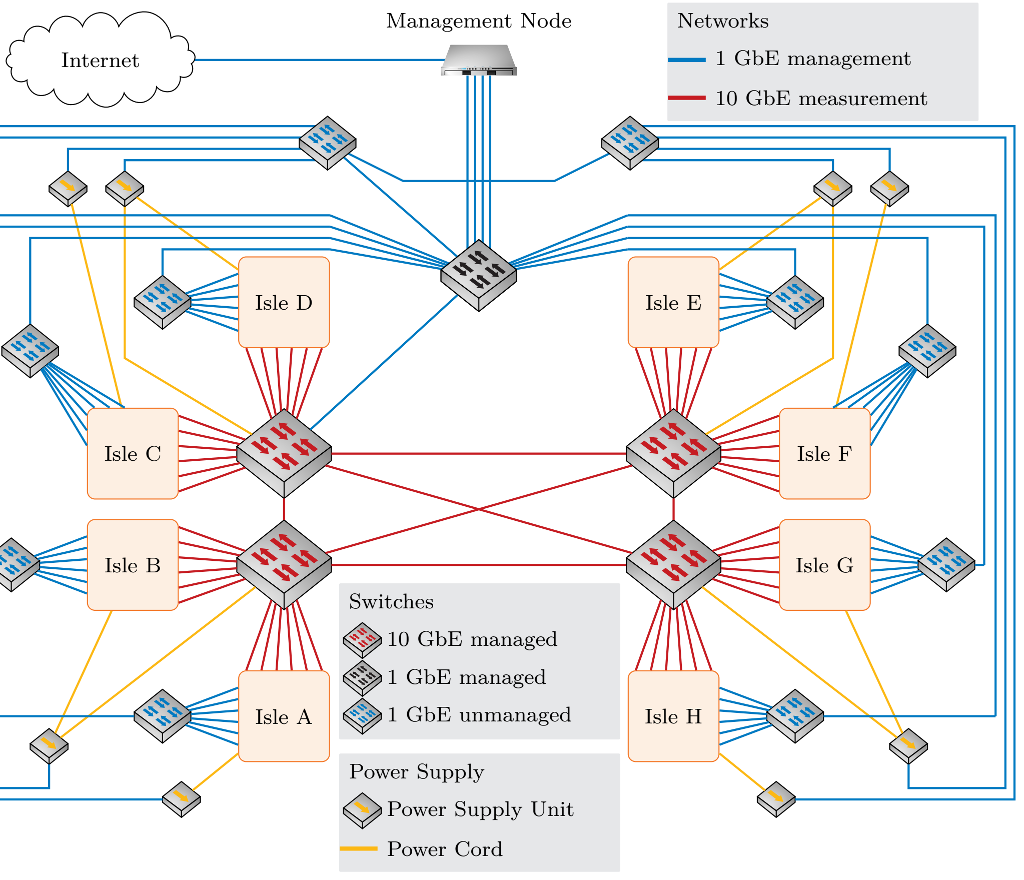
Infrastructure Overview: The lab room consists of eight identical isles. Each isle as shown in Figure 3.8a is set up to be used by a team consisting of two students on-site.

Per-Isle Workspace A single isle consists of six commercial COTS desktop nodes (**RQ_T1**), two 1 GbE switches and two Cisco 881 routers. These devices can be freely power-cycled and wired by the students using common RJ-45 Ethernet cables (**RQ_T2**). A group of three of the COTS nodes are connected via a keyboard-video-monitor switch to be used with one set of peripherals (**RQ_T4**). A central unmanaged switch connects the management port of every COTS device. Via this switch, the isle is connected to the labroom's management switch.

Management & Measurement Networks All isles are connected via the management switch to the management node, which provides Internet access as shown in Figure 3.8b. Within



(a) Per isle configuration. Six nodes for two on-site users.



(b) Complete lab room topology. Eight isles with independent management and measurement networks.

Figure 3.8: iLab infrastructure: wireable devices (green), and fixed management (blue) and measurement (red) networks.

the management network, the isles are separated using VLANs. This avoids traffic interference between isles (**RQ$_{\mathbf{T}}$6**). To allow usage as research testbed, one port of every COTS node is permanently connected via one of four managed 10 GbE switches. These switches are fully interconnected (**RQ$_{\mathbf{R}}$9**). While this reduces the number of usable ports for students, iLab teaching exercises are designed with only three ports in mind. Using independent hardware for 1 GbE management network and 10 GbE measurement networks has the advantage that one cannot influence the other.

Management Node: The management node manages all devices in the room. Among others, it provides uplink connectivity, an HTTP proxy ($\mathbf{RQ_{T}3}$), and remote access via ssh ($\mathbf{RQ_{R}8}$). It also provides services for the boot process, as well as for monitoring and remote control ($\mathbf{RQ_{R}8}$) of the infrastructure.

The monitoring capabilities provide information relevant when used as both, teaching and research testbed: students can inquire which isles are currently free to be used as spare isle, while researchers can monitor which nodes are currently available as measurement nodes.

The complete configuration of the management node and the network topology Ⓐ is automated using ansible. Its powerful templating engine allows to provide configuration to the COTS nodes, which only differs in details, e.g., the hostname or IP address prefix. This permits supervisors to easily setup a new infrastructure or change parts of an existing one.

A second configuration file is provided by the supervisor to create or update the live image Ⓑ used by the nodes.

Boot Process: A static installation of 48 is not feasible, as updating or even just a one-time installation is not scalable. Further, the users have physical access to the nodes with root privileges ($\mathbf{RQ_{T}2}$). Consequently, users can create erroneous configuration state by modifying files of the OS, which would persist through the boot of a static installation.

When a student team boots an isle ① or a testbed user remotely initiates the process Ⓡ, the node boots via the management network. We use Debian live images to boot a completely clean OS ($\mathbf{RQ_{T}7}$). While the initially downloaded image is stored in RAM, we also store the image on the node's SSD after erasing its previous content. This offers the possibility to boot the node with a clean image, but also select to boot the previous installation in case of an accidental reset or power outage ③. Each node also receives a templated script that configures its management interface and ssh for easy console access between the nodes of an isle. Further, the script disables all unnecessary services and autoconfiguration.

The image is an unmodified up-to-date Debian image that includes all tools used during the labs. The applications required during the labs are statically configured, for instance, the browser to use the management node as HTTP proxy for Internet access. We chose an open-source distribution commonly used for networking tasks to teach students the concepts of state-of-the-art networking on commodity hardware. None of the installed services like DHCP are started after the boot process. This is to provide a clean system that does not perform networking tasks in the background, unknown to the students.

To reduce the load on the management network, the image is only downloaded in one of two cases: the image stored on the node has been tampered with or a newer version of the image exists on the management node. On boot, a hash of the currently stored image is calculated and compared with the hash of the current image provided by the management node. Only if the hash differs, the image has to be downloaded. Further, we use a custom-built multicast provider to deliver the same image to all nodes in parallel. Unicast is used as fallback, to download missing parts and to receive the per-node configuration script. This saves bandwidth of the 1 GbE management network, allowing to boot all nodes simultaneously in the same time as booting a single node.

No nodes require any manual installation of the OS. Instead, only the BIOS, which is password protected to avoid tampering, has to be configured once to enable booting via the management network. Consequently, the infrastructure is scalable: additional nodes can be added or changes to the image can be deployed at any time without the need for upgrading the network or slowing down the existing management operations. Pre-installing all software used during the labs further reduces stress on the management network and uplink connection.

Operation as Research Testbed: The teaching infrastructure is not used continuously, as it is unoccupied during lecture free periods or at night. Even during the term, selected isles might be available at times. This downtime allows all 48 COTS devices to be used as research testbed for distributed experiments, CPU-intensive computation, e.g., for machine learning, or network device performance measurements.

As on-site execution of experiments is unpractical and often infeasible during the night, remote access and control capabilities are required ($\mathbf{RQ_R8}$). This is done by using the management node as central hub, allowing researchers to remotely control and access the nodes. pos is used as testbed management software and to guarantee that experiments do not interfere with teaching ($\mathbf{RQ_R10}$), e.g., by rebooting a node used by students. The life cycles and features of pos match the requirements of the iLab life cycles: multi-user temporal management of available nodes, controlling nodes, deploying experiment tools, running (scheduled) experiments, and centrally collecting measurement artifacts. For logistical reasons, pos is not directly running on the labroom's management node. Instead, the iLab nodes are integrated with another testbed, physically located in separate rooms, and controlled by that testbed's pos instance. pos uses the management node of the iLab labroom as proxy, i.e., pos uses respective proxy interfaces for configuration and initialization.

The nodes boot a separate research image, which does not include the pre-installed software used by teaching exercises. This image is also used to distinguish whether a node is being used for teaching or research. Before trying to boot a node for a research experiment using pos, pos checks the current state of the node. If the node is currently booted into the teaching image, it cannot be used for research. Only if the node is currently offline, i.e., unused, or already booted into the research image, pos is allowed to use the node. This protects against accidentally shutting down a node used for teaching. In contrast, physical access by students allows to always reboot the node into the teaching image, independent of the previous state.

Due to the static 10 GbE measurement network shown in Figure 3.8b, no manual cabling is required ($\mathbf{RQ_R9}$). Students cannot take part in research experiments as the respective port connected to the measurement network is only enabled by pos on-demand. Further, in the teaching image this statically cabled interface is not bound to any driver, i.e., it is hidden from students.

For remote control capabilities, the nodes are equipped with a mainboard that includes an out-of-band management engine, which are state-of-the-art for enterprise PCs. This interface is used by pos as initialization interface. For redundancy and for power cycling, other devices like the Cisco routers are plugged into remote controllable power supply units. These are also connected via the management network and can be controlled from the management node as shown in Figure 3.8b.

Remote Usage during Corona Pandemic

Due to the global Covid-19 pandemic all teaching had to be performed remotely for the summer and winter terms of 2020. This posed two separate challenges for iLab teaching exercises using the lab infrastructure and presented life cycles.

Challenge 1—Remote Access: Remote access is possible due to the requirement for the infrastructure to be used as research testbed. Instead of using access tokens to physically access the lab room, a process to collect and install an ssh public key per student on the management server was implemented. Furthermore, all isles were upgraded with remote controllable power supply units to also remotely control additional devices like the Cisco 881. This setup enables teamwork as the students can still use the nodes of an isle simultaneously.

However, remotely controlling the nodes proved more challenging than regular physical access. For instance, higher-level debugging functionalities that usually exist by looking at the graphical output of the device during the boot process either did not exist or could only be used with additional effort. Considering that the course is intended for students without prior knowledge of networking and Linux devices, the entry barrier was increased significantly.

Monitoring capabilities are vital for remote users to identify the status of the nodes, including error states, and allowing temporal division of the nodes between the teams.

Challenge 2—Network Experiments: An integral part of iLab exercises is to allow the students to create and change the physical network topology by plugging in Ethernet cables between the nodes. For obvious reasons, wiring the nodes is not possible when working remotely. Instead, every week the setup was statically wired by the supervisors. All teaching experimental instructions were updated to not include tasks that require re-cabling, e.g., to investigate the influence of a changing topology when using routing protocols. An exception were experiments where the cable could be virtualized using a Linux bridge. This allowed, for instance, simulating “unplugging” by shutting down the bridge interface. Instead of only one isle, each team received two isles for each lab so that two different topologies could be set up. This further increased the number of experiments that could be performed remotely.

Effects on Teaching: For the students, losing the ability to create the topologies and plugging in and out Ethernet cables by themselves is a huge loss, as it is a central part of computer networking basics. While it is a didactic loss, it also complicates the lab work and increases the difficulty for newcomers. An advantage is that students gained the ability to learn how to

Available for	Total	Teaching	Research
Linux Nodes	48	48	48
CPU cores	288	288	288
RAM	3.07 TB	3.07 TB	3.07 TB
10 GbE interfaces	192	144	48
Wireless interfaces	152	152	144
Cisco Routers	16	16	0

Table 3.2: iLab lab room hardware resources

work remotely, including how to configure and debug networks without physical access. We consider this a valuable skill in the field of networking, especially during a global pandemic and working from home. Further improvements are possible, e.g., by uploading detailed pictures of the physical setup for the students to provide them with a better feeling for the actual physical topology.

Second, the course is designed for teams of two working together, i.e., socially interacting with each other while using networking terminology. Further, interaction with other teams in the lab room is encouraged. This exchange is hardly the same when working remotely.

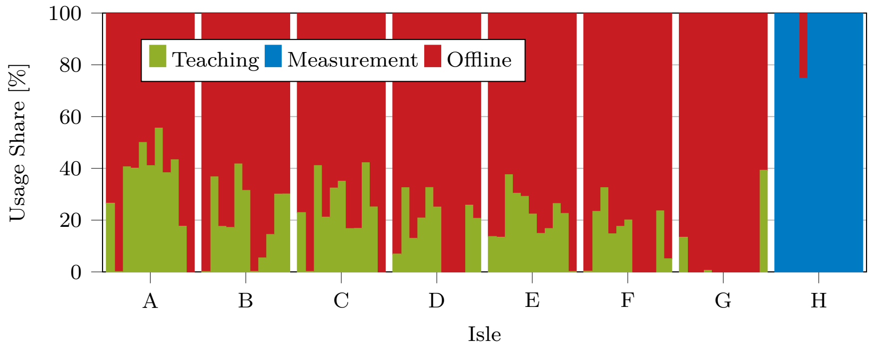
Our metadata evaluation also shows that some labs required significantly more time to complete for the students. In particular, labs that relied on tools with graphical outputs, e.g., using Wireshark or a web browser, were more challenging due to lag introduced by the amount of data being transferred. However, most problems could be solved by using slightly different workflows, e.g., remote capturing using ssh and Wireshark.

Evaluation

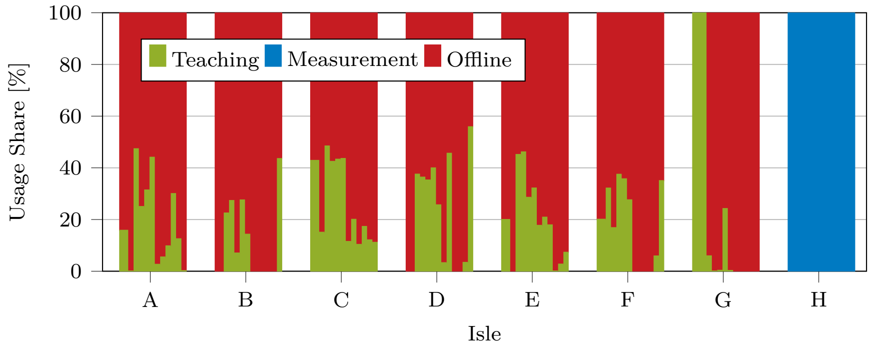
We evaluate the potency of the combined teaching and research testbed using different metrics.

Testbed Resources: An overview of the lab's hardware resources are shown in Table 3.2. With a total of 48 nodes, this results in an aggregated 3.07 TB of RAM, 288 CPU cores, 192 10 GbE interfaces and 152 wireless interfaces. In total, the 10 GbE interfaces in the static measurement setup combined are capable of producing 480 Gbit/s of traffic. As the teaching and the testbed infrastructure share the same nodes, all computing resources are shared, too. Only the 10 GbE interfaces are split: three per node are used for teaching, while one is permanently connected to the measurement infrastructure. Furthermore, an additional per-isle WiFi adapter is only plugged in on-demand for teaching experiments, therefore, cannot be used by the testbed environment by default.

Research Usage: The infrastructure in its current form or using previous hardware generations have been used for various research domains. The lab room hardware was used to evaluate secure multiparty computation using a star topology of up to 18 nodes [139] and to analyze the performance of distributed ledger technologies [13]. The nodes are also well suited for computational tasks, including large-scale analysis and evaluation of experiment data, or machine learning.



(a) Summer term 2020 (eleven weeks)



(b) Winter term 2020/2021 (twelve weeks)

Figure 3.9: Weekly isle usage during iLab1 course. Remote due to Covid-19 pandemic

Quantitative Analysis: As of March 2021, approximately 1500 students have taken part in iLab courses at TUM since 2008. Figure 3.9 shows the isle usage during the summer and winter terms of 2020, influenced by the Covid-19 pandemic. We distinguish usage for teaching and measurement experiments, as well as nodes being offline.

Due to remote work, each team received two isles, which are statically cabled by the supervisors at the beginning of the week, to allow more experiments with up to two setups, which usually would require re-cabling the nodes during the exercise. Isles A and B were used by a total of five dedicated teams, C and D by 4, and E and F by three teams. A team was allowed to use the pair of isles one complete day per week. Usage during the weekend was permitted on a first-come-first-serve basis. Isle G was reserved for supervisors for emergency testing and debugging purposes. Only isle H was allowed to be used as testbed measurement nodes to reduce interference with teaching during the pandemic.

A total of twelve teams participated during the eleven week summer term, while 15 teams participated in the winter period of twelve weeks, including public holidays. While we do not yet have monitoring data for a regular on-site term, we estimate that the occupancy for teaching is usually even higher. Up to 30 teams can use the lab room, each receiving one isle for a whole day. Further, roughly 50 % of the occupancy can be expected during the weekend and for the spare isle. Estimating eight hours per team per day, results in an occupancy of over 30 % per isle. Vice-versa, this shows that even during a full on-site term, the devices have at least 50 % downtime, which can be used for research purposes.

3.2.6 Conclusion

The pos testbed controller models the typical experiment life cycle. During every step, pos' experimental workflow enforces a structure that guarantees reproducible research. Thereby, it simplifies the execution and scaling of several individual measurement runs, required to test a network system with all possible combinations of inputs. Even after the conclusion of the measurements, pos provides means to evaluate the data, automatically generating graphical representations and models. pos increases the efficiency of the testbed, allowing multiple users to use the testbed resources in parallel. Using temporal allocations, it is guaranteed that an individual resource is only used within one experiment at the same time.

The case study, orchestrating a combined teaching and research testbed, has shown that pos can even be used in complex scenarios. Its modular design and differentiation between initialization and configuration interfaces allows to extend the orchestration software to manage additional heterogeneous network nodes. This can include additional requirements, like testbed usage must not interfere with teaching, or even physical separation between management and testbed nodes.

3.3 MoonGen as Dynamic Load Generator

Section 3.3 is based on a collaboration between Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [4].

MoonGen [33] is a high-performance software packet generation framework based on the libmoon/DPDK packet processing platform. Its abilities to perform precise latency measurements and generating 10 Gbit/s of variable traffic using COTS devices makes MoonGen an ideal candidate for replicable network device measurements. It is used throughout this work as load generator in various setups, generating different kinds of traffic depending on the concrete scenario. Although in these use cases MoonGen's performance and precision are of primary relevance, creating traffic based on a multitude of different protocols can become cumbersome when having to craft all packets manually from scratch. To make the generation of different packets using different protocol stacks simpler, flexible and extensible, we implemented a new protocol stack for libmoon, MoonGen's underlying packet processing framework.

3.3.1 Motivation

The continuous increase in performance raises the requirements for soft- and hardware networking devices. With the standardization of 40 and 100 GbE [140], devices have to be capable of managing multiples of this load. To test and verify that networked devices behave as expected in high-performance scenarios, packet generator and processing frameworks are required. In Section 2.1.1, we introduce several software packet processing frameworks that have been developed to meet these demands [25], [28]. They utilize techniques to circumvent traditional bottlenecks of networking applications. A common approach is to bypass the conventional network stack of the OS and perform all packet related operations themselves [28], [37]. While all required features, typically provided by the OS's network stack, have to be implemented from scratch, performance is gained as all unnecessary processing steps can be left out or replaced with optimized solutions. Furthermore, device-specific features of modern NICs are used to offload computationally complex tasks to the hardware [33]. The lack of a unified API across different vendors increases the difficulty using these offloading features.

However, low-level access to and processing of packets is the common functionality of packet processing and generation frameworks. Thereby, the protocol stack employs the protocol header definitions of the different levels of the ISO/OSI model. A packet is made up of a concrete protocol stack, i.e., a specific order of protocol headers. Via the protocol stack's API, modifications to the packet can be made. A wide spread of different protocols with unique usage scenarios and varying complexity and requirements makes the implementation of a proper protocol stack difficult. Combining these aspects with the fundamental requirement towards the packet processing framework—high performance—is challenging. Therefore, the protocol stack is often reduced to offer only basic functionality or compromises are made in regard to utility, flexibility, or extensibility.

libmoon's high-performance, flexible protocol stack is based on dynamic code generation and JIT compilation. We show that dynamically generating the protocol stack offers a high degree of utility without impacting the performance. Further, the protocol stack is modular, allowing to extend it with future protocols, only requiring minimal implementation effort. A well-defined DSL to define the protocol stack not only allows to easily create new stacks, but also to reuse stacks, making setups and results replicable. We verify that the protocol stack fulfills these aspects by applying it to a tunneling setup typically found in data centers. Implementations of high-level protocol semantics beyond header validations are beyond the scope of this work.

3.3.2 Survey of Protocol Stacks

The term protocol stack often refers to fully featured implementations of protocol semantics typically found in protocols of the ISO/OSI model. One example is the networking stack found in all OSes, featuring a large variety of supported protocols and, therefore, protocol stacks. High-performance packet processing frameworks typically use kernel-bypass methods and come without support for such a stack. Specialized user space stacks like mTCP [141] for DPDK exist that fill this gap. However, such implementations are out of scope for this work as we only discuss representing and addressing packet headers on a lower level than the mTCP framework.

Frameworks, tools, and applications for software packet processing at rates up to and beyond 10 GbE such as DPDK, PF_RING, or PFQ allow the user to modify packets in one way or another. The usability, flexibility, richness of features and maturity of the code basis of these protocol stacks differ depending on the project size and general aim of the application. Furthermore, as the primary goal usually is the achieved maximum performance, for instance, saturating a 10 GbE link—or more—with minimum sized packets and minimum effort in terms of CPU utilization, the protocol stack tends to be reduced to core utility functions, if not only raw byte manipulations of the packet. Our emphasis in the following survey is put on the protocol stack rather than the overall architecture of the framework.

Pktgen-DPDK: DPDK comes with data structure definitions and helper functions for common protocols that can be used to build stacks manually. Pktgen-DPDK [142], a packet generator, is an interesting case study for the representation of protocol stacks. The application is capable of achieving line-rate when generating minimum sized packets while allowing to modify, for instance, the Ethernet or IP address of consecutive packets. It supports generating traffic with common protocols such as UDP, TCP, ARP, and ICMP. Moreover, it can generate traffic encapsulated with the Generic Routing Encapsulation (GRE) tunneling protocol, a feature requiring a more sophisticated approach towards the protocol stack if implemented properly. The utility functions per protocol are reduced to filling the header based on a sequence object. This centralized object for the whole protocol stack contains a list of keywords reflecting certain fields like the IP source address field. In addition, keywords, that translate to specific protocol header fields or bit-masks like the IP source address, can be used to set the respective bits to a concrete value. Other header fields like the IP version are preset and hard-coded to fixed values and cannot be customized.

Moreover, Pktgen-DPDK offers a scripting interface which allows to define streams of packets during runtime. By defining the starting and ending address in combination with a delta, consecutive packets will be modified accordingly.

Generating a full protocol stack of, for instance, an ICMP Echo request is only possible by hard-coding the sequence of the respective headers. This is even more cumbersome for encapsulation protocols like GRE, as each type, based on either IP or Ethernet, is hard-coded separately. This approach does not scale well when more protocols are added.

PFQ: The PFQ [43] networking framework focuses on creating applications that make use of "in-kernel functional processing and packets steering across sockets/end-points" [143] using C, C++, Haskell, or their own *pfq-lang* DSL. It is able to transmit and capture packets at 10 GbE and higher line-rates using vanilla ixgbe drivers. The C++ interface offers no support for packet headers, e.g., the packet generator tool[1] creates ICMP packets by manually setting all bytes. The functional language allows to query, filter, and modify the properties of the packet, but is limited to simple stacks with monotonically increasing layers, i.e., tunneling protocols are not supported. It has to be noted that the focus of this application is on sending packets from pcap files.

PF_RING ZC: PF_RING ZC [41] is a network socket intended for capturing, filtering, and analyzing packets in high-performance environments. It utilizes techniques like Direct NIC Access or Zero Copy to speed up or completely circumvent the kernel processing. The sample

[1] `https://github.com/pfq/PFQ/blob/master/user/pfq-tools/pfq-gen.cpp`

packet generator application[1] defines its own header structures. These are then filled member by member, whereby all values are hand-crafted and no utility functions are used, for instance, for the calculation of the IP checksum. Furthermore, the offsets and header sizes have to be calculated manually when allocating the structures. However, PF_RING also supports to send packets from pcap files. The suggested work-flow is to generate the traffic as a pcap with another tool that allows for easy modification and crafting of packets and then sending it with PF_RING.

netmap: The netmap [144] kernel module is designed for fast, but also safe packet I/O using standard system calls. This approach makes it easy to modify existing applications based on raw socket or libpcap to work on top of netmap [37]. The sample packet generator application[2] shows that common C structures are used to build packets. No further utility functions are provided.

Ostinato: The multi-platform traffic generator and analyzer Ostinato [145] is different compared to the previously introduced frameworks as it focuses on providing a powerful graphical interface for the protocol stack to craft, modify and analyze packets. While Ostinato is able to directly send crafted packets, the primary goal is to create or edit pcap files which can then be replayed by a processing framework intended for high-performance. It therefore primarily focuses on the protocol stack and utility functions instead of performance, making it an interesting case study to formulate requirements regarding the usability and flexibility of a protocol stack. The tool supports all common network protocols, a variety of tunneling protocols and also higher level, text-based protocols like HTTP. Not only can these protocols be stacked in any order, but also every header field can be set via the GUI to user defined values. All fields are initially set to reasonable defaults, e.g., the IP length field depending on the packet length. The option to generate streams of packets with a continuously changing member value is also provided. This is a very useful implementation for a packet generator frontend, but unsuitable for general-purpose high-performance packet processing.

Tcpreplay: The Tcpreplay [146] suite focuses on replaying captured traffic. The typical work-flow looks as follows: a pcap file is loaded, modifications to the headers are made and finally the complete stream of packets is replayed.

The editing options via the tcpwrite tool are limited to operate on layers 2 to 4 of the ISO/OSI model. Addresses and ports can easily be rewritten or remapped, but also other protocol fields can be changed. For layer 5 and higher, only padding of the packets, in case these layers were truncated in the pcap file, is supported to create packets of correct length again. Because the focus is on modifying existing packets, a complete protocol stack is not implemented.

[1] `https://github.com/xtao/PF_RING/blob/master/userland/examples/pfsend.c`

[2] `https://github.com/luigirizzo/netmap/blob/master/apps/pkt-gen/pkt-gen.c`

Snabb: Snabb [147] is building on a user space driver to speed up packet processing. As with libmoon, it uses Lua as programming language in combination with LuaJIT to offer the user a simple, yet fast, scripting environment. Snabb provides a protocol stack called *lib.protocol*, which offers versatile utility functions to manipulate headers and craft packets. In general, an unknown packet can be parsed, which as a result returns a list of the headers representing the full protocol stack. Each header can then be manipulated using setter and getter functions for the fields. Furthermore, a full protocol stack even for a complex packet, e.g., for tunneling, can be crafted. However, because of performance issues, these utility functions cannot be used for performance-critical parts of the program [148].

Summary: None of the introduced processing frameworks combines the aspects of fast performance, flexibility, usability, and extensibility in one protocol stack. This is either because the tool is not focusing on packet modifications or a compromise between performance and utility is being done, favoring the former. Furthermore, the extensibility compared to the surveyed protocol stacks has to be improved. Adding new protocols to allow the creation of new protocol stacks should be possible without code duplication, meaning the full protocol stack should not be hard-coded every time to fit the current use-case.

3.3.3 libmoon Framework

libmoon is a framework for building packet processing applications in the scripting language Lua [33], [149]. It combines the user space packet processing framework DPDK [150] with Lua's JIT compiler LuaJIT [151]. Using a high-level language allows for short development cycles of packet processing applications. Lua code compiled with LuaJIT can be as fast as equivalent code written in C and can embed existing C code. This is achieved by using low-level C data structures instead of the default Lua data structures in all performance-critical paths. The disadvantage of this approach is that the usual safety features of a scripting language, e.g., memory-safety, are not available in these paths. However, the critical code is handled by the framework, not by the user application. Related work has shown that libmoon can be used to create applications for manifold networking tasks, including the packet generator MoonGen [4], [33].

The downside of working with libmoon's old, static protocol stack API is that packets have to be crafted by setting every byte manually. While this is fast, it is error-prone. Further, it is neither flexible nor reusable as, for instance, the programmer has to take care of correct byte order and calculating all offsets within the packet. For every new application, the complete protocol stack has to be crafted from scratch. This results in a large amount of duplicated code as protocols are typically reused and several protocol fields are set to the same standardized value.

The typical structure of a libmoon processing task is composed of two different phases: the setup phase and the actual runtime of libmoon. Especially for sending custom-crafted packets, this differentiation is important for the protocol stack.

Setup Phase

Before a task starts receiving or transmitting packets it has to prepare all data structures. This includes, for instance, parsing addresses or setting up statistics counters. Most importantly, it has to allocate memory space for packet buffers that should be send out. As this is based on the underlying *mempool* structure of DPDK, each packet from a specific pool will look the same, i.e., a packet template can be defined. Instead of initializing the packets with zeros, all bytes can be set at this stage to custom-defined values to mimic as much of the final packet as possible. This reflects that a processing task only sends a certain type of traffic, for instance, TCP packets. All of these packets have a vast percentage of their bytes in common, while only minor changes have to be made on a per-packet basis.

Finally, an array of the previously defined template packet buffers is allocated. This is done to allow batch processing of packets in the upcoming processing loop to increase the performance [33], [149].

The implication of this phase for the protocol stack is that the allocation of new packet buffers happens **before** the actual processing runtime of libmoon and, therefore, does not impact the performance. The elapsed time to fulfill these tasks is irrelevant for the final application. However, the API for configuration of packet templates should be easy to use and provide high usability.

libmoon Runtime

During runtime, the task performs the actual processing in terms of receiving, modifying and transmitting packets. All actions performed are critical in regard to the performance. No new buffers have to be allocated in memory, instead modifications are performed either on the received packet, or on a new template taken from a pre-allocated memory pool. Ideally, only slight modifications on a per-packet basis have to be done to further reduce CPU cycles spent per packet. This does not require utility functions that modify the complete stack, but rather target single operations. This is accomplished by iterating over the array of packet buffers and modifying every single packet individually. This may include, for instance, setting a different address or payload.

When all packets of the batch are have been processed, they are transmitted. Sending of packets is asynchronous, pointers to the packet buffers are placed into a queue and then accessed by the NIC [33], [149]. This implies that packet buffers must not be reused or even be modified after calling the send function, as they are only recycled by DPDK when the NIC actually sent them. Instead, for each iteration new, pre-filled template buffers have to be allocated from the *mempool* and then adjusted per packet.

3.3.4 libmoon's Dynamic Protocol Stack

The protocol stacks surveyed in Section 3.3.2 are static: the specific required stack has to be built by the programmer, in most cases from scratch. The advantage is that they can easily be optimized by an ahead-of-time compiler. However, the drawbacks are manifold, as becomes apparent when looking at modern tunneling protocols like VXLAN. Two different protocol stacks, one for each en- and decapsulated packets, are required. With an inflexible approach, both have to be hard-coded separately, yielding code duplication, although a large amount of structure and bytes, the decapsulated packet, is shared by both stacks. Further, the data of the encapsulated part does not have to be modified at all, as merely headers are prepended.

A related problem is that higher layer protocols in the ISO/OSI model can be based on different protocols underneath, a prominent example being IPv4 and IPv6 on the network layer. In most of the frameworks presented in Section 3.3.2, if the user wants to create a UDP packet once based on IPv4 and once on IPv6, both protocol stacks would have to be implemented separately, for all layers and from scratch. This duplication of code continues even further when adding new protocols, generating more and complex stacks that have many layers in common.

The requirements for the dynamic protocol stack of the libmoon framework are as follows.

Performance: The protocol stack must not reduce the processing performance of the framework, i.e., packets have to be processed at line-rate of 10 GbE or beyond. In other words, an equal performance level as if the task would have been implemented with low-level operations has to be achieved.

Flexibility: While granting the user full customizability to create even malformed packets, the protocol stack must be flexible to allow complex operations like packet en- or decapsulation with low computational overhead. Thereby, the user should not have to calculate offsets when building a complete stack out of individual protocol headers.

Usability: The API of the protocol stack must be easy to understand and intuitive to work with. It must offer functions providing utility to the user by obsoleting repetitive tasks. This includes data type and byte-order conversions, as well as generating complete, pre-filled, and legal packets per default.

Extensibility and Composability: The protocol stack must be modular, i.e., every protocol header is implemented by itself, allowing to easily add new protocols in the future. Individual headers must be combinable to complex stacks. At the same time, this process should be easy and automated, requiring only minimal developing effort whenever possible.

The survey of existing packet generators has shown that usually a trade-off between performance and usability is required. The following outlines the concept and architecture of libmoon's protocol stack, which not only unifies these contradictory points, but, moreover, fulfills all listed requirements.

Conceptual Overview

Not all functions need to be fast: functions used during the setup phase are allowed to be slow. Only the startup time of the application will be delayed, which is not a time-critical aspect. In turn, these functions offer as much utility as possible during this phase, generating templated packets with even only one line of code. During the actual runtime of the application, all called functions are critical to the performance, i.e., need to be fast.

The only actual implementation effort is defining the layout of a concrete protocol header, i.e., defining the syntactical structure of the header. However, this has to be done only once, i.e., for protocols or headers that libmoon does not yet support. The complete generation of a protocol stack is performed automatically and dynamically through JIT compilation on demand. Users can define their desired protocol stack by defining which header should appear in what order. This concept follows and is based on the ideas of the ISO/OSI model. Each protocol layer—ideally—is independent and self-contained from the next layer, wherefore each protocol header can be implemented separately. Within a certain stack, the header itself must contain information that reveals which data follows next.

Statically Implementing a New Header

libmoon's protocol stack requires only the minimal information about a protocol header, which has to be implemented once. Primarily, this includes the structure of the header, i.e., its members and their respective bit-sizes. Based on this information, utility functions for all members are generated. This process is automated and only requires manual code changes by the developer when using data types with additional semantics like IP addresses. Optionally, further semantic information can be specified manually. This includes determining the size of the header or the next following header, based on the data of the current header.

Header Structure: The format of a header is defined as if it was a C data structure object, consisting of the data types and names for each member. In fact, libmoon creates an actual C structure based on this information of the header using LuaJIT's FFI. Data types can be basic ones, or specifically created as is the case, for instance, for Ethernet or IP addresses. Members with variable size, e.g., TCP options, have to be implemented as variable-length array members. Furthermore, the name of such a member has to be marked as variably sized in the header's Lua object. This allows for special treatment within the complete protocol stack and to create concrete, fixed-sized instances of the header, depending on the current data of a packet.

Wrapper for Members: To prevent working with low-level data types, wrapper setter and getter functions for all members are created in Lua. These functions provide a first level of utility as data type conversions, correct byte order and other data type problems are taken care of. This process is automated for standard data types like integers. Only if the automatically generated utility functions are not sufficient, e.g., because of custom data types, the set and get function has to be manually defined by overwriting the generated one.

As these functions are merely thin wrappers in Lua they do not cause performance losses. As a result, these functions are fast while providing a minimum of usability. Therefore, the wrapper functions should be used during the actual runtime of libmoon to modify packets on a per-packet basis.

Wrapper for Header: libmoon also adds several templated functions that perform tasks on the complete header. Their purpose is twofold: firstly, functions that provide utility for the user in form of a set and get function for the complete header. In the case of the set function, the passed argument is a table of labels, each referencing a concrete member of this header.

Within the function, for each member the respective set function is called. This way, using one function call, the complete header is filled with default values, while each single member can be customized by the user. This function comes at the cost of a significant performance loss. Hence, this function belongs to the group of methods that should only be used before the runtime of libmoon to pre-allocate packet buffers when initializing a templated memory pool. At this abstraction level, performance is not the primary concern, instead, usability on a broader range becomes important.

The second set of functions is used to provide semantic information about the header in the context of a complete protocol stack: how to resolve the next following header, e.g., the EtherType of an Ethernet frame; how default parameters change, e.g., IPv4's protocol field based on the next header; how the size of the variable member is determined, e.g., the size of the IPv4 options field depending on the Internal Header Length member; or how the sub-protocol can be determined, e.g., a VLAN tagged Ethernet header or TCP options. If one or multiple of these apply to the protocol, the developer has to overwrite the automatically generated empty functions to provide the semantic information.

The end result of this is that a new protocol, its layout, and required semantic information, is defined. From now on, an instance of it can be used by libmoon within a complete protocol stack.

JIT-compiled Protocol Stack

The implemented protocols can be used by an application developer to generate arbitrary protocol stacks dynamically on demand. The implemented header information is sufficient to fully specify one layer of a protocol stack. For a concrete stack, merely the order of protocols within the required stack has to be defined. Internally, libmoon then performs multiple steps to dynamically offer complex utility functions for the whole stack.

Protocol Stack Definition: Creating a new protocol stack can be performed with one function call, passing as argument a list of protocols using the simple and intuitive embedded DSL defined in Listing 3.1.

```
<stack>    ::= <protocol> | <protocol>, <stack>
<protocol> ::= <header> | { <header> [,name = "<str>"] [,subType = "<str>"] [,length = <int>] }
<header>   ::= "eth" | "ip4" | "ip6" | "udp" | ...
```

Listing 3.1: DSL to define a protocol stack

In the simplest case, the DSL defines the order of required protocols, starting with the lowest layer. Syntactic sugar is added by allowing to specify a table instead, which allows for optional arguments to cope with special scenarios: in case a protocol is used multiple times within a stack it has to be uniquely labeled. Second, the subtype, e.g., Ethernet with or without a VLAN tag, can be specified. Lastly, for headers which can be variably sized, the length of the variably-sized member can be set.

A concrete example for a realistic and complex protocol stack that can be observed in a data center environment is illustrated in Listing 3.2. This stack consists of an Ethernet frame with

```
-- create protocol stack
local asVxlanStack = createStack(
    {"eth", subType="vlan"},
    "ip4",
    "udp",
    "vxlan",
    {"eth", name="innerEth"},
    {"ip4", name="innerIp4"},
    {"udp", name="innerUdp"},
    {"sflow", subType = "ip4"}
)
```

Listing 3.2: Creating a VXLAN protocol stack

VLAN tag and is used for VXLAN tunneling. The tunneled packet, consisting of another set of Ethernet, IPv4, and UDP headers prefixed with “inner”, is of type sFlow.

Creating a stack creates a function to cast a packet buffer to the desired stack.

Internal Data Structure: Based on the defined sequence of protocols using the DSL, internally a C structure for the full protocol stack is automatically generated. The data structure’s members are the C structures of the respective defined protocols. Depending on the subtype or length, concrete structures are dynamically defined and loaded once by the JIT compiler.

Casting a packet buffer to this structure allows to interpret the array of bytes as defined by the structure of the stack and its protocol members. The C object is extended with a Lua metatable to define utility functions for the complete protocol stack. This includes the per protocol implemented utility functions.

As final member of each protocol stack, a payload structure is added, which allows to access arbitrary bytes beyond the last specified header.

Generated Utility Functions: Functions operating on the complete protocol stack are twofold. First, setter and getter functions, operating on the complete stack are created. These functions use the per-protocol implemented functions, passing a table of labels referencing individual members. To uniquely reference members of different headers, which could appear multiple times within the stack, each label is built by prefixing the member name with the header name. An example based on the previously introduced VXLAN example is given in Listing 3.3.

```
pkt:fill{
    -- member of the outer Ethernet header
    ethSrc      = "01:02:03:04:05:06",
    -- member of the inner Ethernet header
    innerEthSrc = "0a:0b:0c:0d:0e:0f"
}
```

Listing 3.3: Referencing stack members

Default values for undefined labels are intelligently set using the logic implemented per header. This can be based, for instance, on the next following header, e.g., for the EtherType of an Ethernet frame, or the accumulated length of the preceding headers, e.g., for the IPv4 length member. Automatically setting these default values is possible as the necessary information is available at the abstraction level of the complete stack. These setter and getter functions allow to easily create legitimate packets, or parse and recursively resolve even completely unknown packets as far as possible. These functions, however, are slow because of the use of Lua tables or recursive operations, while providing high utility, and should only be used to generate templates or for debugging.

The second group of functions are performance-critical functions to be used during actual runtime. This includes the calculation of checksums or setting all attributes that depend on the size of the complete packet. In this case, the function is optimized for performance by loading it dynamically through JIT compilation during runtime. Whenever possible, the offloading features of the NIC should be utilized to gain performance.

Variably Sized Headers: Headers with variable size complicate the generation of a complete stack, as the C structures of the respective protocol cannot be increased on demand. Using an undefined size results in wrong alignment of the following header. Therefore, the only solution is to create a completely new stack for each different size. While this increases the number of generated stacks, it has no negative impact on the application as recasting the packet buffer to another stack costs virtually no performance. Furthermore, the new stack is comprised of the same utility functions, adjusted slightly to accommodate the new size.

Creating a new protocol stack is a CPU-intensive task because of internal processing overhead. However, for most applications, the protocol stacks of interest are known in advance and can, therefore, be generated during the setup phase. Even during runtime, creating a new protocol stack is a one-time performance penalty.

Summary

libmoon's protocol stack offers functions to dynamically create new packet types and functions for a particular protocol stack. Because libmoon uses DPDK's *mempool* structure, functions are either slow and provide usability, or optimized for performance, but only fulfill one certain task. Overall the API offers functionality on five different levels of abstraction as shown in Listing 3.4: the low-level DPDK structure (1), which is fast, but has low utility; the low-level protocol stack structure (2), which is like (1), but takes care of offsets within the protocol stack; the protocol member wrappers (3), which are like (2), but add support for data types; the full header wrappers (4), which are slow, but allow to automatically set all members of the header with default or custom values; and the full stack wrappers (5), which are like (4), but perform everything for the full stack with intelligent default values.

```
-- Setting the source IP address of an IP packet
local ip = parseIPAddress("10.0.0.5")

-- create protocol stack
local getIP4Packet = packetCreate("eth", "ip4")

mbuf.pkt.data[25] = 5           (1)
mbuf.pkt.data[26] = 0
mbuf.pkt.data[27] = 0
mbuf.pkt.data[28] = 10

-- Cast to IP4 protocol stack
local pkt = mbuf:getIP4Packet()

pkt.ip4.src.uint8[0] = 5        (2)
pkt.ip4.src.uint8[1] = 0
pkt.ip4.src.uint8[2] = 0
pkt.ip4.src.uint8[3] = 10

pkt.ip4.setSrc(ip)              (3)
--                                              fast
-------------------------------------------------------
--                                              slow
pkt.ip4:fill{ ip4Src=ip }       (4)

pkt:fill{ ip4Src=ip }           (5)
```

Listing 3.4: Summary of libmoon's protocol stack API

3.3.5 Performance Evaluation

We use measurements to verify that the JIT-compiled protocol stack retains the required performance level of libmoon. As sample application we en- and decapsulate packets using the VXLAN protocol. The DuT is equipped with an Intel Xeon E3-1230 v2 at 3.3 GHz and 8 MB L3 cache and an 82599ES 10-Gigabit SFI/SFP+ NIC. A second, directly connected host is generating and receiving traffic using the MoonGen [33] packet generator.

On the load generator and sink the transmitted and received packets are counted to calculate the throughput and packet rate of the DuT. All measurements were run for up to 60 s to obtain reliable results. All graphs plot the mean value with error bars showing the standard deviation interval where applicable as calculated by the statistics module of MoonGen. Error bars are omitted if they would be smaller than the mark indicating the average.

Encapsulation Sample Script

The DuT is used as Virtual Tunnel Endpoint (VTEP), encapsulating incoming packets by prepending an Ethernet, IPv4, UDP, and VXLAN header. This requires flexible handling of multiple protocol stacks, as well as utility functions to fill in the header information. The full libmoon script implementing this scenario is accessible online[1], an excerpt is shown in Listing 3.5 and discussed in the following.

Before processing packets, the required protocol stacks are defined and a template of the encapsulating headers is created. This is done with one function call during which all members that have to be modified are set using their respective labels.

During runtime, every single received packet is looked at as raw data. Only its total size is required to copy all data as payload to the created encapsulating protocol stack. The remaining tasks are to update the size of the buffer and setting the length members of the IPv4 and UDP headers. All used functions are dynamically generated, yet optimized for the concrete protocol stack. As last step, the NIC is instructed to calculate the checksums before transmitting the packet.

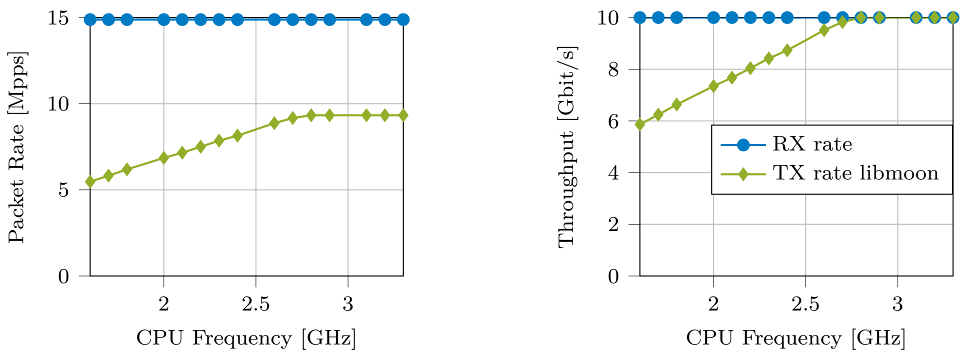
[1] https://github.com/emmericp/MoonGen/blob/master/examples/vxlan-example.lua

```
-- create stack
local asRawPacket = createStack()
local asVxlanPacket = createStack("eth","ip4","udp","vxlan")
-- creation of packet template
local mem = memory.createMemPool(function(buf)
    asVxlanPacket(buf):fill{
        -- define the VXLAN tunnel
        ethSrc="aa:bb:cc:dd:ee:ff",
        ethDst="00:11:22:33:44:55",
        ip4Src="192.0.2.1",
        ip4Dst="192.0.2.254",
        vxlanVNI=1234,
    }
end)
local txBufs = mem:bufArray()
-- [...]
while libmoon.running() do
    local rx = rxQ:tryRecv(rxBufs, 0)
    txBufs:allocN(rx)
    for i = 1, rx do
        -- cast to generic packet
        local rxPkt = asRawPacket(rxBufs[i])
        local size = rxBufs[i]:getSize()
        -- cast tx template to VXLAN packet
        local txPkt = asVxlanPacket(txBufs[i])

        -- copy rx raw payload to tx packet payload
        ffi.copy(txPkt.payload, rxPkt.payload, size)

        -- add length of added headers to size
        local totalSize = 46 + size
        -- adjust buffer size
        txBufs[i]:setSize(totalSize)
        -- set the IP/UDP length members
        txPkt:setLength(totalSize)
    end
    -- offload checksums and send
    txBufs:offloadChecksums()
    txQ:send(txBufs)
end
```

Listing 3.5: Excerpt from the VXLAN encapsulation task using libmoon's protocol stack



(a) Packet Rate

(b) Throughput

Figure 3.10: Peak performance when encapsulating Ethernet frames using VXLAN

Performing these operations without utility functions is error prone and cumbersome. It requires knowledge about the header offsets within the complete stack to copy the data to the correct position. While the copy operation itself remains the same, setting the bytes of the encapsulating headers, including addresses, ports, and length members, byte by byte is time consuming, prone to errors and requires more lines of code.

Encapsulation Performance Comparison

This first measurement analyzes the maximum throughput and packet rate when encapsulating minimum sized 64 B packets at line-rate. Both, packet rate and throughput, are displayed in Figure 3.10. Because of the nature of the VXLAN scenario, the size of received and transmitted packets differ.

libmoon is able to process all incoming packets already at a CPU frequency of 2.8 GHz, even when using the dynamically generated utility functions. The transmitted packets are encapsulated and therefore larger, resulting in not reaching the maximum packet rate of 14.88 Mpps, as the link capacity of 10 Gbit/s is reached first. Before this point the throughput and packet rate increase linearly with the configured frequency.

Influence of Memory Locality Effects

Figure 3.11 illustrates the achieved packet rate at a CPU frequency of 1.6 GHz when encapsulating packets of different size. The only difference in the encapsulation program is the amount of data that has to be copied to the TX buffer.

Figure 3.11 shows that the location of the read or modified data is important. The performance does not decline linearly with increasing packet size and, therefore, number of copied bytes. Instead, multiple continuous performance levels can be identified. Within one level, the maximum achieved rate decreases only marginally by about 0.1 Mpps. The levels have a size of 64 B, indicating a correlation with the cache line size. Packet data is an array of sequential bytes, which the dynamically generated protocol stack only casts to a different structure, retaining memory locality. A performance loss is only noticeable when accessing data in a different cache

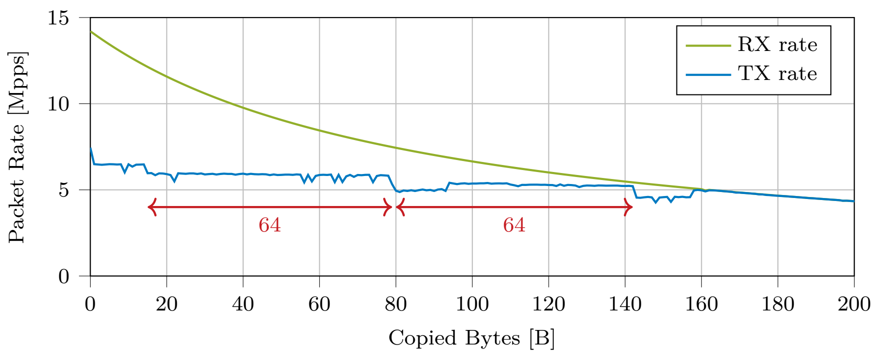


Figure 3.11: Performance when copying different amounts of consecutive bytes. Blue arrows indicate the identified performance levels, correlating to the CPU's cache line size of 64 B.

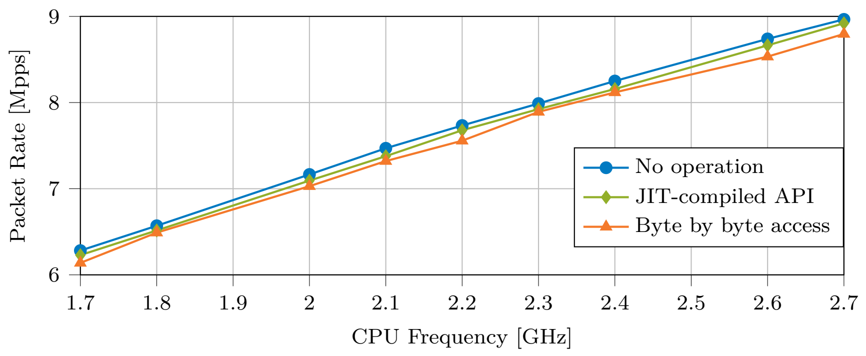


Figure 3.12: Performance when using low-level DPDK structures versus libmoon's protocol stack

line, generating an additional cache miss. Actual data manipulation within that line only costs cycles depending on the performed operation.

Comparison with Low-level Byte Access

The JIT-compiled utility functions are optimized and produce no new performance overhead. This experiment compares the performance when performing a typical operation on the packet using the JIT-compiled utility functions, to setting the bytes of the packet manually. The concrete task is to set the source IP address of the packet.

Figure 3.12 shows the results for different CPU frequencies when sending minimum sized packets at 10 GbE line-rate. The total performance loss with less than 0.2 Mpps for this operation can be explained with the results of the previous section. In the concrete example, the JIT-compiled functions yield better results compared to manually setting bytes.

3.3.6 Conclusion

libmoon's protocol stack combines requirements that seem mutually exclusive at first. It offers a dynamic, flexible, and extensible API, while maintaining the performance of direct low-level byte operations. High-level utility functions to perform packet modifications provide usability and flexibility, but retain acceptable performance through JIT compilation. The API of the protocol stack in combination with the Lua scripting language lowers hurdles for developers creating new software-based networking applications. The library of existing protocols in libmoon's protocol stack can be extended with low effort: the structure and low-level semantics of each protocol header is implemented completely separately, ensuring modularity. Utility functions are generated automatically and extend to the full stack. New headers are available immediately to be used in a custom made protocol stack using a DSL. The DSL allows the creation of complex stacks with multiple layers, including tunneling and other encapsulating protocols.

Compared to the state-of-the-art, a wide-ranging set of utility functions without the need for tedious manual implementation, is automatically generated. In fact, the JIT compiler is able to optimize the dynamically generated code such that even complex operations are performed as if manual low-level byte by byte manipulations were used instead. While utility functions for setting the full header or protocol stack are slow, these are primarily intended to be used during libmoon's setup phase to create packet templates where performance is not critical

The proposed protocol stack also aids the replication of results: a full protocol stack can easily be described using libmoon's protocol stack DSL and recreated by others. The protocol header syntax is implemented for several protocols that are commonly used. As libmoon and its packet generation framework MoonGen are software packet processing and generation frameworks available for COTS hardware systems, these tools can be used for research in various networking domains without the need for specialized hardware.

3.4 Key Results

This chapter introduced the tools used throughout this thesis. Emphasis is put on creating and conducting reproducible experiments. For this, the pos testbed controller was developed, which not only enables, but enforces a reproducible experimental workflow. All experiment artifacts are collected to fully define the experiment, allowing future reproduction of the results. Repeating a measurement with slightly adjusted parameters, commonly used to analyze the effect of these parameters on the DuT, is directly supported using pos' loops. Furthermore, pos fulfills the requirements for managing multi-user testbeds consisting of heterogeneous software and hardware devices. Importantly, all node resources can be used by each user using temporal division. Time conflicts, i.e., scheduling multiple experiments on the same node, are prohibited by pos to prevent interferences. Full automation, even towards publishing experiment results, is supported. The support for heterogeneous devices and a broad range of users is also vital for pos to be used for several different testbeds, including the combined teaching and measurement iLab testbed.

A second important tool used throughout this work to achieve reproducible results is the MoonGen load generator. We have extended MoonGen and its underlying libmoon framework with a flexible high-performance protocol stack. Freely and quickly generating traffic consisting of different scenario-specific protocol stacks is a powerful tool for performance evaluations of different network devices. Furthermore, the dynamic protocol stack, built using a classic packet processing framework like DPDK, serves as a point of comparison with modern approaches for data plane programming in the next chapters.

3.5 Statement on Author's Contributions

Sections 3.2.1, 3.2.3, 3.2.4, and 3.3 are based on a collaboration between Sebastian Gallenmüller, Dominik Scholz[1], Henning Stubbe, and Georg Carle [10]; and a joint work between Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle [4]. The author significantly contributed to the concepts of pos and implemented all components of pos. Mandelstamm is based on an interdisciplinary project by Alexander Kurtz, co-supervised and integrated with pos by the author. The in-depth explanations of the experimental process (Section 3.1), testbed life cycles (Section 3.2.2), and pos including its evaluation (Section 3.2.4), in particular Figures 3.1, 3.3, 3.4, and 3.6, were created for this thesis. Excerpts from the original publication, e.g. pos' limitations, were modified to fit the research questions of this work. Compared to the original publications, the author extended the descriptions of technical aspects of pos and added the statistical evaluation regarding deployments and performance.

The discussion about the hybrid teaching and research testbed presented in Section 3.2.5 was created for this thesis. Based on the addition of these Sections and Figures, the analysis goes significantly beyond the scope of the original publications. vpos was developed as part of a Bachelor's Thesis by Jonatan Juhas, which the author co-supervised and helped integrating with pos.

The author significantly contributed to the concepts of libmoon's protocol stack and provided its implementation. The survey of protocol stacks in Section 3.3.2 and the explanations for the architecture and implementation of the protocol stack in Section 3.3.4 were created for this work. The author performed the measurements presented in Section 3.3.5.

[1] Joint first authorship with Sebastian Gallenmüller

Chapter 4

Performance Modeling of Programmable Network Devices

We use and extend the idea of a high-level view of a generic packet processing system s proposed by Gallenmüller [152]. This system is a single node within a larger network, whereby incoming packets are processed, resulting in them being discarded or one or more potentially modified packets are being sent out. When deployed, the system is subjected to a vector of input parameters $\vec{I}$, for instance, the bandwidth and packet rate of incoming packets, traffic patterns, or packet sizes. Through the processing behavior of the system, the system creates a vector of output parameters $\vec{O}$. These are measurable and can be quantified using KPIs like throughput, processed packet rate, latency, or jitter, as defined in Section 3.1. We use the mathematical notation shown in Equation 4.1 throughout this work to define this generic packet processing system s, also referred to as DuT:

$$s(\vec{I}) \mapsto \vec{O} \tag{4.1}$$

Often, no or only limited knowledge about the internal processing of a packet processing system is available, resulting in a black-box view. The reason is, that such a system is typically the result of a symbiosis between hardware and software. While software might be accessible through open-source access, fully understanding the hardware requires full documentation or its blueprints. For purely software or CPU-based systems, like the Linux network stack, this might be possible, however, for hardware targets, e.g., ASICs, information is typically limited as companies do not release the necessary information.

In reality, such packet processing systems are complex, which becomes obvious when inspecting them as white-box as shown in Figure 4.1. In combination with all possible input parameters, it becomes infeasible to accurately measure and characterize all possible output parameters, i.e., fully understand the system's processing behavior. The cost, measured, for instance, in time or money, to investigate the system, becomes unsustainable. Furthermore, not all output parameters might be of interest.

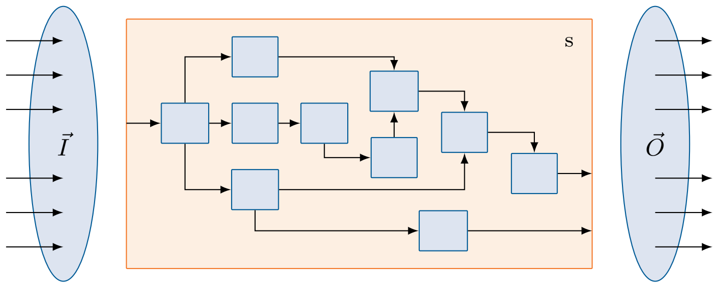


Figure 4.1: White-box view of a generic packet processing system s

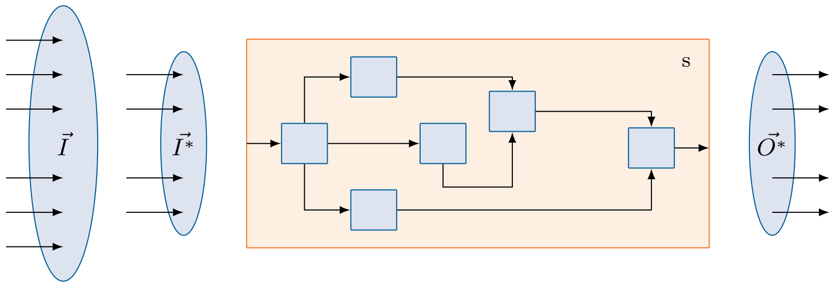


Figure 4.2: Model for a packet processing system s

To combat this complexity, instead of a full system description, we want to derive a model of its behavior as shown in Figure 4.2. This means, that the system is exposed to a selected subset $\vec{I^*}$ of input parameters and a subset $\vec{O^*}$ of output parameters is measured. The system behavior is then described by a modeling function m as defined in Equation 4.2:

$$m(\vec{I^*}) \mapsto \vec{O^*} \tag{4.2}$$

4.1 Modeling Approach

Sections 4.1.2 and 4.1.5 are based on a collaboration between Dominik Scholz, Hasanin Harkous, Sebastian Gallenmüller, Henning Stubbe, Max Helm, Benedikt Jaeger, Nemanja Deric, Endri Goshi, Zikai Zhou, Wolfgang Kellerer, and Georg Carle [14].

In the following, we outline our proposed modeling approach. In this work, we limit our efforts to fully programmable data planes, in particular, such that are programmed using the P4 DSL.

4.1.1 Components of P4 Data Planes

While there is a plethora of different P4 architecture models like PISA, SimpleSumeSwitch, or PSA, they share the same basic set of stages, including parser, processing pipeline, and deparser [83]. Within each segment, it is well defined what operations can be performed. For instance, headers are parsed, added or removed in the parser and deparser stages, while match-action tables are applied in the processing pipeline. Furthermore, modifications to headers and metadata can be performed and target-dependent externs can be called. Simplified, we define a

P4 program P_4 as the sum of its components C_i:

$$P_4 = \sum_i C_i = C_{\text{pars}} + C_{\text{pipe}} + C_{\text{field\_mod}} + C_{\text{ext}} + C_{\text{depars}} \tag{4.3}$$

Each of the components can be further specified. For instance, the field modification component is the sum of all header field and metadata field modifications:

$$C_{\text{field\_mod}} = \sum C_{\text{head\_mod}} + \sum C_{\text{meta\_mod}} \tag{4.4}$$

These individual components can then have additional characteristics, like the size of the field or the performed operation. Another example is the pipeline component, consisting of all applied match-action tables of the program:

$$C_{\text{pipe}} = \sum_{i=1}^{n} C^i_{\text{mat}}(k, e) \tag{4.5}$$

Thereby, each table is defined by the number, type, and size of match keys k, and the number and size of table entries e.

For the sake of simplicity, we treat the parser and deparser stages as one combined stage. The extern component is target-dependent, i.e., what externs are available and how they are implemented differs between targets. We discuss further details about what features of components we identified and modeled in Section 4.4.

This description is not a complete reflection of the actual program, but only a model of selected components. Similarly to the above approach, all other components can be further specified and described by their respective features. While doing so increases the accuracy of the program description, it also increases the complexity of the program model.

4.1.2 Deriving Parameterized Component Models

The advantage of using a standardized, high-level packet processing language like P4, is that it can be analyzed using similar tools as for other standard programming languages. While this includes static verification or bug-hunting approaches, we focus on tools for generating CFGs. Given a P4 program, such a tool generates and visualizes the control flow, i.e., all possible processing paths, of the program. Therefore, for a data plane program, all possible paths through data plane components to process incoming packets are derived. Mapping this representation of one program onto a programmable packet processing system enables a partial white-box view of it. While the exact hardware or software processing steps are unclear, the system must be composed of and, therefore, process the components C_i identified in the CFG. These components represent the complete program defined in Equation 4.3. While mapping one concrete program to the data plane might not cover all processing capabilities of the target system as the CFG only represents a subset of the complete system, the components important for common P4 programs are represented.

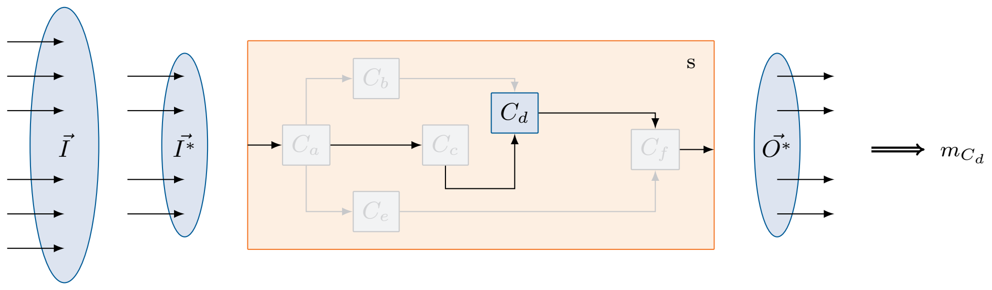


Figure 4.3: Modeling of an individual P4 data plane component

As it is infeasible to individually model the performance of all programs for all target devices, we instead propose modeling individual components identified by the CFG of a P4 program. For this, we use short, synthetic programs targeting individual components, i.e., P4 language constructs, similar to the modularized approach introduced by Dang et al. [83]. The main idea is to evaluate the behavior of different components in P4 programs behave on a target system using a bottom-up approach. Focusing on an behavioral analysis of small building blocks in P4 programs serves multiple purposes. First, these experiments can be used as regression tests by developers of the compilers. Second, the measurements allow application developers to gain a fundamental understanding for the cost of P4 language constructs in software or hardware data planes. This is required as the impact of one component on performance and resource metrics can change drastically based on the concrete target. For instance, an LPM functionality can be implemented using different algorithms, which can map to different hardware building blocks. As a consequence, this influences not only the performance but also the required resources. A model for the impact of components on a concrete target can be used to estimate the cost of the complete application by extrapolating and adding up the costs of individual components.

We derive these lumped component models using measurements for individual components. Each measurement only contains a single language feature in addition to a baseline program, i.e., the minimum P4 program to forward a packet. Throughout a measurement series, the program includes the feature 1, 2, ..., n times. The goal is to understand and model the impact introduced by this component, e.g., how the latency changes when including this construct x times in a program.

However, the problem of program complexity remains. Depending on the depth of the program model shown in Equation 4.3, the number of components that need to be modeled increases. Therefore, we do not aim for a full model of the program, i.e., model all features of all components to the last detail. Instead, we want to identify those components, that have a high influence on the performance for the given architecture or target device. For instance, simple arithmetic operations are assumed to be of low cost on most platforms, compared to performing a lookup in a match-action table.

4.1.3 Path-specific Performance Model

After having modeled individual components, we can then use these component models to model the performance of entire paths through the data plane. For this, we again use the CFG of the

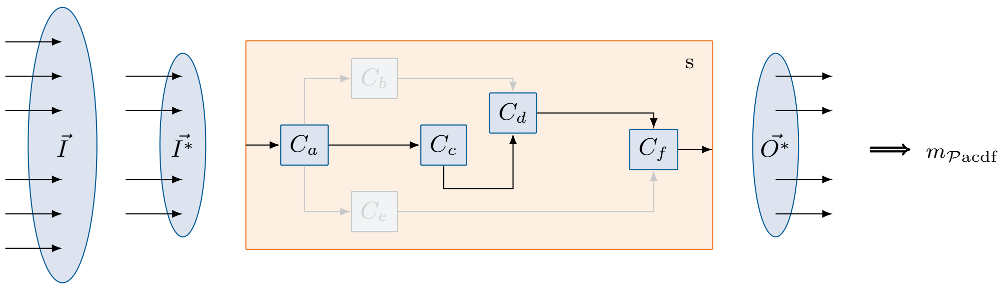


Figure 4.4: Modeling of an individual P4 data plane path

program to determine all paths and the number of occurrences of each component per path. To model the cost for a specific path $\mathcal{P}_x$ as shown in Figure 4.4 we then apply the program model defined in Equation 4.3: we sum up the costs as determined by our component models m_{C_i} of all components C_i found on the path $\mathcal{P}_x$ as shown in Equation 4.6:

$$m_{\mathcal{P}_x} = \sum_{C_i \in \mathcal{P}_x} m_{C_i} \tag{4.6}$$

However, there are metric-specific exceptions. For instance, to calculate the required resources for a program, calculating the cost for individual paths is not helpful. Instead, we add all occurrences of all components for the entire program:

$$m_{P_4}^{\text{resources}} = \sum_{C_i \in P_4} m_{C_i} \tag{4.7}$$

We choose the approach of adding up the individual component models due to its simplicity. Alternative approaches include using, for instance, network calculus or probabilities for the calculation of composed models [15].

4.1.4 Per-packet Performance

After computing path-specific performance models for all paths of a program we can compare and order the paths by cost, e.g., to determine the worst-case path through the P4 program. For each path we are interested in constructing respective packets that, when received by the DuT, will traverse the program using the selected path. Generating, for instance, only traffic that will match the path of highest cost will result in a worst-case evaluation of the DuT. Similarly, best-case and average-case studies can be performed.

While generating all possible paths based on a P4 program's CFG is trivial, deriving the packets that match a certain path, however, is not. Although a P4 program defines the syntax of the data plane, its semantic is provided by the control plane in the form of, for instance, match-action table entries. Only based on the concrete entries it is clear which match keys, i.e., packet data, will result in which path being taken as result of the applied match-action table. As we are only interested in the performance of the data plane, we are not analyzing P4 control planes.

We leave this task for related and future work. To demonstrate per-packet performance, we assume full knowledge about the state of the device, including control plane information.

4.1.5 Model-first Approach

The primary use case for the outlined model is to predict the performance of individual data plane components and paths. Thereby, the derived model's priority is not to be semantically correct, i.e., explain the behavior of the device. Instead, the model should mathematically reflect the observed behavior. This information obtained using a model-first approach—independent of why the device behaved in a certain way—can then be used for performance prediction. Furthermore, the obtained model can be compared with the theoretically expected model. If a difference is observed, e.g., exponential instead of linear growth for a certain component, the actual cause—the semantics—can be further investigated. Consequently, the derived models can be used for regression testing and to uncover and inspect unexpected behavior.

The proposed model-first approach employs a high degree of automation. As result of the measurement and evaluation of an individual component, the derived model is returned. To obtain this mathematically optimized model automatically, we use error metrics that quantize how accurately the model fits the measured data. However, a common problem of such derived models is that more complex models usually result in a smaller error. The more free parameters the model has, the better the fit typically is when using only such an error function. A solution for this are information criteria that include the number of free parameters or even the number of measurement points in the error metric, i.e., penalize complex models [153]. As a consequence the behavior is modeled using a simpler function at the cost of a higher mathematical error and a slightly worse fit. However, typically, device behavior can and should be explained using simpler functions. Only then, the model can be understood, increasing its value.

Depending on the metric that should be modeled, we use black- or white-box measurements. In general, metrics based on black-box measurements are simpler to obtain as they can be observed simply based on the input and output vectors of the DuT. This passive approach also reduces the performance impact caused by the measurement itself. In contrast, white-box measurements require access to and are performed on the DuT, impacting the accuracy of the obtained performance results.

4.2 Related Work

We surveyed two areas of related work: approaches for generating or using P4 CFGs for the analysis of program paths; and performance evaluations and modeling approaches for programmable data planes.

4.2.1 P4 Control Flow Graphs

The official p4c reference compiler compiles P4 programs to the JSON format. This intermediate representation contains all components of the program and how they are linked together, i.e., the program's control flow. p4c also has a backend for generating visual representations of the control flow, called p4c-graphs[1].

An attractive property of P4 is its design that fosters the simplified verification of program behavior, e.g., by the absence of loops in P4. Liu et al. [154] and Neves et al. [155] demonstrate the verification of P4 programs using asserts to identify bugs in applications. Nötzli et al. have proposed p4pktgen [106], a tool that generates packets and match-action table entries, such that all paths of the P4 program will be traversed. Their approach can be used to verify that the program behavior is as expected on a target platform. The authors state that generating all possible paths is not a trivial task. Programs may be complex through a vast number of applied match-action tables or the usage of state that can be changed by processed packets. Therefore, Nötzli et al. refer to falling back to branch coverage instead of full path coverage. Further, externs represent an unknown factor as the respective processing and result of an extern is not known from the data plane program. p4pktgen is currently limited to the bmv2 architecture model and does not yet support all standard P4 language statements. Packets that match a certain path are generated using a satisfiability modulo theories solver. Our model of P4 data planes extends the analysis of program paths beyond coverage to performance predictions. [106]

Kodeswaran et al. [104] use Ball-Larus encoding to track the execution paths of packets through P4 data planes. Due to program complexity, they argue to inspect disjoint parts of the data plane individually. Their prototype for the Intel Tofino target transforms the P4 CFG into separate, smaller CFGs, which are then modified to add tracking capabilities. This process works even for complex programs like the `switch.p4` with approximately 10^{34} different execution paths. Paths taken by processed packets are encoded for further analysis with low overhead. In contrast to our approach, Kodeswaran et al. track program paths at runtime. [104]

Liveness testing of networks, i.e., testing the status of all possible links in the network, is desirable for modern complex ISP or data center networks. Zeng et al. [156] have shown that this is possible, for instance, for the Stanford university network. However, they report limitations of their approach, in particular, devices that behave in non-deterministic ways or keep per-packet state [156]. Similarly, using the control flow of P4 programs allows to perform liveness testing of the data plane, covering all possible paths. Our modeling approach attributes performance predictions for those paths.

4.2.2 Performance Analysis & Modeling of P4 Devices

Performance reported in related work often depends on the target platform, measurement methodology, processing complexity, and testbed setup. Therefore, we focus on the baseline performance of different programmable data planes, either as reported by the respective authors or through performance studies testing multiple devices within the same setup. Further, we discuss proposals for performance improvements and efforts to model the performance behavior of such devices.

[1] `https://github.com/p4lang/p4c/tree/main/backends/graphs`

Target-specific Baseline: As the Mininet-based bmv2 target is only intended as reference implementation it is neither suited nor optimized for high-performance applications [66]. Consequently, we disregard this target in our performance analysis.

Vörös et al. analyze the performance of t4p4s as L2 and L3 router and in a load balancing scenario [69]. Using Mellanox and Intel NICs between 1 and 100 Gbit/s, t4p4s has shown linear scaling with the number of CPU cores and constantly outperforms OvS or PISCES using the same setup. Adding complexity to the processing pipeline significantly reduces the overall throughput of t4p4s, e.g., 7 Mpps for checksum computation in some scenarios. Using 1024 B packets, Vörös et al. show that t4p4s can process up to 67 Gbit/s of packets in the L2 switch example. The authors claim that in this case the limit is not the Ethernet bandwidth of the 100 Gbit/s NIC, but rather the bandwidth of the PCIe interface. While the authors also include an evaluation of Freescale hardware, they do not include latency values or analyze individual P4 program components. [69]

Osinski et al. evaluate the performance overhead of their proposed extension for OvS, called P4rt-OVS [77]. In three simple scenarios they compare complex processing pipelines to basic OvS, each including one or more of using a match-action table, adding a header, or modifying a header field. In these cases, the authors claim a similar performance between P4rt-OVS and OvS. In a second set of measurements, Osinski et al. analyze individual P4 components. Parser and deparser are analyzed for an increasing number of headers. However, as the added headers themselves are heterogeneous, limited conclusions about the overhead caused by adding other headers can be made. In their tests, the deparser, which also includes the cost for modifying header fields, is less costly in terms of CPU cycles required compared to the parser for an increasing number of headers. Regarding match-action tables, CPU cycles increase approximately linearly with the number of tables, while lookups and updates of tables with up to 10000 table entries are constant. We reproduce this behavior using the DPDK-based t4p4s target. In the last scenario, programs implemented using P4 and C are compared. The results show, that equivalent programs in C can be significantly more performant due to optimizations and reduced overhead. We provide a similar comparison and conclusion when comparing a P4-programmed to a manually implemented application, both based on the DPDK platform, in Section 5.3.7. [77]

The NFP-4000 is available for 10, 40, or 100 Gbit/s Ethernet ports [45]. The NIC is capable of line-rate processing for sample P4 programs, however, no further performance analysis for complex programs or individual components is provided by the vendor.

The NetFPGA SUME when programmed with P4 using the P4→NetFPGA workflow can reach 10 Gbit/s line-rate for simple programs, however, Ibanez et al. provide no further performance analysis [72]. Wang et al. show that the latency of P4FPGA is below 0.5 μs for baseline programs, compared to PISCES with more than 6 μs [71].

Harkous et al. evaluated t4p4s, the NFP-4000, and the NetFPGA SUME using a setup consisting of two directly connected nodes. The end-to-end latency of these targets for a baseline scenario under equal circumstances is below 45 μs, 8 μs, and 4 μs, respectively [157]. Our measurements presented in Section 5.3.7 confirm these performance levels.

The Intel Tofino is available in variants of up to 64 ports with each 100 Gbit/s, or up to 32 ports with each 400 Gbit/s for the Tofino 2 as of April 2021. These chips allow operation at line-rate per port independent of the P4 program complexity [74]. Latency depends on the processing complexity, as each additional stage required within the P4 pipeline adds latency. This, however, is limited by the number of stages available and is typically below 1 μs [158].

Geyer et al. [100] investigate P4 in the context of avionic applications. They benchmark P4 implementations of Avionics Full-Duplex Switched Ethernet (AFDX) on different targets. Their investigation shows that latency differs between target platforms, but the latency is comparable to existing dedicated AFDX hardware.

Performance Improvements: Several publications implement a certain functionality on one or more P4 targets and provide a performance analysis. However, as these always try to highlight the performance of the implemented application, these are hardly usable for a discussion of target-specific performance. Instead, in this section we summarize related work that specifically tries to improve general performance aspects of P4 applications and targets.

Laki et al. extend t4p4s to allow the asynchronous execution of externs [70]. Their evaluation, subjecting a given percentage of the overall traffic to an extern that requires a specific number of CPU cycles, reveals a trade-off when comparing synchronous and asynchronous execution. Which mode of operation is superior in terms of maximum throughput processed, depends on the complexity of the extern. In their setup, only for externs exceeding approximately 2000 CPU cycles, asynchronous execution achieves better performance. However, asynchronous execution requires buffering of packets and context between processing units. This is limited in size, causing overflows, i.e., no asynchronous execution of the extern, for externs requiring more than 2000 CPU cycles and when subjecting more than 50 % of the traffic to the extern. [70]

FlexMesh [159] improves performance of various network functions running in parallel on programmable data planes by automatically chaining them together. The author's framework does so for various targets, reducing overhead and redundancies. They show that their proposal can improve throughput and latency. The authors provide absolute throughput and latency values for the NFP-4000 and Tofino platforms in different scenarios. Using a baseline P4 program, Zhou et al. confirm that these platforms can process packets at the respective line-rates of 10 and 100 Gbit/s, while latency is lower for the ASIC-based target (below 5 ms) by a factor of seven. Using the P4 recirculation feature, the throughput of the SmartNIC remains stable for up to three recirculations, while the throughput of the Tofino is roughly halved with every additional recirculation. Latency increases linearly, but steeper for the SmartNIC. [159]

MATReduce [160] follows a similar goal of cutting redundant operations in P4 programs by combining and reducing duplicate match invocations. Instead of using the same match key multiple times in different tables, they combine these tables, consequently also reducing the number of matches performed. The authors evaluate their framework for bmv2 and the NetFPGA SUME. The FPGA was able to reach line-rate in all scenarios. The authors report an end-to-end latency below 1.4 ms in their complex scenario consisting of a firewall, access control list, network address translation, and routing. Thereby, MATReduce can yield improvements up to 45 % compared to regular P4 implementations. [160]

Bressana et al. enhance programmable ASICs with FPGAs for the use case of generating storage fingerprints [158]. They propose this heterogeneous hardware design to overcome the shortcomings of both ASIC and FPGA-based platforms. However, this comes at the cost of increased latency and the potential for the lower throughput FPGA to throttle the ASIC. For the scenario of storage fingerprinting the authors demonstrate a latency of approximately 5.5 μs using the proposed architecture. [158]

Modeling of Data Plane Components: Breaking down the processing of a network device into different components and evaluating and modeling individual components separately is no new approach. For instance, Rotsos et al. [161] have used this approach in their OFLOPS benchmarking suite for OpenFlow switches. They focus on inserting and benchmarking the behavior of the switch with different OpenFlow rules. The results show that the performance of OpenFlow switches depends heavily on the hardware or software implementation. This approach to benchmark individual components has also been proposed for P4 by Dang et al. They present the WhipperSnapper benchmarking suite [83] to evaluate the performance of a wide range of P4 target devices. Whippersnapper is split into target-independent benchmarks, i.e., benchmarks executed for all target platforms available on the test system, and target-specific benchmarks, i.e., benchmarks executed only for a specific target platform. They divide the P4$_{14}$ language into the five components parser, processing, including match-action tables and checksum calculation, state access through registers, packet modifications like adding and deleting headers, and action complexity composed of writing fields and expressions. Depending on the target, they focus on different metrics that have to be investigated: throughput and latency for software and NPU platforms; and resource usage for FPGAs and ASICs. The authors perform an abstract evaluation of selected components of the bmv2, PISCES, and P4FPGA targets. We extended this approach for the modeling of components and apply it to the P4$_{16}$ specification of the language.

Harkous et al. [84] propose a method for estimating the packet processing latency as a function of the configured P4 program when running on different P4 targets. They measure the latency cost of basic P4 constructs when running on the NFP-4000 SmartNIC. The authors note no measurable impact of modifying one or all headers fields using Ethernet, IPv4, or UDP traffic. Similar, binary operations have no significant impact on the latency, while arithmetic operations show an increase up to 1.9 μs for 25 operations. Parsing up to three headers has no impact on latency, however, when modifying, for instance, the complete UDP header, an increase up to 3.3 μs is noted. Further, Harkous et al. state that neither this increase is additive, nor the number of parsed and modified headers is independent in terms of resulting latency. Lastly, they show that the latency increase for an additional number of match-action tables can be approximated with a polynomial of second degree, whereby ten additional tables yield approximately 5.5 μs increased latency. Based on their measurements, Harkous et al. propose a model based on the number of parsed headers, modified headers, and tables. Further, they provide a lookup table for one to three parsed or modified headers and their polynomial approximation for the number of table entries. Their validation shows an error up to 187 ns. [84] In comparison, our approach for the evaluation and modeling of these components is independent of existing or future protocols. Further, our framework can use arbitrary metrics for evaluation.

Harkous et al. extended their work to include t4p4s and the NetFPGA SUME target platforms [157]. They note target-specific influences. For instance, increasing the number of parsed, modified, copied, or removed headers has a measurable impact for the NFP-4000. However, only the former operation has a noticeable impact for the NetFPGA SUME. The t4p4s framework shows no noticeable changes for any of those actions applied up to ten times. However, for 256 B packets, starting with four match-action tables, the DPDK-based target experiences packet loss. For all other targets and packet sizes, latency increases linearly. For each of the targets the authors determine a vector including the estimated coefficient of the respective approximated slope for each tested component. Based on the CFG of the P4 program, Harkous et al. determine the number of occurrences of each of the features within the program. The latency estimation for a program is given as sum of the baseline latency plus the product of the target-specific vector times the occurrence of each feature within the program. The validation shows a measurement error below 1 μs. We provide further insights into the processing pipeline of the t4p4s target, in particular, that header modifications do have a performance influence for this target. [157]

Lukács et al. propose a probabilistic model of the program execution to calculate the expected cost for a given control flow graph [162], [163]. The authors explicitly decouple the analysis of the control flow from the requirement of black-box testing. Only based on the P4 source code and information about the execution environment, i.e, target platform and implementation, the worst-case and the expected cost when executing the program can be statically determined. Thereby, this cost is a result of the sum of the cost of all possible execution paths times their respective execution probability. In their model the authors consider that execution conditions depend on program state and might not be independent. Through incremental refinement, constantly modeling more parts of the program and adding additional target-dependent information, Lukács et al. plan to improve the accuracy of the model. They apply their model to LPM, comparing a trivial linear search algorithm to DIR-24-8 used by DPDK. In a case study, they show that their model can predict the effects of increasing cache sizes on the execution cost of performing a single LPM lookup depending on the probability that the prefix is longer than 24 bit. This confirms the properties of the DIR-24-8, risking performance penalties for longer prefixes, as it assumes that the majority of prefixes is 24 or less bits. Our evaluation confirms these findings for the DIR-24-8 LPM data structure and extends the modeling for exact and ternary match types. [162], [163]

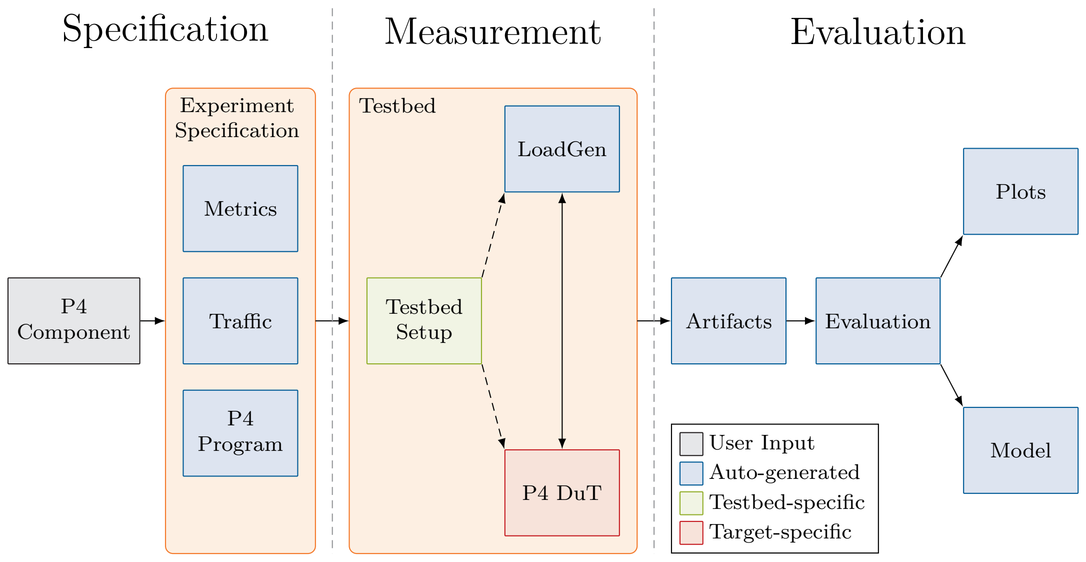


Figure 4.5: Automated measurement framework for data plane component modeling

4.3 Measurement Framework for Automated Component Modeling

Section 4.3 is based on a collaboration between Dominik Scholz, Hasanin Harkous, Sebastian Gallenmüller, Henning Stubbe, Max Helm, Benedikt Jaeger, Nemanja Deric, Endri Goshi, Zikai Zhou, Wolfgang Kellerer, and Georg Carle [14].

P4-programmable devices come in a wide variety regarding their underlying hardware architecture, such as CPU-based systems or ASICs as representatives of both ends of the spectrum. CPU-based P4 target platforms offer limited performance but are easily extensible. In comparison, ASIC target platforms have dedicated P4 processing pipelines with limited programmability but offer optimized performance in terms of throughput and latency. To obtain and parameterize the component model presented in Section 4.1.2 for target platforms with such fundamental differences, we propose a framework for the automated measurement, evaluation, and model-deriving for P4 data plane components. The framework displayed in Figure 4.5 is available as open-source [16] and operates in three stages: for the chosen component, an experiment specification is generated, the measurements are executed, and the generated artifacts are evaluated to derive the component model.

4.3.1 Component Experiment Specification

The goal of the framework is to model different metrics, e.g., throughput, latency, or resources, of programmable data planes with entirely different properties, e.g., software- or ASIC-based. For the sake of reproducibility, each experiment analyzing a single P4 component or feature is defined by a threefold specification.

First, the framework specifies which performance metrics are of interest for the evaluation of the device behavior. Per default, measurements include throughput, packet rate, and latency metrics. If supported by the target device, the resource consumption of the P4 program complexity is measured. Furthermore, internal target-specific parameters like CPU cycle usage or cache misses can be evaluated. However, not every metric is interesting for every target device [6]. For instance, resource consumption is of little interest for software-based targets, as the overall resources are virtually unlimited. Vice-versa, throughput and packet rate metrics are less interesting, e.g., for the Intel Tofino that guarantees line-rate independent of the P4 program. Not including irrelevant metrics for a certain target is crucial to limit the search space, reducing experiment complexity and measurement completion time. The specification of performance metrics is vital for all other components, including the load generator, P4 program, and the evaluation.

The second part of the specification defines the traffic that the DuT is subjected to, i.e., the traffic generated by the load generator. This includes the targeted throughput and packet size. Further, the specification defines the headers and payload of each packet. In particular, whether certain bytes of the generated packets have to be changed throughout a measurement series. This is required to generate traffic, for instance, matching different entries of match-action tables.

Lastly, the framework defines the parameters of the P4 program. As we are interested in evaluating individual program components independent of protocols, we define a baseline program. This is the minimal program required to actually forward a packet from the incoming port to the outgoing port. Based on the baseline program, we increase program complexity only for the respective component that we want to analyze. For instance, to evaluate the impact of the number of table entries for a given match-action table on the performance, the specification defines the range for the different numbers of table entries that should be tested. The specification contains the number of occurrences for every component and further details, e.g., how many bits each parser state parses. As a result, the components of the whole P4 program are specified such that an actual P4 program for a specific target can be generated.

4.3.2 Measurement Execution

Actually running a measurement poses two challenges: First, each testbed is different, be it the hardware of the management node, testbed nodes, or the testbed controller software. Therefore, the testbed setup component of the framework requires a testbed-specific implementation. This component is responsible for starting and synchronizing individual measurement runs between different nodes and gathering all artifacts. For this work, we implemented this component for testbeds using pos presented in Section 3.2.

Although P4 programs are intended to be portable, i.e., target-agnostic, in theory, target-specific knowledge is still required. This includes, for instance, the P4 architecture model that the P4 program needs to adhere to, supported extern interfaces, and the control plane interface required for loading the program and inserting match-action table entries. Therefore, the automatically generated P4 program specification needs to be translated to an actual P4 program fitting the P4 target. For the purpose of this work, we have implemented this target-specific component for the DPDK-based t4p4s target. Further, the testbed-specific component needs to implement means to manage the concrete P4 device.

The MoonGen-based load generator component can be automatically generated based on the experiment specification. The resulting MoonGen script generates traffic with the properties set by the traffic specification. By default, two sets of measurements are performed. First, the maximum throughput and packet rate that the DuT can process without packet loss are determined. For this, the DuT is subjected to traffic at line-rate. Based on the resulting maximum packet rate we measure the device's latency in the second phase. We subject the target to low, middle, and high traffic load, representing 10 %, 50 %, and 70 % of the maximum packet rate, respectively.

4.3.3 Artifact Evaluation

As the configuration and execution of the load generator is fully automated, all metrics obtained from this source can be evaluated, modeled, and visualized automatically. Only data obtained from the DuT requires a target-specific processing implementation.

We describe our packet processing systems and the model we want to derive using Equations 4.1 and 4.2. To automatically derive the model for an individual component, we use the fully automated experiments of the framework to obtain measurement data g that defines the behavior of this component:

$$M(x) = g \tag{4.8}$$

with $x \in G$ defining the measurement domain. Based on g, we want to derive a model as defined in Equation 4.2 represented by a modeling function M with an error metric H that quantizes the quality of M:

$$m : (M, H) \tag{4.9}$$

Automated Model Derivation

To derive a model for the measurement data g for the whole or a part of the measurement domain $X \subseteq G$, we use curve fitting applying the non-linear least squares Levenberg-Marquardt algorithm [164]. We use the *curve_fit* algorithm of the python scipy module[1], which works as follows: for a given function prototype τ with free fitting parameters $\vec{p^*}$ the algorithm tries to determine $\vec{p}$ to fit the given measurement data using τ as close as possible. The result is the parameterized fitting parameter vector $\vec{p}$. An example for such a function prototype with three fitting parameters is the polynomial of degree two:

$$\tau(x) = p_1^* x^2 + p_2^* x + p_3^* \tag{4.10}$$

To improve the quality of automatically generated models, we use a set Λ^* of different function templates λ. Each template is defined by the function prototype τ for curve fitting and an

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html`

associated function rank ψ that will be explained later:

$$\lambda = (\tau, \psi) \tag{4.11}$$

As function prototypes we use polynomials of degrees zero to five, exponential functions, logarithmic functions, and the inverse of all mentioned functions, i.e., $\tau_{inv} = \frac{p_0^*}{\tau}$. For every function template λ from the set of function templates Λ^*, the curve fitting algorithm is applied to solve for the fitting parameters $\vec{p}$ using the least linear squares approach. The result is the set of possible solutions Λ for the given measurement data:

$$\Lambda := \{\text{curve\_fit}(\lambda) = (\tau, \psi, \vec{p}, \eta) \bigm| \forall \lambda \in \Lambda^*\} \tag{4.12}$$

Model Quality Quantization: We use an error metric η to quantify the quality of each calculated fitting in Λ. Thereby, the experiment specification defines which metric should be used. Currently, the framework supports the mean absolute percentage error (MAPE) [153] and the symmetric MAPE (sMAPE) [153] metrics for regular measurement data. However, other metrics can be added as plugins.

For MAPE we calculate the absolute percentage error between every measurement and fitting point and then use the combined mean:

$$\eta(x)^{\text{MAPE}} = \left| \frac{g(x) - \tau(x)}{g(x)} \right| \tag{4.13}$$

$$\eta^{\text{MAPE}} = \frac{\sum_{x \in X} \eta(x)}{|X|} \tag{4.14}$$

However, MAPE has drawbacks, e.g., it is sensitive to outliers or artifacts in the measurement data [153]. SMAPE improves on the issues of MAPE, wherefore, we use the variation defined in Equation 4.15 as default error metric for the remainder of this work:

$$\eta^{\text{sMAPE}} = \sum_{x \in X} \frac{|\tau(x) - g(x)|}{|g(x)| + |\tau(x)|} \tag{4.15}$$

MAPE, sMAPE, and other metrics, are vulnerable to overfitting, resulting in complex functions being preferred for fitting. This becomes clear when looking at measurement data for a clearly linear dependency as shown in Figure 4.6. The resulting parameters for fitting a polynomial of first (τ^1) and fourth degree (τ^2) are shown in Table 4.1. Although the polynomial of first degree visually matches the measurement data, the higher degree polynomial has a lower error and would be preferred according to the sMAPE-based error metric η. Thereby, $|p_0^2|$ is extremely small and even $|p_1^2|$ and $|p_2^2|$ are below 0.1, i.e., the respective high degree polynomial factors are of low relevance for the overall model. While it is mathematically correct to choose the higher degree polynomial, semantically a polynomial of degree one is desired to fit the linear

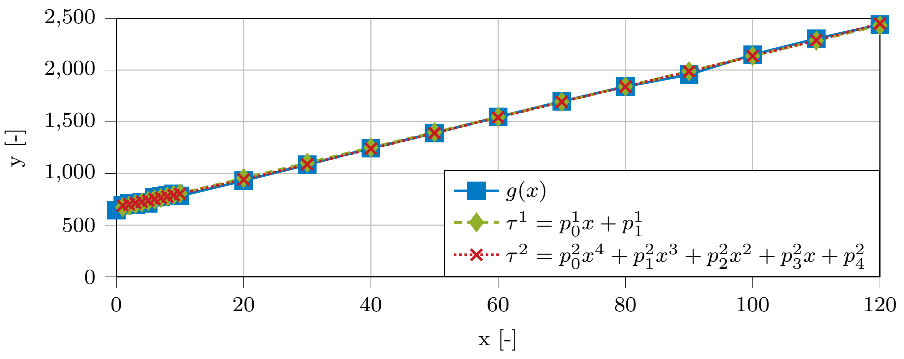


Figure 4.6: Overfitting for a clearly linear dependency

	p_0	p_1	p_2	p_3	p_4	η
τ^1	14.77241	659.44034	-	-	-	1.418%
τ^2	$\|p_0\| < 10^{-5}$	-0.00106	0.09374	11.73365	676.72004	1.196%

Table 4.1: Parameterized fittings showing overfitting

dependency. Therefore, we counteract this behavior using two independent strategies to improve the calculated error metric η.

First, we forbid small fitting parameters by defining an absolute minimum value γ for the fitting parameters. All fitting parameters p returned by the curve fitting algorithm are processed according to Equation 4.16:

$$p' = \begin{cases} \gamma, & \text{if } |p| < \gamma, p > 0 \\ -\gamma, & \text{if } |p| < \gamma, p < 0 \\ p, & \text{otherwise} \end{cases} \tag{4.16}$$

The error metric is then calculated, e.g., using Equation 4.15 for sMAPE, based on the capped fitting parameters p'. We argue, that limiting the granularity of the fitting parameters is warranted and justified as it also reflects the limited measurement accuracy.

The second strategy is based on the Akaike information criterion (AIC) [153]: we assign ranks ψ to every function prototype, whereby the rank is equal to the number of fitting parameters:

$$\psi = |\vec{p^*}| \tag{4.17}$$

If the difference in error metric for two fittings is between a certain margin ω, we choose the simpler function. ω is based on the minimum of both fitting errors multiplied by a margin factor κ_{rel}. As this would have close to no effect for already small errors, we also define an absolute minimum margin κ. For two fittings for the same domain F_1 and F_2, if $|\eta_1 - \eta_2| \leq \omega$, we choose the fitting with lower rank ψ:

$$\tau_{\text{chosen}} = \begin{cases} \tau_1, & |\eta_1 - \eta_2| \le \omega, \psi_1 < \psi_2 \\ \tau_2, & |\eta_1 - \eta_2| \le \omega, \psi_1 \ge \psi_2 \\ \tau_1, & \eta_1 < \eta_2 \\ \tau_2, & \text{otherwise} \end{cases} \tag{4.18}$$

with

$$\omega = max(min(\eta_1, \eta_2) \cdot \kappa_{\text{rel}}, \kappa) \tag{4.19}$$

We consider an error metric increased by up to $\kappa = 10\,\%$ acceptable to prioritize simpler model functions. We do not directly use the AIC formula as initial measurements have shown that this metric is too aggressive in preferring simpler functions in some scenarios. As this is a common point of critique for AIC, we chose our adjustable approach.

sMAPE in combination with our AIC-based function ranks is the default model quality metric. However, the framework is designed to support other metrics like MAPE that can be used as plugins instead. For measurement data that should be modeled using probability distributions, e.g., gaussian or trapezoid, other metrics, like the earth mover's distance (EMD) [165], can easily be integrated.

Resulting Model: The result of this process is a function prototype τ in combination with the calculated fitting parameters $\vec{p}$ being the best solution according to the error metric η and function rank ψ out of all calculated fittings Λ. The function models a part of or the full measurement domain, i.e., $X := \{x \in G \mid \alpha \le x < \beta\}$. We denote this chosen fitting as shown in Equation 4.20:

$$F = (\lambda, \vec{p}, \alpha, \beta, \eta) \tag{4.20}$$

For simple systems, this fitting, including the adjustable parameters κ, κ_{rel}, and γ, represents the complete model:

$$m : (F, \eta; \kappa, \kappa_{\text{rel}}, \gamma) \tag{4.21}$$

Multiple Partial Fittings

The behavior of complex systems cannot be modeled using only the function prototypes available in Λ^*. Events like overloading the system or exceeding the capacity of memory caches can drastically alter the performance behavior. Therefore, both sides, before and after the event, should be modeled independently. An example for such a case is shown in Figure 4.7. The device drastically changes its behavior between $x = 350$ and $x = 512$. In this case, the best single fitting F_1^1 still has an error of $\eta = 34.07\%$. Clearly, this fitting can not accurately model the entire measurement domain.

To reflect this, we split the measurement domain G into $n \in \{2, 3, 4, 5, 6\}$ separate domains and use the above outlined approach for every individual segment. The resulting combined fitting

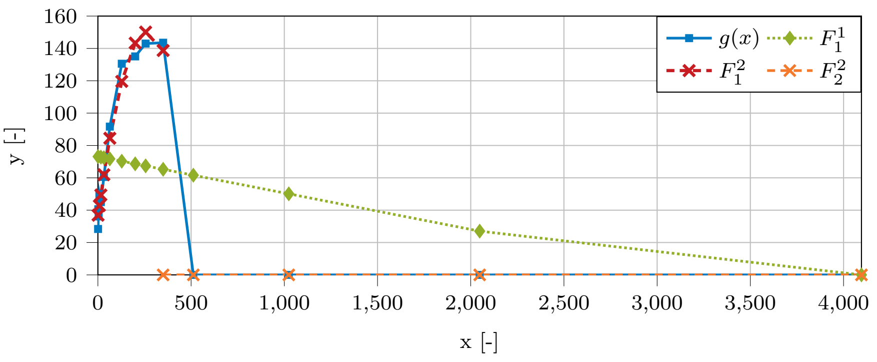


Figure 4.7: Drastic change in device behavior requiring two individual fittings

$\mathcal{F}$ is denoted as shown in Equation 4.22:

$$\mathcal{F}(x) = \begin{cases} F_1^n(x), & s_0 \leq x < s_1 \\ F_2^n(x), & s_1 \leq x < s_2 \\ \quad \vdots & \\ F_n^n(x), & s_{n-1} \leq x \leq s_n \end{cases} \tag{4.22}$$

The combined fitting $\mathcal{F}$ consists of n individual fittings F_i^n:

$$F_i^n = (\lambda_i, \vec{p}_i, s_{i-1}, s_i, \eta_i) \tag{4.23}$$

These fittings are divided by $n+1$ splitting points $\vec{s}$ from the set of possible splitting points:

$$\vec{s} \in S_n \tag{4.24}$$

s_0 and s_n denote the lower and upper bound of the measurement domain G, respectively. The error θ for the combined fitting is a weighted sum of the individual fitting errors η_i:

$$\theta = \frac{\sum_1^n |X_i| \cdot \eta_i}{|G|} \tag{4.25}$$

Similarly, the rank Φ of the combined fitting is a weighed sum of the individual function ranks:

$$\Phi = \frac{\sum_1^n |X_i| \cdot \psi_i}{|G|} \tag{4.26}$$

To determine the best combined fitting, we again apply the AIC-based metric of Equation 4.18 to always compare two combined fittings $\mathcal{F}_1$ and $\mathcal{F}_2$, however, this time using θ and Φ:

$$\mathcal{F}_{\text{chosen}} = \begin{cases} \mathcal{F}_1, & |\theta_1 - \theta_2| \leq \omega_s, \Phi_1 < \Phi_2 \\ \mathcal{F}_2, & |\theta_1 - \theta_2| \leq \omega_s, \Phi_1 \geq \Phi_2 \\ \mathcal{F}_1, & \theta_1 < \theta_2 \\ \mathcal{F}_2, & \text{otherwise} \end{cases} \tag{4.27}$$

with

$$\omega_s = max(min(\theta_1, \theta_2) \cdot \kappa_{\text{rel}}, \kappa) \tag{4.28}$$

With applying Equation 4.27 we eventually select the best modeling function $\mathcal{M}$ consisting of multiple partial fittings for this number of splitting points:

$$\mathcal{M} = (\mathcal{F}, \theta, \Phi) \tag{4.29}$$

The final model is then as defined in Equation 4.30:

$$m : (\mathcal{M}, \theta; \kappa, \kappa_{\text{rel}}, \gamma) \tag{4.30}$$

Using this approach with $n = 2$ for our example shown in Figure 4.7 results in the two fittings F_1^2 and F_2^2. The algorithm correctly detected the overload event and modeled both sides accurately. Thus, the resulting error is $\theta = 2.67\%$, more than ten times better compared to the model using a single fitting.

Determining Splitting Points

We use two methods to determine the set S_n of $n + 1$ splitting points s for multiple partial fittings. For two or three fittings, we use brute force, i.e., we calculate fittings for all possible combinations of splitting points:

$$S_n := \{\vec{s} \in \mathbb{N}^{n+1} \mid s_0 < s_1 < s_n, s_1 < s_2 < s_n, ..., s_{n-2} < s_{n-1} < s_n\} \tag{4.31}$$

with $s_0 = 1$ and $s_n = |G|$. This results in a total of $\mathcal{O}(|G|^n)$ fittings that need to be calculated. Computing these can be parallelized, while a single curve fitting requires less than 0.5 s, depending on the number of data points.

For a higher degree of splitting points, brute force is not feasible due to the increase in complexity. Therefore, we use a heuristic to determine likely occurrences of an event that merits a splitting point. We assume, that an event is indicated by a drastic change in inclination of the measurement data curve, e.g., a point after which the slope of the curve alters the direction. To determine these points, we calculate the second derivative of the measurement data and select the index, i.e., the x-axis index, of local maxima and minima. From the calculated set of maxima and minima we use the l highest absolute extrema as splitting points:

$$\mathcal{S}_l := \{\mathrm{idx}(\max_k(\{|g''(x)| \mid \forall x \in G\}))\} \tag{4.32}$$

As the measurement data is neither continuous nor do we know a function approximating the data, we calculate local piecewise derivates using finite differences.

To reduce the impact of measurement artifacts on the calculation of derivatives, we repeat this process while including one or more intermediate steps. For instance, one intermediate step can be to smooth the measurement data using local linear or polynomial regression. Again, the l highest extrema are added to our set of splitting points for which we calculate fittings.

We specifically perform several different rounds of manipulating the measurement data and determining splitting points for our total set of splitting points, as each operation, e.g., smoothing measurement data, may improve or worsen the quality of the determined splitting points for the current scenario. We further extend this set by adding the j surrounding data points for each determined splitting point:

$$\mathcal{S}_{l,j} := \mathcal{S}_l \cap \{x + a \mid \forall x \in \mathcal{S}_l, \forall a \in [-j, j] \in \mathbb{N}_0\} \tag{4.33}$$

We then use Equation 4.31 replacing $\mathbb{N}^{n+1}$ with $\mathcal{S}_{l,j}^{n+1}$. Extending the set of possible splitting points improves the overall accuracy at the cost of computation time. However, compared to brute force, the number of splitting points is drastically reduced and can be further restricted by the parameters l and j.

Similar to our error metric, the method to calculate derivatives for the measurement data can be replaced with other approaches [166] as plugins. For instance, different filters to reduce the impact of noise, like the Savitzky-Golay filter [167], can be applied to the measurement data before calculating the derivative function. However, these only provide scenario-specific optimized solutions, wherefore, the outlined approach is used by default.

4.4 Modeling a Software Data Plane Target

Section 4.4 is based on a collaboration between Dominik Scholz, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle [6]; The exact match model for increasing number of table entries in Section 4.4.4 is based on a collaboration between Dominik Scholz, Hasanin Harkous, Sebastian Gallenmüller, Henning Stubbe, Max Helm, Benedikt Jaeger, Nemanja Deric, Endri Goshi, Zikai Zhou, Wolfgang Kellerer, and Georg Carle [14].

We apply our measurement framework for P4 data planes generated using the t4p4s compiler. t4p4s produces DPDK-compatible code that runs on COTS CPU-based systems. As is typical for such systems, t4p4s' behavior is influenced by many factors, including interrupts, memory hierarchies, and cache sizes [28]. While achieving lower throughput and higher latency, the advantage is the virtual non-existence of limits regarding the P4 program complexity and custom functionality.

4.4.1 Framework Integration

All measurements were conducted in an automated and reproducible fashion using the framework presented in Section 4.3, integrated with our pos testbed controller introduced in Section 3.2. Due to pos' enforced automation and reproducibility, the whole process of the modeling framework is automated and reproducible, too. The framework employs the two-node setup shown in Figure 3.2, directly connecting the MoonGen load generator and the DuT.

Unless mentioned otherwise, we use an upstream t4p4s version from September 2020[1] with small changes due to performance[2] or functionality reasons. In particular, we templated the control plane such that it can be generated by the P4 program generator component of our framework. To further improve the t4p4s startup time, especially for experiments with extreme numbers of, e.g., match-action table entries, we also templated parts of the data plane startup code. In some cases, this reduces the startup time from several minutes, to a few seconds. Lastly, for selected experiments, we have increased limits, e.g., for the maximum size of match-action tables or hash data structures. We have not modified any components of the P4 processing pipeline.

The DuT running the t4p4s P4 switch is equipped with an Intel Xeon CPU E5-2640 v2 clocked at 2.0 GHz and an Intel X540-AT2 NIC. For all measurements, turboboost and hyperthreading were disabled to reduce performance jitter. Further, CPU cores were isolated and the t4p4s processes were pinned to CPUs. Unless mentioned otherwise, the CPU frequency was set to 2.0 GHz and we used DPDK version 19.03.

We use white-box measurements using Linux' perf utilities to measure CPU cycles and cache misses.

4.4.2 Resource Utilization

Memory consumption is typically no issue for P4 programs on a CPU-based system as modern COTS servers can provide RAM up to the TB range. However, the actual usage of memory has an impact on the performance when different levels of caches are involved. This may happen for match-action tables with a large number of entries, i.e., large BGP routing tables. Therefore, we use the white-box profiling measurements to analyze the impact of memory consumption and present the results as part of our performance model.

4.4.3 Baseline Model

We use the three-fold approach shown in Figure 4.8 to determine a baseline model for the t4p4s target. The first step, referred to as *DPDK Only*, does not include any t4p4s related code, focusing solely on receiving and sending packets using DPDK. This is done to understand the performance overhead of the underlying DPDK runtime. The second step, referred to as *Baseline*, adds a minimal P4 pipeline that each packet traverses, consisting only of a minimal

[1] Commit SHA: b3870d6aee2b68b8f340cabe50ccb21e93a217eb

[2] `https://github.com/P4ELTE/t4p4s/issues/23`

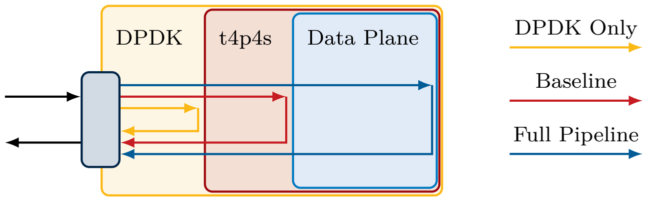
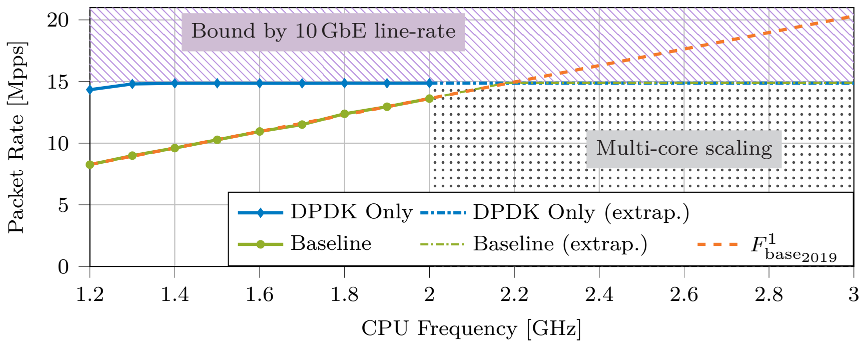
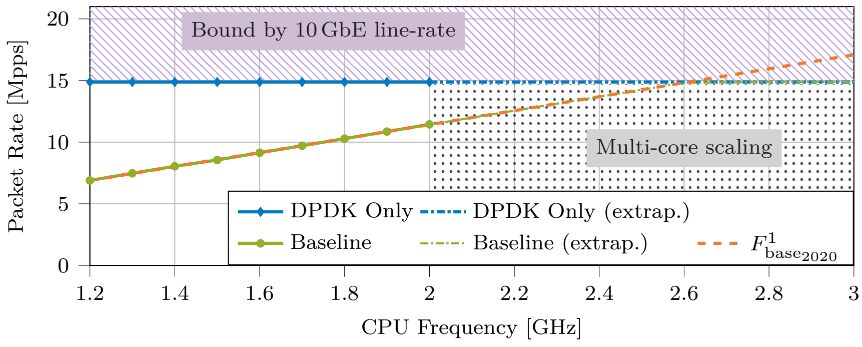


Figure 4.8: t4p4s' internal traffic flow



(a) Version March 2019



(b) Version September 2020

Figure 4.9: t4p4s baseline scaling of packet rate with CPU frequency

parser and deparser, and setting the egress port via the default action of one exact match-action table. Similar as before, the goal is to understand the processing overhead generated through t4p4s' boilerplate code. We use this *Baseline* model for comparison with all further measurements using the full pipeline.

CPU Cycles

Figure 4.9 shows the scaling of processed packet rate with CPU core frequency for two different versions of t4p4s: the March 2019 version used in our publication [6] (cf. Figure 4.9a), and the September 2020 version used for this work (cf. Figure 4.9b). At 1.3 GHz the *DPDK Only* program for the 2019 version is bound by the 10 GbE line-rate. Simply adding the boilerplate code generated by the minimal P4 program reduces the packet rate by approximately 6 Mpps. As the *DPDK Only* program for the 2020 version is bound by the line-rate even for the minimum

CPU frequency, we cannot exactly estimate the overhead generated by the boilerplate code. However, it must be at least 8 Mpps, showing that the 2020 t4p4s version is able to process at least 2 Mpps fewer packets for the same CPU frequency.

Increasing the CPU frequency on one core or processing with multiple cores results in linear scaling, bound by the line-rate. A higher frequency equals more CPU cycles per second, allowing more packet processing operations in the same interval. This is reflected by the linear models $F^1_{\text{base}_{2019}}$ and $F^1_{\text{base}_{2020}}$.

Typically, the bottleneck of a CPU-based software packet processing system is the CPU, i.e., the number of CPU cycles required per packet limits the number of processed packets. Based on Equation 2.1, we use the measured maximum packet rate $\widetilde{p}$ and the configured CPU frequency f to calculate the cycles per packet $\widetilde{C}$:

$$\widetilde{C} = \frac{f}{\widetilde{p}} \tag{4.34}$$

We integrated the calculation of this metric into the framework and use CPU cycles per packet for the following experiments as default metric. We want to note, that CPU cycles per packet can only be calculated when we observe packet loss. Only in this case we can actually determine the maximum number of packets that could be processed.

Using Equation 4.34 to calculate the cycles per packet for the baseline scenarios results in models consisting only of a constant factor. Processing of a single packet requires approximately 84 and 146 cycles for the *DPDK Only* and *Baseline* scenarios of the 2019 t4p4s version, respectively, and 175 cycles for the 2020 *Baseline* version. While these values are in the expected range for DPDK packet processing [28], the calculation also confirms that the 2020 t4p4s version has a higher baseline cost of almost 30 CPU cycles when processing packets. We attribute this worse performance to added functionality and, therefore, also the increased complexity of the newer version.

For all further experiments, we use the 2020 t4p4s version and the CPU is clocked at $f = 2\,\text{GHz}$. Using the notation for the best fitting and resulting model introduced in Equations 4.20 and 4.21, the model for constant CPU cycles consumption of the baseline program can be written as shown in Equations 4.35 and 4.36:

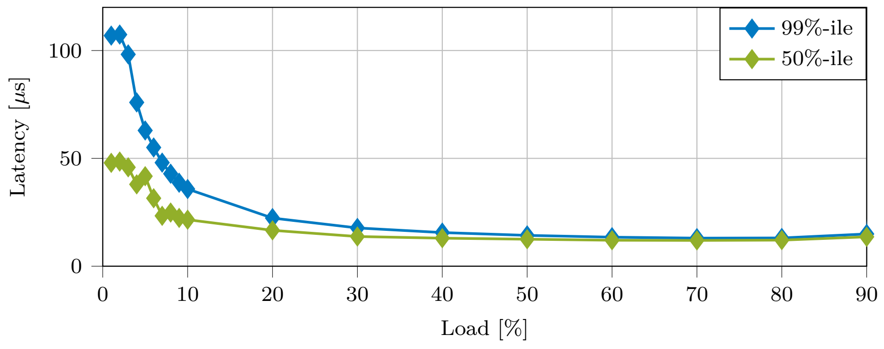
$$F_{\text{base}} = (\langle \tau(x) = p_1^*, 1\rangle, \langle 175\rangle, 1.2, 2.0, 0.13\%) \tag{4.35}$$

$$m^c_{\text{base}} : (F_{\text{base}}, 0.13\%; 10\%, 0.5\%, 10^{-5}) \tag{4.36}$$

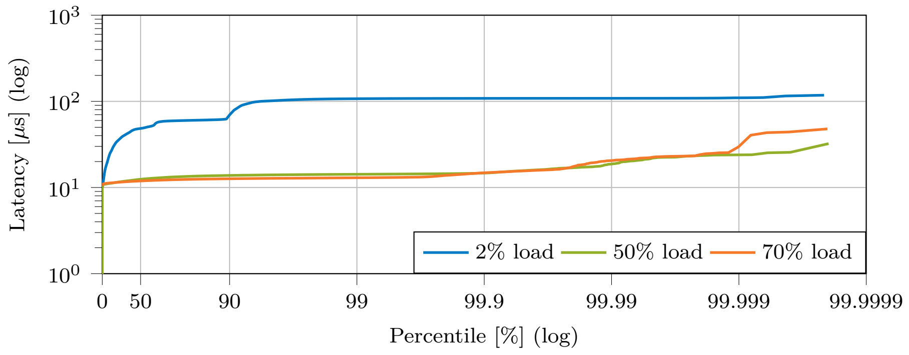
As the modeling function is constant, the vector of free fitting parameters in Equation 4.35 is empty. To shorten this verbose notation we assume $\kappa_{\text{rel}} = 10\%$, $\kappa = 0.5\%$, and $\gamma = 10^{-5}$ for the remainder of this work, unless mentioned otherwise. The shortened form of the model presented in Equation 4.36, solving the function prototype τ with the fitted parameter vector $\vec{p}$, is then written as follows:

$$m^c_{\text{base}} : (M(x) = 175, 0.13\%) \tag{4.37}$$

We use this notation for the remainder of this work.



(a) Median and 99%-ile



(b) HDR for selected loads

Figure 4.10: Baseline latency of t4p4s for different CPU loads on a single CPU core

Latency

While latency as metric is not the primary focus of this study, we want to provide a brief overview of the baseline t4p4s latency. We present different latency percentiles for increasing load percentages relative to the maximum processed packet rate in Figure 4.10.

t4p4s shows a latency behavior typical for software packet processing systems. As depicted in Figure 4.10, for less than 10 % load, latency is higher due to batch processing of DPDK. As shown in the HDR histogram for 2 % load in Figure 4.10b, batches are processed every 100 μs in the worst-case [168], resulting in the plateau for the 99 %-ile. For higher loads, batches fill quicker and are processed before the timeout, resulting in the higher percentiles being closer to the median latency. For these loads, system interrupts and other side effects cause a long tail [169], which persists throughout all following measurements.

4.4.4 Component Models

We derive the component model presented in Section 4.1.2 for t4p4s by applying our automated measurement and modeling framework introduced in Section 4.3. As it is infeasible to accurately model every single component that the P4 language provides in detail, we try to identify performance dominating factors and focus our modeling on their features. We assume that other, unmodeled components, either have a minor impact or add a constant cost to the overall

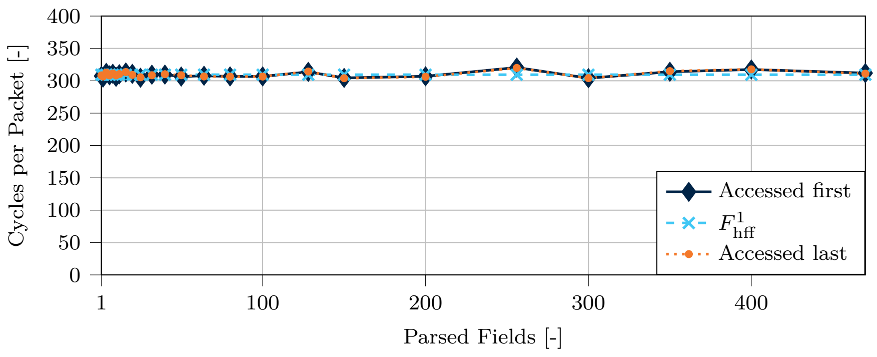


Figure 4.11: Scaling parsed header fields

model. The remaining task is to determine, which P4 language constructs and their features are dominating factors. To answer this, we look at common L2 switch and L3 router programs and focus on their included P4 language components. We focus on built-in P4 components, i.e., externs, like registers or hashing, or not included in our evaluation.

For selected experiments we artificially increase the complexity of the program, e.g., by applying multiple match-action tables. This is done to ensure that the measurements, when scaling the actual component of interest, are not limited by line-rate. If this would happen, the actual maximum processed packet rate and CPU cycles per packet could not be calculated. As this added complexity remains constant throughout the measurement series of one component experiment, the performance overhead will be a constant factor in the resulting model. For the actual scaling of the component, this constant offset has to be removed from the model.

Parser and Deparser

We identified two primary actions performed in the parser and deparser stages: parsed header fields that are used in subsequent processing pipelines; and changing the length of the packet by adding or removing header fields.

Number of Parsed Header Fields: Thereby, we differentiate two scenarios for the parser stage. First, the parsed header fields are not used throughout the data plane, and, second, the trailing parsed header fields are used as keys for a match-action table. The distinction is important to verify that the parser actually parses the header fields, as parsing of unused header fields could be optimized away during the P4 compilation stage. Furthermore, the compiler might reduce parsing only up to the last header field that is used throughout the remaining program.

As shown in Figure 4.11, the derived model is constant in both cases, i.e., the cost of simply parsing additional header fields is not measurable.

A noteworthy curiosity is that t4p4s allows the parsing of more bytes than the length of the received packet. This is possible because of DPDK's underlying data structure used to store packets in memory, the *mbuf*. This structure has a fixed length to store packets of at least 1500 B. Therefore, parsing additional fields results in continuing to parse the respective *mbuf*

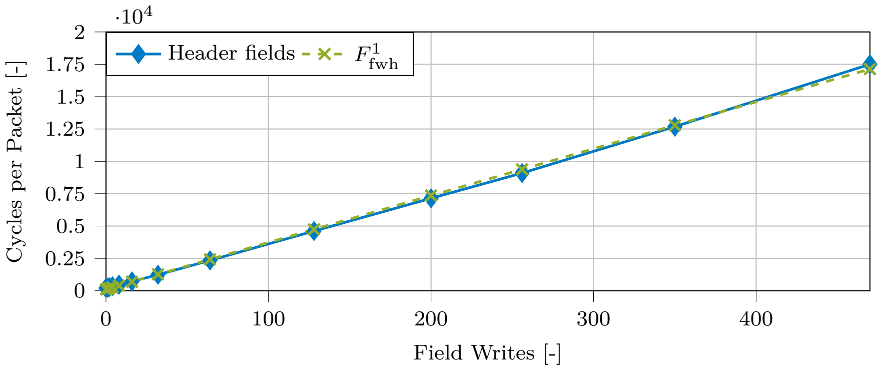


Figure 4.12: Scaling field writes

structure, independent of packet size. Similarly, writing beyond the packet, as long as it is within the *mbuf*'s reserved memory, is possible.

Number of Modified Fields: In contrast to the previous experiment the parsed fields are actually modified by writing a constant value. Although this action is not exclusively performed in the parser and deparser stages, we consider it tightly coupled with the number of parsed header fields. We differentiate writing 1 B fields that are parsed from packet headers and metadata fields. For the former, we use 512 B packets and always parse all fields as single bytes, allowing us to test up to almost 500 individual header field writes. For metadata fields we use 64 B packets parsing no additional header fields.

The resulting CPU cycles scale linearly with the number of modified header fields as shown in Figure 4.12. We exclude showing metadata fields as it displays the same behavior, however, is limited to approximately 50 modified fields.

$$m^c_{\text{fwh}} : (M(x) = 36.22x + 109, 1.49\%) \tag{4.38}$$

Based on the resulting model, shown in Equation 4.38, modifying a 1 B header field costs approximately a constant 36 cycles.

Adding and Removing Headers: This experiment evaluates the feature of adding or removing additional bytes in the form of headers to or from the parsed packet. Thereby, we differentiate between adding or removing either one large, or several small headers.

Figure 4.13 shows that adding a single header of increasing size has only a small additional cost per added byte. In contrast, adding multiple headers of 1 B results in a linearly increasing model. This is due to the overhead generated by creating and handling separate headers, in comparison of only increasing the size of a single header. The same relation can be seen for removing headers. However, compared to the model for adding headers m^c_{hadd}, the per-header cost is less then a fourth for the removing headers model m^c_{hrem}. This difference fits with the assumption that removing bytes from the packet is simpler than adding them.

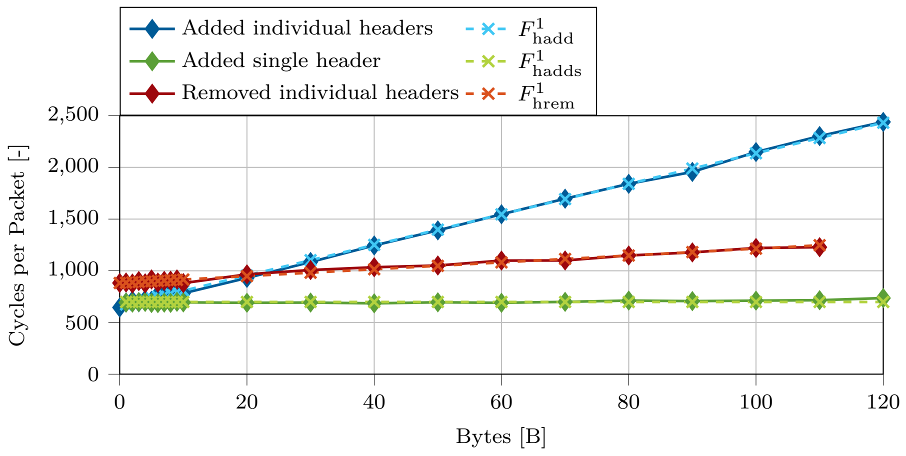


Figure 4.13: Scaling added header bytes

$$m^c_{\text{hadd}} : (M(x) = 14.77x + 659.44, 0.55\%) \tag{4.39}$$

$$m^c_{\text{hrem}} : (M(x) = 3.32x + 881.84, 0.59\%) \tag{4.40}$$

When removing the constant factors from Equations 4.39 and 4.40, we conclude that every added header causes an overhead of approximately 15 CPU cycles, while removing one costs less than 4 cycles.

Adding or removing more than 120 B is not possible for t4p4s due to a hard-coded limitation by DPDK.

Match-Action Tables

In P4, packet processing tasks are expressed as a series of matches and actions on packet or metadata. Being at the center of every P4 program, the match-action performance is crucial to understand the performance of the entire packet processing pipeline. Therefore, we argue that analyzing and modeling match-action table-related features is vital for accurately modeling paths through the data plane.

We have identified the following features for match-action tables:

- the number of match-action tables in a P4 program
- the table's match types—exact, LPM, or ternary—which determine the mode of comparison between a packet header or metadata field value and available table entries
- the size of individual table entries, defined by the size and number of keys
- the amount of action data associated with a table entry
- and the number of entries in the match-action table

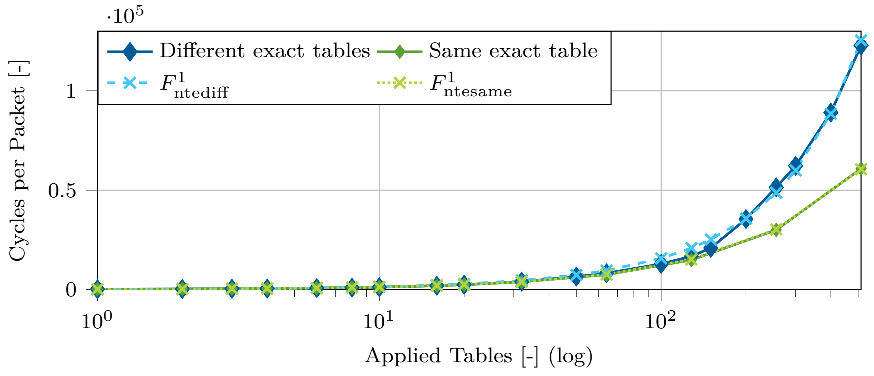


Figure 4.14: Scaling number of match-action tables

For the sake of simplicity, we assume that one match-action table only consists of one match type.

Number of Applied Match-Action Tables: First, we want to analyze and model the number of applied tables in the P4 program. As, e.g., the `switch.p4` program[1], a feature-complete switch intended for use in data centers, contains approximately 160 different match-action tables in its entirety, we analyze up to 1024 match-action tables.

While hardware P4 targets typically do not allow the application of the same table multiple times per packet, this restriction does not exist in t4p4s. This flexibility allows a closer estimation of the cost of the actual table application, i.e., hash calculation for exact matches, without fetching data for keys and entries. The data fetched is the same for successive table applications, getting cached and amortizing the cycles required. Therefore, we analyze both scenarios for exact match tables, applying the same table n times and applying n different tables.

$$m^c_{\text{ntesame}} : (M(x) = 101.74x + 97.30, 0.67\%) \tag{4.41}$$

$$m^c_{\text{ntediff}} : (M(x) = 109.82x + 98.40x, 2.63\%) \tag{4.42}$$

As shown in Figure 4.14 and Equation 4.41, per-packet CPU cycles increase linearly with the number of repeated applications of the same exact match table. The primary factor is the cost of calculating the hash used to access the table as the cost for loading the match keys is amortized through caching. When applying different exact match tables—the more common usage in P4 programs—the derived model in Equation 4.42 shows only a slight increase for the linearly scaling factor.

For the ternary and LPM match types, we show the derived models for applying different tables in Equations 4.43 and 4.44, respectively. While they also show a linear dependency, applying match-action tables of those match types is less expensive. Ternary matches are a simple lookup

[1] `https://github.com/p4lang/switch`

in a linked list, i.e., no hash has to be calculated. Similar, the lookup in the DIR-24-8 data structure for the LPM match does not require lengthy calculations, explaining the reduced per-table cost.

$$m^c_{\text{ntter}} : (M(x) = 44.75x + 99.57, 0.96\%) \tag{4.43}$$

$$m^c_{\text{ntlpm}} : (M(x) = 77.97x + 93.26, 3.90\%) \tag{4.44}$$

For exact match tables, we use the model shown in Equation 4.42, applying different tables. Further, we deduct the constant factor as we are interested in the scaling of the component:

$$m^c_{\text{ntex}} : (M(x) = 118.82x, 2.63\%) \tag{4.45}$$

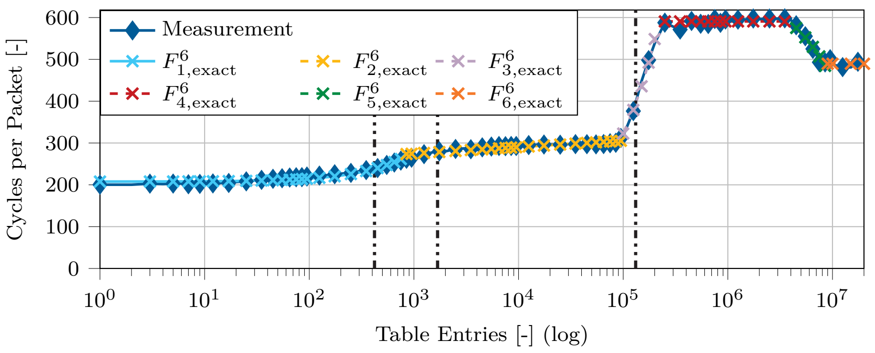
Increasing the Number of Table Entries: The number of table entries is a key factor for many P4 applications. As the CPU target has plenty of memory compared to hardware targets, we can push the limits for the possible number of table entries. A possible deployment scenario are BGP IPv4 routing tables, which have a steadily increasing count of unique routable prefixes, reaching more than 860 000 IPv4 entries at the beginning of 2021 [170]. Another example are firewall rule-sets, which might scale to thousands of exact, ternary, or LPM matches. Scaling the number of table entries up to millions, might reveal other bottlenecks than the CPU. Each table entry requires memory for storage, including the match keys and action data. Consequently, we expect the memory hierarchy, i.e., different caches and their capacity, to influence the performance. To identify these bottlenecks, we use profiling to determine cache misses as additional metric.

Another aspect are match types that can be realized using specialized hardware, e.g., ternary content-addressable memory (TCAM) for ternary and LPM matches. In a software target, however, different algorithms with varying properties in regard to limitations and expected performance are used.

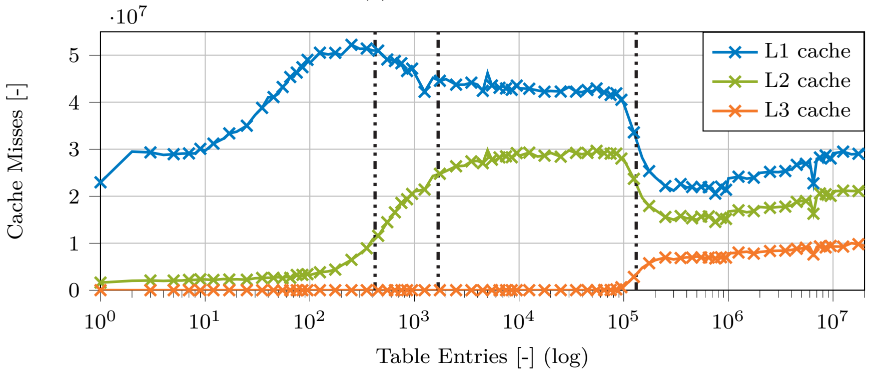
Caches can accelerate memory accesses by saving entries that are queried with a high probability. We aim for the worst-case scenario, generating traffic such that every packet hits another table entry. Consequently, all table entries have to be constantly loaded, evicting old entries. For all match types, table entries consist of a single 32 bit match key.

Exact Match Figure 4.15a depicts the CPU cycles per packet for an increasing number of exact match entries. The modeling framework has identified six different parts that are modeled individually. Using profiling, the first four parts can be explained. Initially, a model with linear factor of 0.072 is proposed, i.e., the behavior is almost constant for less than 100 table entries. DPDK exact match tables use cuckoo hashing with constant worst-case lookup time[1]. Fetching the data from the packet headers used as match keys is also constant as it is the same for all numbers of table entries. However, for the linearly increasing number of table entries, the memory that has to be loaded is also linearly increasing. Starting with approximately 10^2 table

[1] `https://doc.dpdk.org/guides/prog_guide/hash_lib.html`



(a) CPU cycles



(b) Cache misses

Figure 4.15: Scaling number of exact match entries. The vertical lines from left to right represent the estimations for L1, L2, and L3 cache exhaustion using the resource model

entries, the data does not fit into the L2 cache any longer, resulting in an increasing number of L2 cache misses as shown in Figure 4.15b. As a consequence of this increase, at circa 10^3 table entries, the model switches to a logarithmic function. For more than 10^5 table entries, table entry data has to be fetched from main memory as indicated by the increase in L3 cache misses. As these fetches take significantly longer than fetches from fast caches, the CPU cycles per packets double after a transition period. This transition is modeled independently as linear function. After the transition, the load operations from main memory outweigh other effects, resulting in a constant model. However, after approximately $4.5 \cdot 10^6$ table entries, the level changes once again, being modeled by another linear transition and constant level thereafter. This last effect is not explained by any other metric. The resulting CPU cycles model for this experiment is given in Equation 4.46 with a combined error of 0.730 %.

$$\mathcal{M}_{\text{exact}}(x) = \begin{cases} F^6_{1,\text{exact}}(x) = 0.072x + 207.71, & 0 < x < 8.5 \cdot 10^2 \\ F^6_{2,\text{exact}}(x) = 4.66 \cdot \log_2(188.07x) + 193.01, & 8.5 \cdot 10^2 \leq x < 1.0 \cdot 10^5 \\ F^6_{3,\text{exact}}(x) = 0.00226x + 96.63, & 1.0 \cdot 10^5 \leq x < 2.5 \cdot 10^5 \\ F^6_{4,\text{exact}}(x) = 590.74, & 2.5 \cdot 10^5 \leq x < 4.5 \cdot 10^6 \\ F^6_{5,\text{exact}}(x) = -0.00002x + 677.12389, & 4.5 \cdot 10^6 \leq x < 9.0 \cdot 10^6 \\ F^6_{6,\text{exact}}(x) = 489.47, & 9.0 \cdot 10^6 \leq x \end{cases} \tag{4.46}$$

Inspecting the implementation of exact matches results in Equation 4.47 to model the required resources m^r_{exact} in bytes for an exact match table based on the number of table entries e, total key size r_{keys} in bytes, and size of the action data r_{actions} in bytes:

$$\begin{aligned} m^r_{\text{exact}}(e, r_{\text{keys}}, r_{\text{actions}}) &= 2 \cdot 64\,\text{B} + \underbrace{((r_{\text{keys}} + 64\,\text{B}) \cdot e)}_{\text{Hash table}} + \underbrace{(8\,\text{B} \cdot e)}_{\text{Entries}} + \underbrace{(r_{\text{actions}} \cdot e)}_{\text{Actions}} \\ &= 128\,\text{B} + e \cdot \underbrace{(r_{\text{keys}} + r_{\text{actions}} + 72\,\text{B})}_{\text{Table entry size}} \end{aligned} \tag{4.47}$$

The size of the exact match-action table structures amounts to $2 \cdot 64\,\text{B}$ due to cache-line alignment. The size of each table entry is defined by the total key size, the calculated hash (cache aligned), action data (cache aligned), and a fixed-size pointer. We solve Equation 4.47 for e, using $r_{\text{keys}} = 16\,\text{B}$ and $r_{\text{actions}} = 64\,\text{B}$:

Setting $m^r_{\text{exact}} = 20\,\text{MB}$, the L3 cache size of the used processor, results in $1.32 \cdot 10^5$ entries to fully fill the L3 cache. This point, marked in Figure 4.15a, is an overestimation as the cache is not exclusively used for table entries. For instance, the packet data is also loaded into the caches. Similarly, the point for the performance drops caused by increasing number of L2 cache misses can be roughly calculated using Equation 4.47 and $m^r_{\text{exact}} = 256\,\text{kB}$, resulting in $1.68 \cdot 10^3$ table entries. While this estimation is close to the point of the detected event, due to access time difference between the L2 and L3 caches of less than 5 ns, the performance loss is not as noticeable as when exceeding the L3 cache [171]. The increase in L1 cache misses is not detected by the automated modeling approach. This is due to the even smaller difference in cache access times from L1 to L2 cache. However, using the resource model of Equation 4.47 with the L1 cache size of $m^r_{\text{exact}} = 64\,\text{kB}$, results in 420 table entries.

The accuracy of the resource model is less accurate when predicting the L1 and L2 cache sizes. We attribute this to the remaining data that is stored in the caches. However, the performance impact is also less when exceeding the L1 and L2 caches. Only when exceeding the L3 cache, the performance in CPU cycles doubles, for which the resource model in Equation 4.47 is accurate. Therefore, we argue, that the point of exceeding the L3 cache size is the critical factor to model.

While the modeling approach correctly identified the events that lead to a change in performance, the model might be unnecessarily complex for use in performance prediction. In particular, the

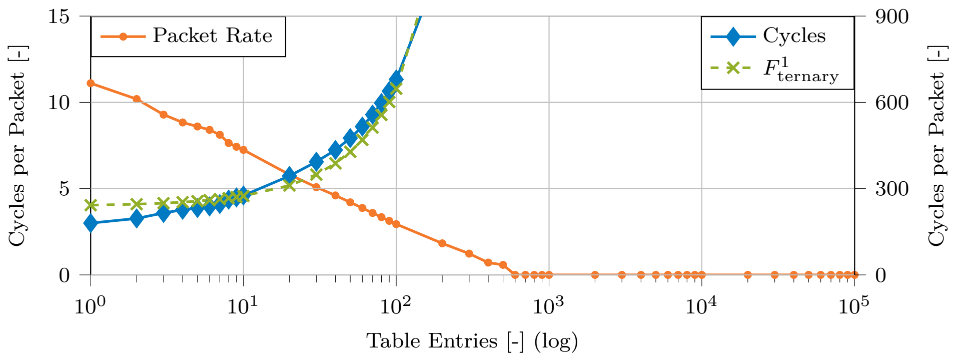


Figure 4.16: Scaling number of ternary match entries

last two parts of the model are unnecessary for two reasons. First, the analyzed effect is both unexpected and unexplained, and, second, the number of table entries is in an extreme region, unlikely to be seen in most P4 applications. Therefore, we simplify the resulting model by dropping $F^6_{5,\text{exact}}$ and $F^6_{6,\text{exact}}$. The resulting model is simplified with only four different fittings, whereby all individual parts can be explained by the memory-based resource model. Further, we deduct the constant baseline factor of 207.78 to derive the component scaling.

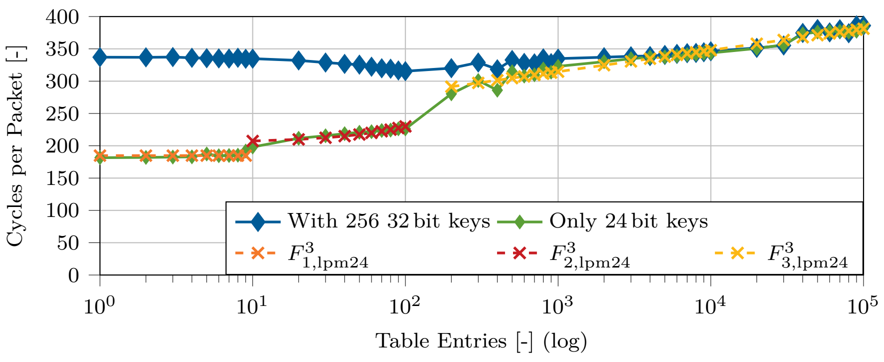
Ternary Match Due to the lack of specialized hardware like TCAM, efficiently implementing ternary matches in software is difficult. True parallelized lookups can hardly be implemented. Further, more than one entry could match. The current implementation of t4p4s simply iterates through the entire linked list of table entries, returning the last matching entry. Thus, in theory, this results in a search complexity of $\mathcal{O}(n)$. As shown in Figure 4.16, only less than approximately 600 ternary entries can be processed at maximum. The resulting model uses a second degree polynomial:

$$m^c_{\text{ternary}} : (M(x) = 0.00606x^2 + 3.48x + 238.89, 3.639\%) \tag{4.48}$$

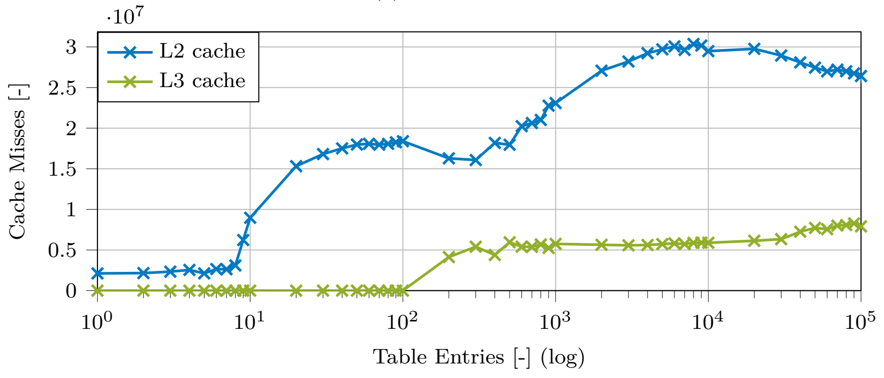
Thereby, the quadratic factor of 0.00606 is very small, only becoming impactful for more than 500 entries. As this is close to the limit of maximum ternary entries, the quadratic factor is essentially without impact. Consequently, the linear factor, indicating approximately 3.5 CPU cycles per ternary table entry, is dominating, matching the theoretic complexity of $\mathcal{O}(n)$ for this number of ternary table entries.

Due to the lower number of table entries ($< 10^3$) in this scenario, there is no visible impact of memory accesses and L2 or L3 cache misses on the performance.

LPM Match t4p4s uses a DIR-24-8 data structure [172] for 32 bit key sizes, i.e., IPv4 LPM. While a different data structure is used to allow 128 bit keys for IPv6 LPM, we focus on the former. DIR-24-8 uses two different types of tables: `tbl24` is a single table to store the most significant 24 bit of the prefix in up to 2^{24} entries. The second type of table is the `tbl8` of which, by default, 2^8 tables exist to store the remaining 8 bit. Prefix lengths of less than 24 bit can



(a) CPU cycles



(b) Cache misses

Figure 4.17: Scaling number of LPM match entries

be resolved with one lookup in the first table, longer prefixes require a second lookup in the respective `tbl8`. DIR-24-8 assumes that routes with a prefix length of greater than 24 bit are rare, optimizing lookups for smaller prefix lengths, while limiting the number of greater than 24 bit prefixes that can be stored. This represents a trade-off between memory complexity and performance. [172]

Figure 4.17a shows the results when comparing using exclusively 24 bit prefixes with using an additional 256 32 bit prefixes. Inserting more than 100 000 entries takes a considerable amount of time wherefore those measurements are excluded. For the former case, the derived model shows three different parts. For only up to nine table entries, the derived model shows the expected constant scaling. Profiling presented in Figure 4.17b reveals that at this point the L2 cache misses increase. For the second part, the modeling framework determines a linear scaling. Thereby, the factor is only 0.25, reflecting the small added cost of the L2 cache misses happening for a percentage of packets. Starting with 200 LPM table entries, the L3 cache is the limiting factor, as already at this point approximately 30 % of the lookups require data fetched from main memory. This is due to the large size of the DIR-24-8 structure, as already the `tbl24` requires 64 MB [172], [173]. Furthermore, the packet data is stored in the caches. As a consequence, the shared L3 cache is filled with per-core DIR-24-8 structures. For this third part, we use a logarithmic function as model:

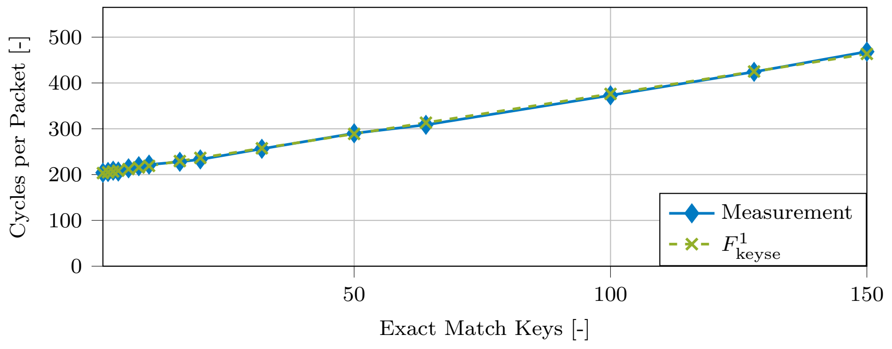


Figure 4.18: Scaling the number of exact match keys

$$\mathcal{M}_{\text{lpm}}(x) = \begin{cases} F^3_{1,\text{lpm24}}(x) = 184.79, & 0 < x < 10 \\ F^3_{2,\text{lpm24}}(x) = 0.25x + 204.88, & 10 \leq x < 200 \\ F^3_{3,\text{lpm24}}(x) = 14.43 \cdot \log(253.41x) + 135.11, & 200 \leq x \end{cases} \tag{4.49}$$

$$m^c_{\text{lpm}} : (\mathcal{M}_{\text{lpm}}, 0.655\%) \tag{4.50}$$

As with the previous cases, we deduct the constant effect for one table entry of 184.79 cycles from the model to reflect only the scaling with table entries.

The influence of prefix length is also shown in Figure 4.17a. We added a constant 256 32 bit prefixes to `tbl8` such that every table match now requires of two lookups. For less than 200 entries a performance increase is noticeable. The additional lookup almost doubles the required CPU cycles. When further increasing the number of table entries, the cost for the additional 256 entries amortizes, resulting in the cost for the additional lookup in the `tbl8` structure. For simplicity, we do not include the additional cost in our model.

Increasing the Entry Size: The size of an individual table entry can be increased either by increasing the size or number of match keys, or by increasing the associated action data. In both cases, this changes the parameters r_{keys} and r_{actions} for our resource model m^r_{exact} and, therefore, when the different cache limits are reached. Our experiments show that this is indeed the case. However, for ternary and LPM tables the already discussed limitations of the respective table implementations outweigh this effect.

Number of Match Keys The first property that we scale to increase the size of match-action table entries is the number of 1 B match keys. Figure 4.18 presents the results for up to 150 exact match keys. As expected, it scales linearly, as each additional match key requires another data access. According to the derived model shown in Equation 4.51, this costs approximately 1.75 cycles per match key.

$$m^c_{\text{keyse}} : (M(x) = 1.75x + 201.54, 0.407\%) \tag{4.51}$$

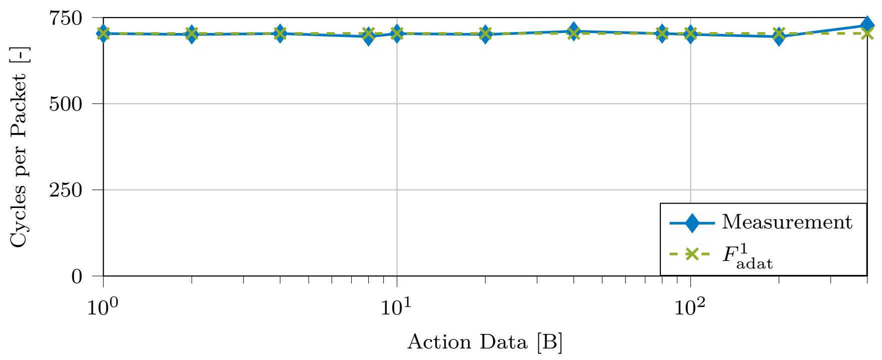


Figure 4.19: Scaling amount of action data

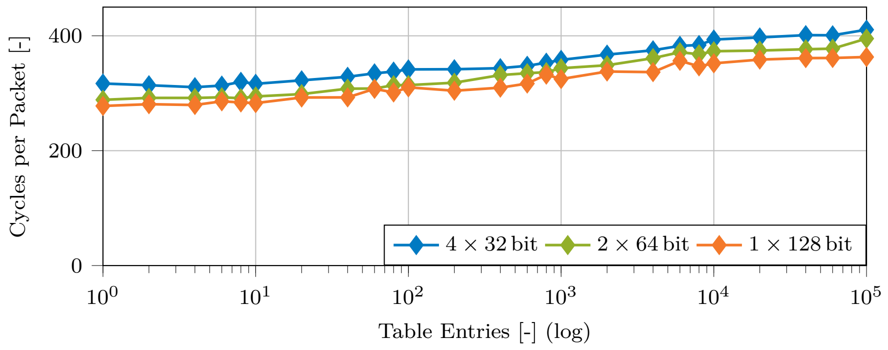


Figure 4.20: Influence of different data types to represent a 16 bit exact match key

Repeating this experiment for ternary matches results in similar per-key cycle costs. However, we do not include the number of table keys in our final model for match-action tables for two reasons. First, every table key is already part of our model for the number of table entries. Second, the data used as table key originates from the incoming packet or derived metadata and as such is limited in comparison to potential thousands or more table entries. Thus, we argue, that memory and caching effects inherited through the number of table entries are dominating.

Due to implementation restrictions, changing the number of table keys is not implemented for the LPM match type. Furthermore, the supported width for table keys is limited for all match types, as they depend on available and supported data types. Therefore, we do not include these factors in our model.

Action Data Size Lastly, we scale the number of 1 B action data that is returned for the matching table entry. As shown in Figure 4.19, no measurable increase is noted, resulting in a constant model. Similar to the previous features increasing the size of a table entry, the action data itself has no unexpected impact. However, storing more than 400 B of data for an entry in t4p4s is not possible. While we expect this to be no problem for most applications, this does not allow storing, e.g., cryptographic key material of more than 400 B.

Data Type Influences Data can be represented using different data types on CPU-based systems. For instance, a 128 bit match key can be represented using four 32 bit, two 64 bit, or a single 128 bit data structure. While the former is used as default in t4p4s, we repeated the measurement with increasing number of exact match entries using the latter two data structures. As shown in Figure 4.20, the default implementation shows worse performance compared to the alternative approaches in this scenario. As the total key size is the same, we can rule out memory access effects as cause for the performance differences. However, profiling reveals that the reduced performance for the key composed of four times 32 bit is architecture-specific. In this case, Store Forwarding failed in 100 % of the cases as the data is passed incorrectly to the hash function. Store Forwarding is a feature to directly forward previous memory writes to a subsequent memory read. Performance can be saved as the data does not have to be written to main memory in-between [174]. According to the data sheet for the CPU architecture used in the DuT, this failure results in a 12 CPU cycles performance penalty [174]. Optimizing the code generated by t4p4s would allow Store Forwarding to succeed, yielding better performance. Our measurement shows that the hash function performs better when the passed data is stored in large consecutive chunks of memory.

Summary

We have identified the dominating component for P4 programs on the t4p4s target to be match-action tables. Every applied table adds approximately 136 CPU cycles per packet. Thereby, the content of the table is less important, however, differences between the match types can be observed. For exact matches, only when the number of table entries that have to be constantly matched exceeds the L3 cache size a significant performance drop is observed. In contrast, for ternary matches the memory architecture is not the bottleneck. Instead, the data structure and resulting search complexity are the throttling factor. LPM matches can be efficiently executed for up to 100 entries. Afterwards, the L3 cache becomes the bottleneck due to the large DIR-24-8 data structure. The amount of action data or match keys only shows little influence on the performance.

For other components, single byte header field modifications are costly with approximately 36 cycles. Further, adding bytes to the packet costs up to 15 CPU cycles, while removing bytes has only a small influence with 3.3 CPU cycles.

After having modeled the individual components, we can revise our baseline model. As it consists of one applied exact match-action table, we deduct the respective value from the original baseline:

$$F_{\text{base\_revised}} = F_{\text{base}} - F_{\text{ntex}}(1) = 175 - 109.82 = 65.18 \tag{4.52}$$

According to our model, the DPDK-based t4p4s code requires approximately 65 cycles, independent of the remaining processing in the data plane. We use this constant offset for our path calculations as adjusted baseline.

4.4.5 Modeling of Program Paths

We use the component models for the t4p4s target to model paths of two of the most common data plane programs: an L2 learning switch and an L3 router. As discussed in Section 4.2.1, there are currently no tools for reliably deriving all paths from a P4 program. Therefore, we perform this step manually based on the program's intermediate JSON representation generated by p4c. For each of the paths we then sum up the cost as determined by our models of the individual components on the path. To judge the quality of the resulting path models, we perform measurements that target each individual path.

Generating packets that match a certain path requires knowledge about state information provided by the control plane. Therefore, we limit our discussion to whether, using this approach, it is possible to determine the worst-case or average case path and to create respective matching packets for the programs at hand.

Layer 2 Switch

The L2 learning switch example applies a total of two tables after parsing the Ethernet header: the first table matches on the packet's source layer two address. If the address is known, no action is performed, otherwise, the address is learned using a digest to communicate with the control plane. The second table matches on the destination Ethernet address. If the outgoing port was already learned, it is set and the packet is forwarded. Otherwise, the packet is broadcast.

Both tables are always applied, whereby each has two possible actions as outcome, resulting in a total of four different paths. However, whether the packet is forwarded or broadcast only results in one header field being modified, i.e., according to our model, these paths can be merged. Therefore, we only differentiate two paths: with and without sending the digest to the control plane, referred to as $\mathcal{P}^1_{\text{L2}}$ and $\mathcal{P}^2_{\text{L2}}$. Information about the number of each table's entries can only be retrieved from the control plane and might change during program execution. For this model, we assume that the tables only consist of one entry such that the desired path is taken. Both paths consist of the same components, aside from the additional digest for the former path:

$$\mathcal{P}^1_{\text{L2}} = \mathcal{P}^2_{\text{L2}} + C_{\text{digest}} \tag{4.53}$$

$$\mathcal{P}^2_{\text{L2}} = m^c_{\text{base}} + m^c_{\text{ntex}}(2) + m^c_{\text{exact}}(2) + m^c_{\text{fwh}}(1) = 321.18 \tag{4.54}$$

Using measurements, we observed that the actual cost for the second path is 253.81 CPU cycles, i.e., the model presented in Equation 4.54 results in an overprediction of approximately 26.55 %. We identified three reasons for this overprediction. First, the full program does not strictly consist of components that can be added up. For instance, the P4 frontend and the t4p4s backend compilers might perform optimizations. While we observe influences like the memory architecture for individual components, their influence for the whole program might be different. The model does not take into account that data might already be loaded into the cache, because it was used by a previous component. The experiments to derive component models, however, assume that each byte has to always be loaded. Further, every measurement to determine the

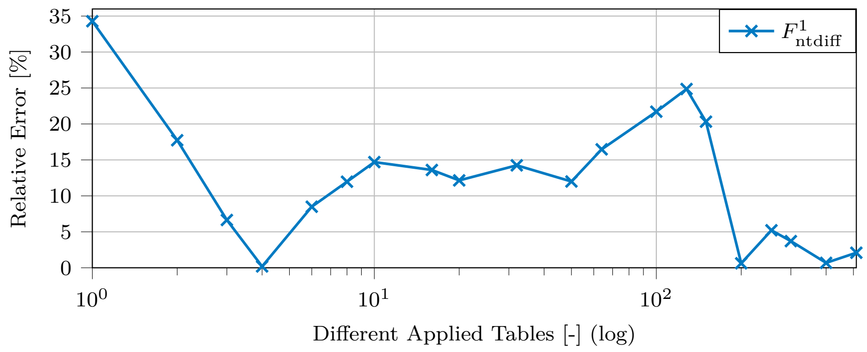


Figure 4.21: Relative error of the model for scaling different numbers of tables

scaling of an individual component always also includes other components. While we try to reduce their impact, it is possible that their influence is notable in the derived model.

Second, the high error is a result of using the curve fitting approach. We intentionally scaled each component to extreme values, e.g., thousands of tables or millions of table entries. Modeling the resulting slope with our curve fitting approach may result in selected data points being inaccurately modeled. Figure 4.21 shows the relative error between every measured and modeled data point for the number of different applied exact match tables. In particular for the case of two tables, the value used for the above path calculation, the error is at approximately 18 %.

The accuracy for fewer tables could be improved by modeling, e.g., only up to 100 applied tables.

The last reason are fluctuations caused by DPDK. We commonly observe that the maximum packet rate of a program differs for each restart of the program and DPDK. These fluctuations can be up to 0.5 Mpps. While our measurements for deriving the model use the maximum of three repeated measurements, this maximum might still be influenced by these fluctuations. Further, the measurement for the complete program path is subject to these fluctuations. This leaves the chance that, for instance, not the true performance optimum was modeled, resulting in a higher per-packet cycle usage for a specific component.

The digest is an extern function we did not include in our model. However, as the cost will be greater than zero cycles, $\mathcal{P}^1_{\mathrm{L2}}$ including the digest will be the worst-case path. The average case for the L2 program is one of the paths where the source Ethernet address is already learned. Generating traffic for this path is possible: for every packet with new source layer two address, the first packet will trigger the learning mechanism via the digest, however, subsequent packets will take the cheaper path. For the same reason, generating traffic only for the worst-case path is not possible. Each initial worst-case packet influences the state of the device, resulting in the source Ethernet address being learned, i.e., future packets will not take this path. Consequently, the traffic has to consist of packets with always new source Ethernet addresses, however, this is limited by the size of the 48 bit address field.

Layer 3 Router

The L3 router is more complex than the L2 switch, consisting of up to three tables. After parsing Ethernet and IP headers, an exact match-action table is applied on the destination Ethernet address for the purpose of whitelisting. If no entry is found, the packet is dropped, i.e., no further processing occurs. Otherwise, the next table is applied using an LPM match on the destination address. The result is either the next hop or the packet is dropped. Finally, using another exact match on the next hop, the third applied table determines and modifies the outgoing Ethernet addresses, port, and TTL. We use an L3 router example without checksum calculation as this hash function is not included in our model.

The program consists of four different paths as the packet might be dropped immediately after each of the table applications. We list the components and cost of each path in the following:

$$\mathcal{P}^1_{\text{L3}} = m^c_{\text{base}} + m^c_{\text{ntex}}(1) + m^c_{\text{exact}}(1) + m^c_{\text{fwh}}(1) = 211.29 \tag{4.55}$$

$$\mathcal{P}^2_{\text{L3}} = m^c_{\text{base}} + m^c_{\text{ntex}}(1) + m^c_{\text{exact}}(1) + m^c_{\text{ntlpm}}(1) + m^c_{\text{lpm}}(1) + m^c_{\text{fwh}}(1) = 289.33 \tag{4.56}$$

$$\mathcal{P}^3_{\text{L3}} = m^c_{\text{base}} + m^c_{\text{ntex}}(2) + m^c_{\text{exact}}(2) + m^c_{\text{ntlpm}}(1) + m^c_{\text{lpm}}(1) + m^c_{\text{fwh}}(2) = 435.37 \tag{4.57}$$

$$\mathcal{P}^4_{\text{L3}} = m^c_{\text{base}} + m^c_{\text{ntex}}(2) + m^c_{\text{exact}}(2) + m^c_{\text{ntlpm}}(1) + m^c_{\text{lpm}}(1) + m^c_{\text{fwh}}(5) = 544.03 \tag{4.58}$$

The actual measurements for each of the paths shows 162.21, 219.54, 356.51, and 440.53 cycles per packet, resulting in a relative error of 30.26 %, 31.79 %, 22.12 %, and 23.50 %, respectively. We note that the error always remains within the same range. In fact, for paths of higher complexity, the model becomes more accurate. This is due to the component models being more accurate for high values.

For this example, the worst-case path can be clearly identified as the path that traverses all match-action tables successfully, i.e., $\mathcal{P}^4_{\text{L3}}$. As there are no externs or other functionality that changes the program state based on previous processed packets, we can derive the worst-case packet, assuming full knowledge about table entries. Our manual measurement to determine the actual cost for the fourth part represents this measurement. Similar, an average case evaluation would evenly hit all of the paths, which can be generated in this scenario.

4.5 Model Limitations

The proposed modeling approach is target dependent, i.e., a model derived for one target platform will likely be inaccurate for other targets. E.g., a model derived for a software device might not be accurate for another target system using COTS hardware due to differences in the system's hardware components like the CPU. However, this is the reason for the highly automated model-first approach: if the details of the hardware change, the modeling framework can be automatically executed for the target platform again, updating the parameters for the derived component and path models.

As shown in Section 4.4.5, cost, be it measured in throughput, latency, or resources, caused by different components of a packet processing system, is not always strictly additive. Different targets or compilers might optimize the code, such that adding up individually measured components for path models might result in an overestimation. For instance, several actions in the processing pipeline that are stated individually in the P4 program and, therefore, also in the CFG of the program, might be merged by the compiler. For this example, the model will add up the cost n times, however, on the target it only occurs once. Consequently, the path model will provide an upper bound. Vice-versa, unknown side effects or unmodeled components cause the real performance to be even worse.

The measurement and modeling approach assumes that each packet during a measurement is processed individually. In particular, packets do not influence other packets by changing the state of the DuT. Furthermore, it is assumed that no packet blocks other packets. While this assumption might be true on a highly pipelined ASIC platform, packets influencing each other is more likely on software-based platforms.

The accuracy of the modeling framework is limited by the set of possible prototype functions Λ^*. Only the functions presented in Section 4.3.3 are currently included with the framework and can be used for the curve fitting approach. Extending this set, for instance, with higher degree polynomials, is possible, however, we deem it unlikely that such a high-degree polynomial would semantically fit the device behavior. Furthermore, for each function, removing certain free fitting parameters, or setting them to a fixed no-influence value, would improve the overall result. In this case, the function rank ψ would be lower, i.e., according to the AIC-based metric this function would be treated as less complex, and, therefore, be preferred over other more complex functions. For example, the function $\tau = p_0^* x^2 + p_1^* x + p_2^*$ has $\psi = 3$, however, adding the same function template while setting $p_0^* = 1$ would result in $\psi = 2$. Applying this approach to generate all possible functions with lower ψ for all current functions of Λ^* would increase the overall computation time.

Components that only allow measurements with a low number of data points will result in models with low accuracy. For instance, outliers will have a higher impact, resulting in overfitting. In such cases, alternative error metrics or higher error tolerances have to be used

4.6 Key Results

This chapter presents our approach for modeling the performance of P4-programmable data plane programs. The P4 DSL allows to derive the CFG of the program, which can be used for further analysis. We use this representation to analyze different components and features of the language in isolation. By measuring programs of increasing complexity in only one component, we can model and predict this components behavior. Later, full applications can be broken down into how often they contain these components. Thus, for each path, a performance model can be derived.

This approach is target-dependent, i.e., component models for one target platform are likely not applicable for other platforms. Therefore, a high degree of automation in deriving these models is required. We propose a measurement and modeling framework that provides a clear specification of each experiment. Thereby, a report for the whole experiment is generated, specifying the metrics, traffic parameters, P4 program, etc., enabling reproducibility. While the framework includes testbed- and target-dependent components, these are modularized, implementing the specification for the concrete testbed or target device. By integrating it with our pos testbed environment we achieve full automation, resulting in a model-first approach. Each data plane component is individually analyzed and, based on the resulting measurement artifacts, a performance model is derived automatically. Our curve fitting-based approach uses different metrics, which can be changed or customized as plugins. Further, the model detects events that change the performance behavior of the system, resulting in individual models for each part.

By applying our framework to the software-based t4p4s target we have highlighted the advantages of such an automated approach. Important components found across all P4 programs, e.g., the parser, header field modifications, and match-action tables, are analyzed. The automatically generated models can be used to detect regressions between different implementation versions [17] or can be compared to theoretical models, e.g., for the implementations of different table match types. Furthermore, by scaling to extreme values, limitations of the components and, therefore, the target architecture, can be discovered. For match-action tables using exact matches, we explained the automatically generated model with a manually derived resource model. As a result, we can model the relationship between the number of table entries and the cache sizes. In particular, when exceeding the L3 cache limit, entries that have to be fetched from main memory cause a huge performance penalty.

By evaluating many components, we identified, which statements of the P4 language specification have a significant influence on the performance. We argue that, in the case of P4, the focus of performance studies should be on the main component of P4 processing pipelines, the match-action tables. While other actions, like adding or removing bytes to the packet headers, has a measurable influence, the number of applied tables is the primary performance impacting factor, at least for a software target.

Combining these component models to model the performance of complete program paths has revealed that the resulting models overestimate between 20 and 30 percent. While each component can be modeled by itself, revealing influences related to such a component, the combined model does not include individual effects and the interplay of multiple components. Individual model inaccuracies add up, while compilers might optimize, essentially merging components together. Therefore, the path models only provide an approximation for the upper bound.

Using the path models for worst-case evaluation of a program can be challenging. While deriving the worst-case path is typically possible based on the path models, externs or other components that are either not known or not modeled add uncertainties. Even when, e.g., the worst-case path can be determined, reliably generating traffic to only match the isolated path is complex. For instance, if the path uses an extern that changes the state of the device, e.g., by inserting a new table entry through sending a digest to the control plane, packets might suddenly take another path. In all cases, this information cannot be derived just form the data plane. Only with data provided by the control plane a P4 data plane program receives its semantics, e.g., in the form of table entries.

Further, we argue, that not all metrics like throughput, latency, or resources, are of interest for all targets. For a software target, resources are virtually unlimited, wherefore modeling them is of low interest. Only when exceeding certain boundaries, e.g., for extreme numbers of table entries, local effects can be observed. Vice-versa, for hardware targets like the ASIC-based Tofino, throughput is guaranteed, but fitting the program is the challenge. Here, modeling the resource consumption should be prioritized.

4.7 Statement on Author's Contributions

The model for generic packet processing systems presented in Figure 4.2 and Equation 4.2 is based on the dissertation by Sebastian Gallenmüller. This model was significantly extended and applied to P4 programmable data planes by the author. Thereby, the notation was adjusted to fit the notation used throughout this thesis. Sections 4.1.2 and 4.1.5 are based on a collaboration between Dominik Scholz, Hasanin Harkous, Sebastian Gallenmüller, Henning Stubbe, Max Helm, Benedikt Jaeger, Nemanja Deric, Endri Goshi, Zikai Zhou, Wolfgang Kellerer, and Georg Carle [14]. These sections were extended and adjusted to fit the context of this thesis. Sections 4.1.1, 4.1.3, and 4.1.4, including Figures 4.3 and 4.4, were added for the purpose of this work.

The discussion on related work in Section 4.2 was performed by the author.

Section 4.3 is based on a joint work between Dominik Scholz, Hasanin Harkous, Sebastian Gallenmüller, Henning Stubbe, Max Helm, Benedikt Jaeger, Nemanja Deric, Endri Goshi, Zikai Zhou, Wolfgang Kellerer, and Georg Carle [14]. The measurement and modeling framework was implemented by the author. The framework uses a P4 program generator, which is based on an interdisciplinary project work by Henning Stubbe, co-supervised by the author. The author repurposed and extended the program generator for this work. The mathematical model presented in Section 4.3.3 was developed by the author. The notation was extended and unified compared to the original publication.

Section 4.4 is based on a collaboration between Dominik Scholz, Henning Stubbe, Sebastian Gallenmüller, and Georg Carle [6]. All measurements, figures, and models were performed and created by the author. Compared to the original work, all measurements and models were created with the framework for automated modeling introduced in Section 4.3. The analysis regarding the impact of different data types on data plane performance provided in Section 4.4.4 is based on profiling results reported in the Master's thesis of Maximilian Endrass, which was supervised by the author. The exact match model for increasing number of table entries in Section 4.4.4 is based on a publication by Dominik Scholz, Hasanin Harkous, Sebastian Gallenmüller, Henning Stubbe, Max Helm, Benedikt Jaeger, Nemanja Deric, Endri Goshi, Zikai Zhou, Wolfgang Kellerer, and Georg Carle [14]. The respective resource model for exact match-action tables was extended and improved by the author. The model is based on the same research project by Maximilian Endrass. Further, the t4p4s version used for all measurements in Section 4.4 includes changes provided Maximilian Endrass. The data plane programs used in Section 4.4.5 are based on t4p4s example programs and were modified by the author for the purpose of path modeling. Compared to the original publication, all notations were unified to match the notation used throughout this work. The author excluded a section of the original work about modeling the resources of an ASIC-based target, as it does not fit the focus on software targets in this thesis.

Chapter 5

Cryptographic Hashing in Programmable Data Planes

Secure and resilient networking protocols and applications are vital in modern network communication. While P4 introduces a standardized way for data plane programming, it lacks language building blocks for cryptographic mechanisms, including functions for encryption and hashing. These functions require complex, therefore, resource-intensive implementations [9], which can be highly target-dependent, or require the complete packet data as input. Therefore, cryptographic functions are currently considered language externs. Especially cryptographic hash functions have a wide range of potential use cases: secure communication including data integrity and authentication, challenge-response mechanisms, and robust hash-based data structures, among others. Thinking outside of regular, data center-focused applications, these mechanism become even more important in industrial applications like the automotive or aeronautical industries.

P4 allows to perform almost arbitrary computation in the data plane of different software and hardware devices. This opens the general question, which tasks should be performed in the data plane. While the cost for performing packet I/O is typically low in hardware devices, the cost for computation remains. These devices have limited hardware resources, restricting the total amount of computation per packet [175]. Further, complex computation typically comes at the cost of performance, for instance, reducing throughput or increasing latency.

In this chapter we answer the question, whether cryptographic hash functions can be implemented in P4 programmable data planes efficiently. We argue, why the availability of hash functions with cryptographic properties in programmable data planes is important. Then, we discuss how cryptographic functions can be integrated into P4 target platforms and what performance reductions and resource requirements can be expected. Lastly, we use our different hashing implementations in a case study and compare the flexibility, usability, and performance of implementing DDoS mitigation techniques in programmable data planes to conventional software packet processing frameworks.

5.1 Motivation

Section 5.1 is based on a collaboration between Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle [7]; and a joint work between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, and Georg Carle [8].

Hash functions are a vital element of network communication in the modern Internet. They play a fundamental role in many network applications, such as tasks like routing, consensus protocols [112], caching [176], packet sampling, heavy hitter detection [90], [177], or pattern recognition in deep packet inspection [178], [179]. A typical scenario is to provide a deterministic method for distributing flows over a number of fixed outputs. This can be used to select the outgoing queue in scheduling implementations, the concrete link from a set of aggregated links, or the next hop for load balancing. Second, hash functions can be used to implement data structures found in networking applications. As they can be calculated efficiently both in software and in hardware, while requiring a low memory footprint, hash functions excel in structures like simple hash tables, count-min sketches, or more complex bloom filters. Lastly, hash functions are required to provide secure communication in the form of authentication and data integrity. For instance, such security requirements are supposed to become even more important for the next generation 6G standard [180].

Thereby, different hash functions have varying properties. For the sake of this work, we split hash functions into two groups: regular hash functions and hash functions with cryptographic properties. Regular hash functions provide a deterministic way to map an input of arbitrary length to an output of fixed size. Cryptographic hash functions need to also withstand cryptanalytic attacks [181]. To classify as cryptographic hash function, they need to provide the three properties of pre-image resistance, second pre-image resistance, and collision resistance [181].

5.1.1 The Need for Cryptographic Hashing

While many applications require hash functions, often hash functions without a cryptographic security level are sufficient. We identified two use cases for hash functions with cryptographic properties: to improve the reliability of data structures susceptible to attacks; and to provide secure communication, using authentication or data integrity.

While hash-based data structures can be built from regular hash functions, doing so can create attack vectors on the data structure. Various attacks have been proposed on poorly implemented hash-based data structures. A prime example are poorly implemented hash tables: if an attacker is able to craft packets in a certain way and, therefore, also the input to the hash function, the hash table can degenerate to a linked list. This can lead to worse memory footprint or high CPU usage [182]. While this can be counteracted with chained hash tables, collisions can still lead to data loss or inaccurate sampling. A family of hash functions often used for such data structures due to their simplicity and efficiency are cyclic redundancy checks (CRCs). However, CRC is susceptible to the birthday paradoxon and can be attacked, for instance, when used for packet sampling [183]. Therefore, a hash function with high collision resistance, reducing hash collisions, is vital when the hash-based structure is attackable from the outside. For these use cases, keyed pseudo-random functions and cryptographically-strong random number generators are recommended [182], [183].

5.1.2 Usage in Networking Protocols

Cryptographic hash functions are found in various network protocols that are based on or require cryptographically secure communication [184]. For instance, message authentication codes (MACs), which provide data integrity and authenticity for packet content. They can be found in numerous networking protocols like IEEE 802.1AE MACsec, IPsec, or TLS, in the form of hash-based message authentication codes (HMACs). While TLS is a widely used protocol in the Internet, implementing TLS in the data plane is challenging as it requires a functioning implementation of TCP.

The IEEE 802.1AE MACsec standard provides data confidentiality and integrity on the data link layer. For this, it uses a security tag and a MAC. These properties on the data link layer are especially interesting for industrial use cases, including automotive [185], [186] and aeronautical applications [187]. While programmable data planes are interesting for these industries as they allow rapid prototyping, they also have strict security requirements [18]. For instance, airplanes may include several Ethernet-based networks, e.g., for cabin communication or entertainment, using specialized protocols like Avionics Full-Duplex Switched Ethernet [100]. However, using Ethernet-based communication for critical infrastructure requires the authentication of Ethernet frames and switches and data integrity.

MACsec and IPsec implementations for P4 programmable data planes have been proposed, however, the authors note the lack of support for cryptographic hash functions, especially for hardware targets [188], [189].

Other use cases are digital signatures or challenge-response protocols. These use token or cookie mechanisms to either prove the possession of an authentication token or to encode state that is being exchanged. One example are TCP SYN cookies [190], which are calculated for each incoming TCP SYN packet during an ongoing attack and, therefore, have to be efficiently generated and verified.

Other approaches try to establish new functionality in the data plane. One example is the SPINE [191] framework for surveillance protection. Datta et al. provide a prototype for the PISA architecture, using the pseudo-cryptographic SipHash hash function due to its high performance and low memory footprint [191].

Supporting hash functions with strong cryptographic properties will be a key enabler to allow offloading these network protocols to the data plane.

5.2 Feasibility and Integration of Hashing with Cryptographic Properties

Section 5.2 is based on a collaboration between Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle [7].

A wider set of hash functions with cryptographic properties may be beneficial for a variety of applications. We want to address two classes of secure and advanced applications in the data plane: First, resilience to hash collisions can be improved for programs that utilize hash-based data structures. High susceptibility to hash collisions can create attack vectors, leading to poor resource usage or DoSs [192]. Second, integrity protection, which is typically implemented using a MAC or HMAC with cryptographic hash functions, for instance, for digital signatures or challenge-response protocols. These are essential for secure communication not only on the Internet, but also in industrial networks.

We argue for the benefits of including hash functions with cryptographic properties in P4 platforms. In this section, we present our prototype implementations for and evaluation of three different P4 software and hardware target platforms: the t4p4s software platform, the Netronome Agilio NFP-4000 SmartNIC, and the NetFPGA SUME.

5.2.1 Related Work

We analyze the current usage and existence of hash functions in P4 data planes with regard to security and resilience. Further, we discuss recommended hash functions and their properties for networking applications. Lastly, we present a brief discussion of hash function implementations in hardware.

Hashing in P4 Applications

Use of hash functions for networking applications implemented in the P4 data plane can be found in various work. Ghasemi et al. [193] investigate performance diagnostic of TCP with Dapper, using standard 5-tuple hashes. Zaoxing et al. [177] propose UnivMon for network flow monitoring based on a sketch data structure where multiple, pairwise-independent hash functions are used. Cidon et al. [194] propose AppSwitch, a cache for key-value storage using hashes of keys. Sivaraman et al. [90] introduce HashPipe for heavy-hitter detection, using a pipeline of hash tables, which retain counters for heavy flows while being memory efficient. Finally, Kucera et al. [91] also address heavy-hitter detection using Elastic Trie, a novel trie-based data structure. The mentioned works either do not detail the hash algorithm used, or make use of CRC32 as hash function, making them potentially vulnerable to security attacks. Only Ghasemi et al. [193] explicitly describe the strategy used to deal with hash collisions. They use a hash chaining technique combining the hashed value and the TCP sequence number. Depending on the workload and the level of security required, these applications may benefit from the usage of cryptographic hash functions to minimize the impact of hash collisions.

SPINE [191] is a framework for surveillance protection proposed by Datta et al. They encrypt header fields containing sensible data, for instance, IPv4 addresses, and store the result in IPv6 headers. Datta et al. use SipHash for performance reasons, noting the constrained memory resources available on P4 platforms. Hauser et al. propose P4-MACsec [188] for the automation of MACsec deployments by shifting the MACsec implementation entirely to the data plane of P4 targets. They implement prototypes for the bmv2 model and the NetFPGA SUME, but the solution for the latter was not feasible due to problems encountered during integration of the hashing externs [188]. Hauser et al. also propose a similar approach for IPsec [189], but their prototype implementation for the NetFPGA SUME has the same restrictions as their MACsec proposal. They also developed an ASIC prototype, however, all cryptographic processing is performed by a CPU-based controller. The ASIC neither offers cryptographic algorithms nor allows adding new functionality as externs. This limits performance and functionality [189].

We observe that current P4 targets neither support secure nor resilient communication. Therefore, applications and protocols, which require MACs, authentication, or hashing structures that are resilient against attacks such as DoS, cannot be implemented.

Hash Functions for Networking Applications

Various work already evaluated the suitability of hash algorithms for network packets. Molina et al. [195] and Henke et al. [196] evaluate several different functions, with a focus on packet sampling. Both works highlight that CRC32 is not recommended due to its linear dependency between hash input and hash value, making it vulnerable to bias and security attacks. They recommend BOB [197] as hash algorithm in non-adversarial scenarios due to its performance and avalanche properties. Another non-cryptographic hash function with fast calculation on CPU systems is MurMurHash [198]. While it only uses simple mathematical operations, it has low collision rates [199].

Due to the use of relatively small messages in packet processing, the choice of a hash function with cryptographic properties for efficient processing is not straightforward. While the SHA2 family of hash functions is a strong candidate regarding cryptographic features and security, these functions were not designed with good performance for small inputs.

Aumasson et al. propose the pseudo-cryptographic SipHash [200] function. It is optimized for performance when using short inputs and is used as hash function in many programming languages, for hash table implementations, and DoS protection [200]. We use SipHash-2-4 throughout this work, consisting of two rounds of computation per message block and four finalization rounds. HalfSipHash uses the same scheme as SipHash, but produces an easier to brute-force 32 bit output hash.

Another popular candidate is the BLAKE2 family [201], designed for good performance for small inputs. It provides similar cryptographic security compared to SHA3 [201].

Poly1305-AES [202] was proposed by Bernstein et al. as an algorithm for MACs in IPsec [203] and TLS [204]. The cryptographic properties of the hash function are similar to AES and can be computed efficiently on software systems [202].

Hardware Implementations of Hash Functions

Regarding hardware implementations of non-cryptographic hash algorithms for networking applications, Hua et al. [205] evaluated 18 different functions. They propose a family of hash functions achieving good properties in terms of hashing at a reduced cost regarding hardware footprint and cost per cycle.

Pseudo-cryptographic and cryptographic hash functions are available on FPGA and even ASIC platforms. Over the years, the performance of SHA-based and other hash functions in ASICs has been improved [206], [207]. Lately, large-scale experimental prototypes such as Bitcoin have shown that these cryptographic operations can be implemented in hardware with high-performance. In fact, the performance of hashing operations in ASICs has increased such that it can threaten the decentralization of blockchain-based networks [208].

5.2.2 Interface for Hashing in P4

The interface for using hash functions has changed between the initial version of P4, $P4_{14}$, and the current specification $P4_{16}$. Initially, hashes could be calculated using so called *calculated fields* or *direct function calls*. Listing 5.1 shows the usage when hashing the 5-tuple, for instance, to identify flows, commonly found in data plane applications. The programmer could define a list of values that should be used as input for the hash function. Later, the field list calculation defines the input, hash function algorithm and output width. This calculation can then be used to modify arbitrary fields.

```
// define field_list
field_list five_tuple_list {
    ip4.src;
    ip4.dst;
    l4.src;
    l4.dst;
    ip4.protocol;
}

// define hash calculation
field_list_calculation five_tuple_hash {
    input {
        five_tuple_list;
    }
    algorithm : crc32;
    output_width : 32;
}

// use hash calculation
action some_action() {
    modify_field_with_hash_based_offset(some_field,
                                        offset,
                                        five_tuple_hash,
                                        32);
}
```

Listing 5.1: Hash function interface in $P4_{14}$. Definitions taken from [209]

In $P4_{16}$, hash algorithms can be accessed via externs, i.e., standard function calls of external libraries. The interface for the extern and supported algorithms depends on the concrete architecture model shown in Listing 5.2. For instance, the v1model.p4 offers the generic *hash* function with a predefined set of algorithms, including identity, CRC32, checksum, and random functions. The hash function's parameters include the hash algorithm to use, the output field, minimum and maximum values, and a list of parsed header or metadata fields to be used as input. Compared to P4_{14}, the invocation of the extern more closely resembles the invocation of a standard C function call. Using a different architecture model, e.g., the ubpf_model, the interface for the hash function changes.

```
// v1model architecture
extern void hash<O, T, D, M>(out O result,
                             in HashAlgorithm algo,
                             in T base,
                             in D data,
                             in M max);

// sample invocation using 5-tuple as input
action some_action() {
    hash(result_variable,
         crc32,
         32w0,
         { h.ip4.src, h.ip4.dst,
           h.tcp.src, h.tcp.dst
           h.ip4.protocol },
         max_value);
}

// ubpf architecture
extern void hash<D>(out bit<32> result,
                    in HashAlgorithm algo,
                    in D data);

// PSA architecture
extern Hash<O> {
    Hash(PSA_HashAlgorithm_t algo);
    O get_hash<D>(in D data);
    O get_hash<T, D>(in T base,
                     in D data,
                     in T max);
}
```

Listing 5.2: Hash function interfaces in P4_{16}. Definitions taken from [210] and https://github.com/p4lang/p4c/tree/master/p4include

The PSA defines five different hash functions in addition to the identity function that should be available for switch data planes: four variations of CRC and the 16 bit one's complement used for IP, TCP, and UDP checksum calculation.

While the hash functions defined by these architecture models may serve as useful hash functions in common networking applications [211], none of them provides any cryptographic properties. Therefore, the standard set of hash functions available in P4 makes it a limited platform for applications that require or benefit from security properties. The extern interfaces are specifically intended to add custom functionality to individual target platforms. Extending P4 and its software and hardware targets with cryptographic algorithms enables offloading of secure applications to the data plane. As it is the current specification of the language, we limit our integration of hash functions to P4_{16}.

5.2.3 Choice of Hash Function

Operating on Ethernet frames, the input for the hash function will typically be less than 1500 B. However, when hashing full frames, this can be as few as 64 B for minimum-sized Ethernet frames. When hashing only parts of the packet headers it can be even less, e.g., 13 B for the typical five tuple consisting of IPv4 addresses, protocol field, and transport layer ports. Consequently, the hash function must retain its cryptographic properties even for short inputs.

Algorithm	Cryptographic Hash	Output [B]	Implementation
Checksum	no	16	DPDK (v19.02)
CRC32	no	32	DPDK (v19.02)
MurmurHash3	no	32/128	[198]
Half-SipHash-2-4	pseudo (MAC)	32	[200]
SipHash-2-4	pseudo (MAC)	64	[200]
Poly1305-AES	yes (MAC)	128	[212]
BLAKE2s	yes	8-256	OpenSSL (v1.1.0)
BLAKE2b	yes	8-512	OpenSSL (v1.1.0)
SHA256	yes	256	OpenSSL (v1.1.0)
SHA512	yes	512	OpenSSL (v1.1.0)
SHA3-256	yes	256	OpenSSL (v1.1.0)
SHA3-512	yes	512	OpenSSL (v1.1.0)
HMAC-BLAKE2s	yes (HMAC)	8-256	OpenSSL (v1.1.0)
HMAC-BLAKE2b	yes (HMAC)	8-512	OpenSSL (v1.1.0)
HMAC-SHA256	yes (HMAC)	256	OpenSSL (v1.1.0)
HMAC-SHA512	yes (HMAC)	512	OpenSSL (v1.1.0)
HMAC-SHA3-256	yes (HMAC)	256	OpenSSL (v1.1.0)
HMAC-SHA3-512	yes (HMAC)	512	OpenSSL (v1.1.0)

Table 5.1: Properties of investigated hash functions

We assume use cases where the output of the hash function is stored in the packet. Consequently, a hash function producing short outputs, e.g., below 100 B, will typically be sufficient.

The performance of the hash function should be high, i.e., in the worst case it needs to perform at line-rate using 64 B packets. To improve portability of the P4 program, the hash function implementation should be available for different platforms, including software, FPGA, and ASIC.

We investigate software implementations for the set of non-cryptographic, pseudo-cryptographic and cryptographic hash functions shown in Table 5.1. The non-cryptographic functions are included to serve as a baseline comparison. The SipHash and HalfSipHash family of hash functions, optimized for hashing on 64 bit and 32 bit CPU architectures, respectively, are only pseudo-cryptographic hash functions. However, they are designed to be used as MACs using short inputs, which is sufficient for our use case. Similarly, Poly1305-AES is only intended to be used as MAC. While in this work we focus on applying a single hash operation, we also include HMAC implementations for the mentioned OpenSSL implementations for the sake of providing a better picture regarding performance levels. We want to note, that depending on use case, an HMAC scheme might be necessary to achieve the required cryptographic properties.

Metrics for Hash Functions

We differentiate cryptographic and performance metrics when comparing cryptographic hash functions. Cryptographic properties of a hash function may depend on a defined input length, which is relevant in our scenario as we hash, e.g., a typical 5-tuple or the complete packet payload. Similarly, the type of input data, comparing the entropy of passwords with that of

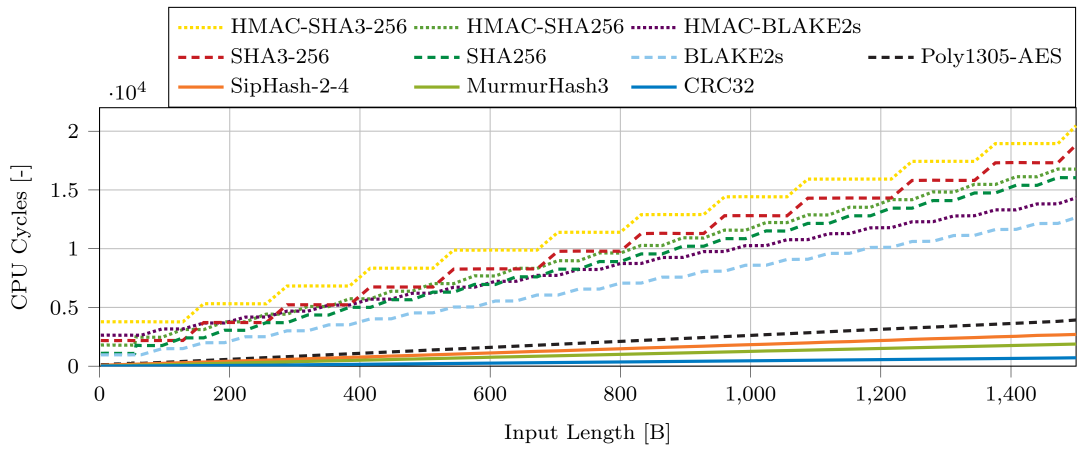


Figure 5.1: Performance benchmark results of selected hash functions on COTS CPU

random data, is of relevance. These properties might influence, for instance, the function's collision resistance. Finally, different applications may have different constraints regarding the length of the produced output hash. While a short HMAC included in an Ethernet frame causes only minor packet overhead, it can negatively impact its effectiveness.

Several performance metrics depending on the requirements of the application and capabilities of the target software or hardware platform are of relevance when choosing a hash function. In high-performance applications, the performance of the hash function in terms of latency and processing time, e.g., measured using clock cycles, is an important characteristic. When implementing the hash function, its memory footprint and, when implemented in hardware, resources of the hardware required, e.g., logic elements and registers for an FPGA, have to be considered. As an example, a VHDL SipHash implementation takes 10 % of available logic cores on a Cyclone II FPGA [213]. Whether the algorithm can take advantage of parts of the hardware target's architecture, e.g., embedded memory blocks, greatly influences the achieved performance [214].

In this work, we focus on the performance aspects of the introduced cryptographic hash functions. We refer to related work regarding their cryptographic properties and cryptanalysis.

Hash Function Performance

The cost to calculate a (cryptographic) hash for input data of m bytes consists if two parts: a fixed part to initialize the calculation and a cost per byte of input data. Consequently, the fixed cost is amortized and less significant for large m.

Using an artifical benchmark, we measure the fixed and per-byte costs of the introduced hash functions on a COTS server system equipped with an Intel Xeon E5-2640 v2 CPU clocked at 2 GHz. We calculate the hash for input data lengths from 2 B to 1500 B, which are relevant for our use case. We measure the execution time in CPU cycles using the timestamp counter. This process is repeated 1 000 000 times for every hash function.

Algorithm	Cycles/B	Fixed Cycles	$m = 64\,\text{B}$	$m = 1500\,\text{B}$
Checksum	0.39	5.25	29.86	594.89
CRC32	0.47	0.00	17.20	712.87
MurmurHash3	1.25	7.79	82.48	1881.07
Half-SipHash-2-4	3.52	67.65	293.46	5345.67
SipHash-2-4	1.76	71.93	184.16	2700.92
Poly1305-AES	2.51	106.21	237.31	3929.02
BLAKE2s	7.85	682.72	968.16	12 644.99
BLAKE2b	4.81	1029.96	1244.96	8267.09
SHA256	10.21	876.31	1755.84	16 038.59
SHA512	7.10	1216.02	1464.20	11 614.31
SHA3-256	10.89	1729.56	2179.94	18 828.76
SHA3-512	20.76	1506.47	2179.72	32 076.32
HMAC-BLAKE2s	7.85	2355.07	2632.07	14 324.56
HMAC-BLAKE2b	4.81	3179.02	3385.79	10 398.10
HMAC-SHA512	7.21	2320.12	2585.61	12 845.54
HMAC-SHA256	10.23	1584.61	2467.06	16 769.44
HMAC-SHA3-256	10.91	3326.17	3782.52	20 462.48
HMAC-SHA3-512	20.75	3097.79	3763.00	33 655.29

Table 5.2: Hash function performance benchmark on COTS CPU

We want to note that the performance figures are highly dependent on the concrete CPU used. However, while the exact numbers might vary, different performance levels separating individual functions will remain the same.

Figure 5.1 displays the measured CPU cycles for different input lengths for a selection of hash functions. In general, we can split the hash functions into two different groups of performance levels: non-cryptographic and pseudo-cryptographic hash functions, including Poly1305-AES; and SHA-based cryptographic hash functions and HMAC schemes. In the former group, SipHash and the cryptographic Poly1305-AES stand out for having a performance comparable to the non-cryptographic hash functions, while being suited for MACs. SHA-based functions do not show a linear increase in CPU cycles. As they operate on certain block sizes of input data, they display a typical step-wise increase. HMAC schemes further increase the fixed part compared to the respective regular hash function used for the scheme.

Table 5.2 lists the results for all investigated hash functions on our software DuT. Using linear regression we calculate the fixed and per-byte costs for each hash function. Furthermore, the table displays the two extrema for our use case: hashing minimum and maximum sized packets with 64 B and 1500 B, respectively.

The number of fixed CPU cycles per packet reported in Table 5.2 matches the data presented in Figure 5.1: SipHash and Poly1305-AES are better suited for processing small input data such as network packets than SHA-based cryptographic functions. The latter have significantly larger fixed and per-byte costs, making them unsuited for the intended purpose of this work for software targets. HMAC schemes add up to 1600 of additional additional CPU cycles.

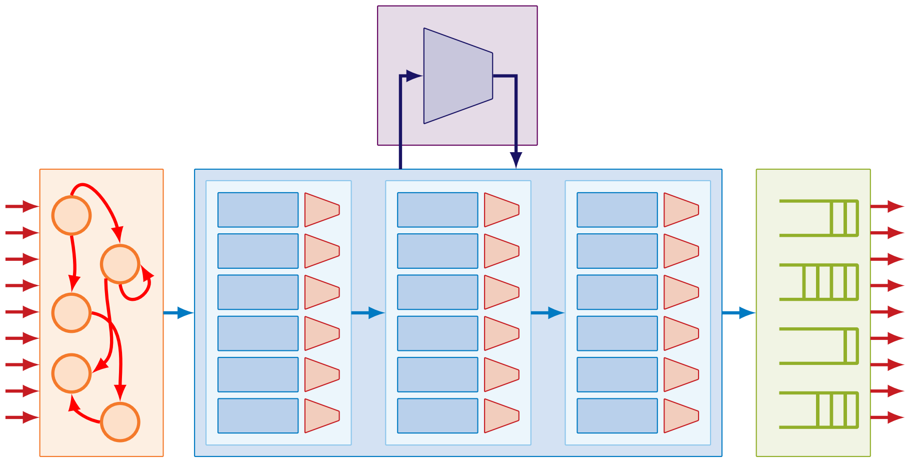


Figure 5.2: Hashing functionality as extern

A relevant metric to compare the values of Table 5.2 is the CPU cycle budget, which can be allocated per packet. To process line-rate of 14.88 Mpps for 10 GbE with 64 B packets on a single CPU core clocked at 2.00 GHz, the processing for each packet must be completed within 134 CPU cycles. On this particular CPU not even SipHash is able to do so within the cycle budget. However, using additional CPU cores or a more potent or newer-generation CPU, the pseudo-cryptographic SipHash function is able to meet our requirements. For the remainder of this work, we will focus on SipHash as hash function as it combines fast execution times with short in- and outputs. Furthermore, implementations for FPGA platforms exist.

5.2.4 Integration Strategies

As introduced in Section 5.2.2, hash functions are regarded as externs in P4. While this is the intended and preferred way of integrating additional functionality with P4 targets, the feasability of integrating hashing as extern can be limited for a given target platform. We will discuss limiting aspects and additional integration approaches in the following section.

As Extern

Figure 5.2 displays the integration of the hash function as extern that can be called from within a P4 pipeline. Thereby, P4 does not specify where or how the extern is implemented. Consequently, the implementation can be in software or hardware, close to the data plane or close to the control plane.

The challenge with any integration is to support operations at line-rate. Further, when permitting hashing of packet payload, i.e., the complete packet, all data needs to be sent to the extern function. While this might be trivial on software targets, aside from cache effects, streaming this amount of data seamlessly to an extern can be challenging on hardware targets. The reasons include the challenge to provide the necessary bandwidth for the programmable connection, e.g., a bus system or wires of fixed size, between P4 pipeline and extern location. Further,

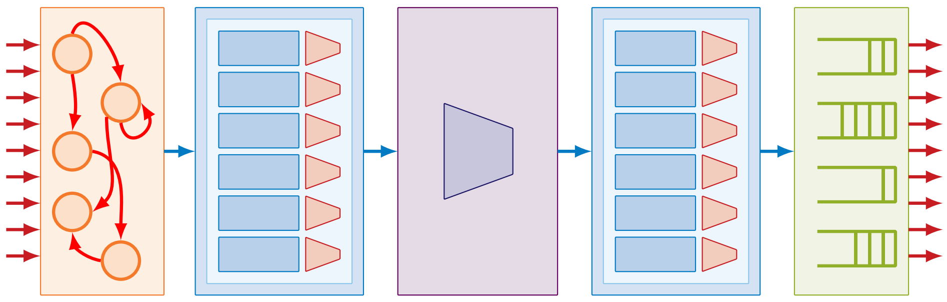


Figure 5.3: Hashing functionality integrated with architecture model

the resources required for the extern implementation and connection might limit the remaining program complexity or the streaming of the data causes high latency.

As Architecture Model Extension

To tackle the mentioned problems, we propose a tradeoff between flexibility and hardware resources: integrating the hash function as separate part of the architecture model as shown in Figure 5.3. Instead of accessing it as extern, the P4 pipeline does not have direct access to the hash function. However, either before or after the pipeline, the full packet data including metadata is sent to the hashing module by default. The hashing module performs the necessary calculation and stores the result in a field of either the packet or metadata. Thereby, per-packet metadata, which traverses alongside all modules of the P4 data plane, can be used to enable programmable control over the hashing module, determining input data, hash function, and output field from within the P4 pipeline.

In this scheme, the P4 pipeline can be before, after, or before and after the hashing module. If the pipeline is only before the hash function, the resulting hash cannot be used by the P4 program for packet modifications or forwarding decisions. Vice-versa, the hashing module cannot be programmably configured per packet if the P4 pipeline is only located after the hash function. If the target is limited to only one pipeline, recirculation of packets can be used to simulate having a pipeline before and after the hashing module. The metadata in the hashing module could also be used to implement more complex HMAC schemes, e.g., providing cryptographic key material via metadata.

While this scheme also requires resources to integrate the hashing module with the data plane, these resources are fixed already during manufacturing. Using them within a P4 program will not take away resources for other constructs.

The disadvantages are that every packet traverses the hashing module, which increases the overall processing path and, therefore, the latency. Further, only one hash can be computed per packet (except recirculation), limiting the usability and flexibility of the hash function in the P4 program. To combat this, the architecture model can be extended to include additional hashing modules and P4 pipelines.

Further changes to the architecture model can improve this concept. For instance, using a traffic manager, packets could be selectively steered into or around the hashing module, depending on whether a hash needs to be calculated or not. This would reduce latency for packets not requiring a hash.

Using Basic P4 Language Components

Instead of using extern functionality, the hash function could in theory be programmed using only native P4 constructs, including match-action table lookups and basic mathematical operations. So far, such an approach has only been proposed for encryption using AES [215]. The downside is the extensive use of the target's native P4 resources required, for instance, for match-action tables. Further, the complex operations, including multiple table lookups and calculations, reduce the performance.

As External Node

Lastly, the hash functionality can be offered outside of the current data plane by another node. Although, strictly speaking, it is not an integration of the hash function with the P4 data plane, it is noteworthy as it can be achieved by altering the network topology or using control plane logic. The downsides of this approach are the latency penalty for sending the packets that require a hash calculation to another node and the added network and controller complexity.

5.2.5 Prototype Implementations

We have extended three different P4 target platforms with externs calculating cryptographic hashes. Each platform has its own approach how P4 externs can be added. We focus on a small selection of hash functions to reduce implementation overhead. As it best fits our use cases and shows satisfying performance, we implemented the SipHash function on all targets.

Hash Externs in t4p4s

t4p4s only supports the TCP/IP checksum calculation as hash algorithm, which is implemented using the `rte_raw_cksum` function from DPDK. We extended this target with a selection of the open-source non-cryptographic, pseudo-cryptographic, and cryptographic hash function implementations listed in Table 5.1: MurmurHash3, SipHash-2-4, Poly1305-AES, BLAKE2s, SHA256, HMAC-BLAKE2s, and HMAC-SHA256. Furthermore, we added an SSE4.2-accelerated, non-cryptographic CRC32 function based on functionality provided by DPDK.

As t4p4s is an open-source software target, integrating these functions as extern libraries is straight-forward.

Hash Externs in the NFP-4000

We have also integrated hash functions as externs for the 10G NFP-4000 Agilio SmartNIC NPU. Out-of-the-box, the NPU supports only the non-cryptographic CRC32 and Checksum functions. The SmartNIC allows implementing P4 externs and other functionality in Micro-C, a variation of C used to program the NPU's processing cores. The compiler then inlines the externs into the compiled P4 program. For the already mentioned reasons, we have implemented the SipHash-2-4 function in Micro-C, calculating a hash for the full payload of the Ethernet frame. Another factor for choosing this pseudo-cryptographic hash function over, for instance, SHA3 is its significantly reduced code complexity. We want to note that some NICs of the NFP-4000 family feature a hardware crypto security accelerator supporting SHA1 and SHA2 hash functions. However, the accelerator module was not available on our device, wherefore we opted for the CPU-optimized SipHash instead.

Extended Architecture Model for the NetFPGA SUME

The extern interface of P4→NetFPGA only allows data widths of approximately 600 B as in- or output. While this is a restriction of the SDNet compiler, even for smaller widths the compiler was unable to resolve resource congestion, resulting in timing violations of the design. Further, the extern interface is designed without a possibility to indicate a backlog, i.e., it has to accept new data for the extern with every clock cycle. Consequently, we changed the NetFPGA's architecture model to integrate the hash function as additional module. We integrated it after the P4 pipeline, i.e., in the egress path, to allow P4 metadata being accessed by the hashing module. We chose two open-source IP cores for hash function implementations: a SipHash-2-4[1] and a SHA3-512[2].

For both hardware platforms, the NFP-4000 and the NetFPGA SUME, we did not implement HMAC schemes. Consequently, we only perform a single hash operation, excluding any cryptographic key material required for HMACs.

5.2.6 Evaluation

We evaluate the feasibility of hash functions with cryptographic properties in the data plane with a performance evaluation. We use the two-node setup presented in Figure 3.2. The server acting as the DuT is equipped with an Intel Xeon CPU E5-2620 v3 (Broadwell) at 2.40 GHz and either an Intel X540 network card, Netronome NFP-4000 SmartNIC, or the NetFPGA SUME. For measurements performed using the CPU target, all traffic is pinned to one CPU core. For the use case of communication integrity and authentication, we evaluate the hashing of complete packets for our prototype implementations and selected hash functions. The P4 program is a simple L2 forwarder that additionally calculates hashes based on the complete Ethernet frames, representing the worst case for the actual hash calculation. We focus on hashing complete Ethernet frames instead of only selected fields, e.g., the 5-tuple, as in the latter case the setup cost for the hash function will outweigh the per-byte cost and further reduce maximum performance. For each platform, we perform a baseline measurement, where the P4 program is only an L2 forwarder, performing no hashing operations.

Throughput

Results for maximum throughput are presented in Table 5.3 and for a selection of hash functions in Figure 5.4. Independent of the packet size, all three target platforms reach 10 Gbit/s in the baseline scenario, with the exception for minimum-sized packets on the CPU target. No platform

[1] SipHash IP Core: `https://github.com/secworks/siphash`

[2] SHA3-512 (KECCAK) IP Core: `https://github.com/freecores/sha3`

Algorithm	64 B	96 B	128 B	512 B	1024 B	1500 B
t4p4s						
Baseline	91.2	100.0	100.0	100.0	100.0	100.0
Checksum	51.0	67.1	84.4	100.0	100.0	100.0
CRC32	54.4	73.4	90.6	100.0	100.0	100.0
MurmurHash3	46.6	59.1	69.8	100.0	100.0	100.0
SipHash-2-4	37.9	46.6	57.5	100.0	100.0	100.0
Poly1305-AES	30.0	36.8	42.6	71.5	82.0	85.5
BLAKE2S	9.2	10.3	13.3	23.6	28.2	29.5
SHA256	8.9	9.9	12.6	22.5	26.6	27.8
HMAC-BLAKE2S	5.2	6.5	8.2	18.2	23.8	26.0
HMAC-SHA256	6.4	7.5	9.6	19.4	24.4	26.0
NFP-4000						
Baseline	100.0	100.0	100.0	100.0	100.0	100.0
CRC32	100.0	100.0	100.0	100.0	100.0	100.0
Checksum	100.0	100.0	100.0	100.0	100.0	100.0
SipHash-2-4	75.6	80.7	91.6	99.2	10^{-6}	10^{-6}
NetFPGA SUME						
Baseline	100.0	100.0	100.0	100.0	100.0	100.0
SipHash-2-4	42.0	42.2	42.3	42.6	42.6	42.5
SHA3-512	48.2	42.5	54.3	65.0	71.8	76.0

Table 5.3: Throughput for hashing full Ethernet frames of different sizes in percent, relative to 10 GbE line-rate

is able to reach line-rate for minimum-sized 64 B packets when calculating the hash using a hash function with cryptographic properties or intended for MACs. With a maximum performance of 75 % compared to the baseline, the best results are achieved by the NFP-4000 SmartNIC using SipHash-2-4. However, despite high throughput for packet sizes up to 900 B, performance degrades rapidly for even larger packets. This behavior can be explained by the SmartNIC's RAM architecture [216] shown in Table 5.4. Only when processing less than 900 B of packet payload the data is stored in buffers residing in the fast memory regions. Exceeding this packet size, data is stored in 10 to 100 times slower shared RAM. The slower access times cause packet loss, resulting in a throughput of only approximately 10^{-6} % of line-rate. When using the non-cryptographic hash functions CRC32 and Checksum as hash algorithms, the NPU can hash packets at line-rate regardless of packet size.

Using SipHash-2-4, the NetFPGA SUME shows almost constant maximum throughput of approximately 42 % line-rate. This is due to the maximum throughput achieved by the SipHash-2-4 IP core, processing a theoretical maximum of 21.33 bit/clockcycle. The SHA3-512 IP core is clocked slower compared to the rest of the P4→NetFPGA pipeline, wherefore this core has to be placed in a separate clock domain. Despite this imperfect intergation, its higher per-cycle throughput results in superior performance for all packet sizes. It reaches a maximum of 76 % line-rate for 1500 B packets. We want to note that the performance figures of the NetFPGA

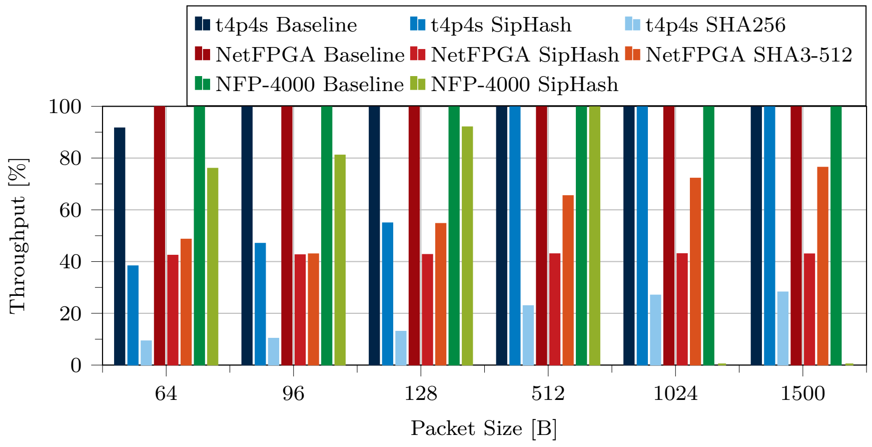


Figure 5.4: Throughput when using SipHash and SHA512-based functions for hashing full Ethernet frames, relative to 10 GbE line-rate

Memory type	Size [kB]	Access delay [cycles]
Local Memory	4	1 - 3
CLS	64	20 - 50
CTM (shared)	256	50 - 100
IMEM (shared)	2 × 4096	150 - 250
EMEM (shared)	3 × 3072	150 - 500

Table 5.4: Memory hierarchy of the NFP-4000 architecture (cf. Wray [216])

SUME prototype platform are limited because of our choice of open-source IP cores for the hash function implementations. Using commercial IP cores, with improved per-cycle throughput and ideal clock frequency, higher maximum throughput can be achieved.

The worst performance is shown by the CPU target, which is unable to reach line-rate for small packet sizes in the baseline scenario. The performance of the different algorithms on the CPU target correlate with the latency results presented in Table 5.2. Compared to the baseline, maximum throughput is more than halved for small packet sizes when using SipHash-2-4. Only for packets larger than 410 B line-rate is reached. Compared to Table 5.2, the ranking of the different algorithms changes as the packet size increases, mainly due to different initialization costs. This becomes apparent when looking at the results for Poly1305-AES, as it shows significantly increased performance for larger packet sizes due to its low per-byte cost. Nevertheless, the data shows that most of the hash functions are able to process packets at a line-rate of 1 Gbit/s, regardless of packet size. Due to the large number of fixed cycles per packet, SHA256, especially when used in HMAC mode, processes less than 7 % line-rate for minimum-sized packets and even for large packets is unable to reach line-rate.

Latency

Median latency figures for the NetFPGA SUME target are shown in Figure 5.5 for a throughput of 2.5 Gbit/s. Overall, latency increases linearly with packet size. As the SHA3-512 IP core

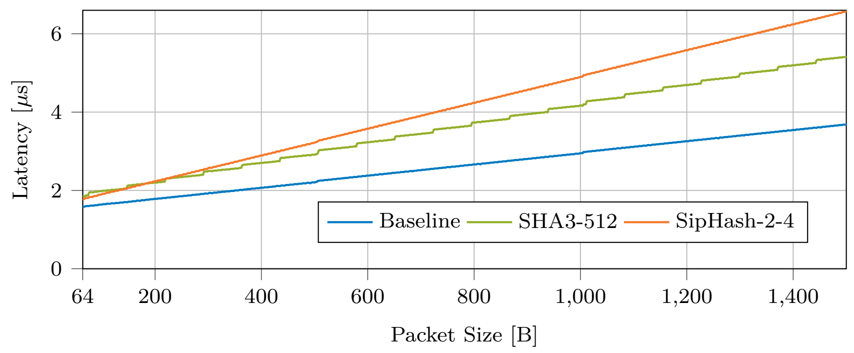


Figure 5.5: Median latency for NetFPGA SUME at 2.5 Gbit/s (taken from [7])

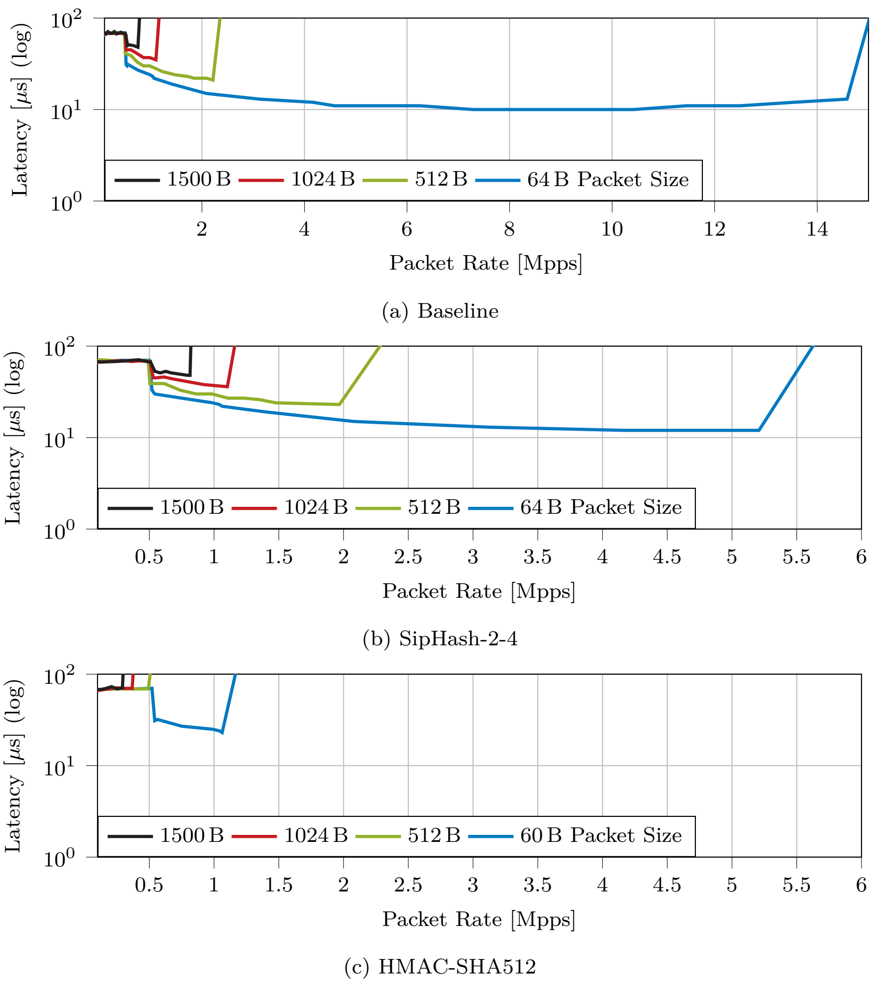
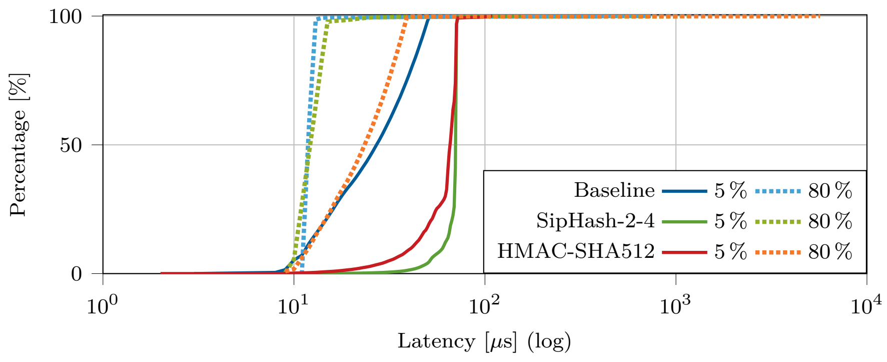
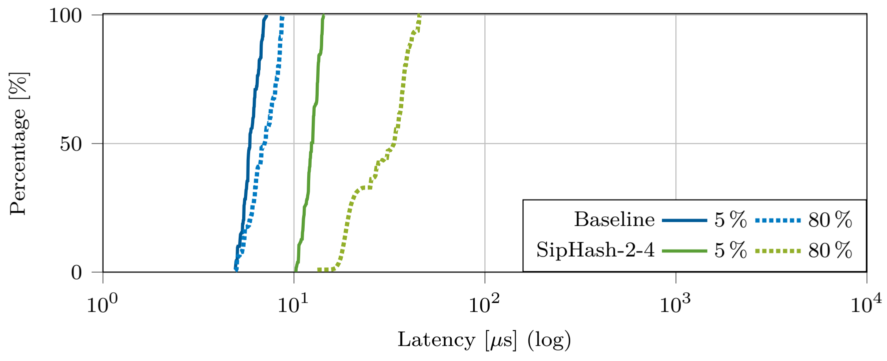


Figure 5.6: Median latency for t4p4s-based CPU target



(a) t4p4s



(b) NFP-4000

Figure 5.7: Latency distribution at 5 % and 80 % of respective maximum throughput using 64 B packets

operates on 72 B data blocks, the throughput increase is non-monotonic. While this is also true for the SipHash-2-4 hash implementation, due to the block size of 8 B the discontinuities are not noticable. Independent of the packet size, the measured latency values do not differ by more than 100 ns. Due to this stable behavior we omit more detailed histograms for the FPGA platform.

Figure 5.6 shows the median latency for the t4p4s target. Latency is influenced by the packet rate as packets are sent either when a burst size of 32 is reached or after a timeout. This causes increased latency for packet rates below 0.5 Mpps as the batch is not filled quickly enough, instead waiting for the timeout. The latency is independent of the algorithm used, however, increases with packet size due to the increased serialization delay. Overall, the latency is between 10 μs and 80 μs, however, outliers, which regularly occur when using DPDK, exist as displayed in Figure 5.7a.

Figure 5.7b shows the NPU's stable latency behavior below 10 μs with no outliers for the baseline scenario. Performing the SipHash-2-4 operation shifts the latency distribution to the right up to 30 μs and increases the long tail.

	LUTs		Registers		BRAM	
	Absolute [-]	%	Absolute [-]	%	Absolute [kB]	%
Baseline	64 533	14.90	109 783	12.67	16 362	30.92
SipHash-2-4	66 380	15.32	114 282	13.19	17 460	32.99
SHA3-512	73 449	16.95	118 689	13.70	17 460	32.99

Table 5.5: Resource utilization for the NetFPGA SUME (taken from [7])

Resource Consumption

Packet processing, in general, is parallelizable, scaling well using multi-queue NICs and multi-core CPUs. Therefore, we disregard a longer discussion about resource consumption on CPU targets. Depending on the use case or library chosen for externs on the CPU target, the hardware of a CPU-based systems can be tailored to meet an application's resource requirements.

Apart from the described performance issues, we did not encounter resource restrictions for the Netronome card as the P4 program is of small size even when adding the SipHash implementation. For other applications, the program may be too large such that the generated firmware image can no longer be loaded onto the card. During runtime, the already mentioned memory architecture was a limiting factor.

Finally, Table 5.5 lists the resource consumption of the NetFPGA SUME implementations in regard to Look-Up Tables (LUTs), registers, and block random access memory (BRAM). Adding the open-source hashing IP cores increases resource consumption only moderately by less than approximately 2 % compared to the baseline program.

5.2.7 Limitations

The performance of the evaluated programmable target platforms depends on the properties of the chosen hash function and their respective, target-dependent implementation. For this work, we restricted our selection to open-source implementations, because we are primarily interested in the general feasibility of using cryptographic hash functions in programmable data planes. While we have shown that this is possible, it could be possible to reduce implementation artifacts, improving the performance and resource utilization. For instance, more sophisticated hash function implementations, e.g., based on commercial FPGA IP cores, in combination with an optimized integration into the P4 program, using, for instance, a higher degree of parallelization or pipelining, could be helpful. However, these solutions would require further monetary costs and engineering effort.

5.2.8 Conclusion

Our review of the current use of hash functions in P4 applications reveals two insights. First, a prevalent use of CRC, making applications vulnerable to potential attacks targeting hash collisions. Second, protocols and applications requiring cryptographic hashes for authentication or integrity cannot at all or only with severe difficulties be described and implemented using P4. We argue, that the implementation of cryptographic hash functions into existing software and hardware targets is possible and would increase the applicability of P4 to a wider range of use cases.

Specialized hash functions, optimized, e.g., for performance exist. While functions like SipHash do not provide the full set of cryptographic properties as classic cryptographic hash functions like, for instance, SHA-based functions, they are often designed for MACs. These are typically well suited for networking applications in the data plane as they work with low amout of input data, producing short outputs. Further, they are optimized for high performance, requiring few CPU cycles comparable to non-cryptographic hash functions, a property critical for network applications operating at line-rate.

We describe our prototype implementations using two different approaches of integrating cryptographic hashing algorithms in three different P4 target platforms—CPU, NPU, and FPGA. Our analysis shows that the CPU target is easily extensible, allowing integration of hashes as platform-specific P4 externs. The disadvantage of this platform is the high worst-case latency of up to several milliseconds, typical for software packet processing platforms. The tested NPU allows directly implementing externs in a variation of C. This target platform offers the highest throughput, but is unable to process packets larger than 900 B efficiently due to a platform-specific restriction imposed by its memory architecture. The FPGA-based target offers the lowest latency with small variance. However, the hashing IP core currently cannot be integrated using native P4 features. Instead, the functionality has to be implemented as separate module, changing the P4 architecture model. Depending on the number and positioning of P4 pipeline and hashing modules, the programmability of the hashing module is limited. However, in 2021 Malina et al. [217] have successfully shown that efficiently implementing not only hashing, but also algorithms for symmetric encryption and digital signatures into P4 data planes as externs is possible for a Virtex UltraScale+ FPGA platform.

We did not include an implementation and evaluation of an ASIC prototype target as the hardware and its architecture cannot be changed after manufacturing. However, as both, ASIC-based P4 target platforms and cryptographic hashing implementations, exist, we assume that merging them is possible and will be available commercially in the future.

Our evaluation shows that the performance of hash functions is highly target, algorithm, and use case specific. Therefore, we cannot recommend a one-size-fits-all solution. However, with different integration approaches available, we rather suggest that P4 targets should implement hash functions from a family of different algorithms. The PSA should recommend which functions and families exist and define how they operate on header and payload data. These recommendations should include cryptographic hashes and take into account the unique characteristics, i.e., memory footprint and resource consumption, of platforms such as CPU, NPU, FPGA, or even future ASICs.

5.3 Case Study: SYN Flood Defense in Programmable Data Planes

Section 5.3 is based on a collaboration between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, and Georg Carle [8]; and a joint work between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle [19], which itself is based on the Master's Thesis [20] by the author.

The SYN flood attack is a common attack strategy as part of DDoS attacks. This specific form of attack generates a flood of spoofed SYN segments pretending to initiate TCP connections. A flaw in the TCP handshake causes highly asymmetric costs for connection setup: server-side state allocation has to be done for each SYN segment received, eventually exhausting the server's resources. Consequently, the main burden is put on the attackee.

The frequency, volume, and diversity of DDoS attacks continue to increase [218]. Due to the simplicity of the attack, TCP SYN floods are a popular attack vector used in larger DDoS attacks [219], [220]. According to Kaspersky Lab's quarterly reports, from 2017 to 2020, the share of SYN flood traffic during large-scale DDoS attacks rose up to 92 %, becoming the "most popular type of attack" [221].

To defend against SYN flood attacks, avoiding downtime of valuable services, malicious traffic has to be separated from legitimate TCP requests. There are two potential mitigation approaches: generic defense, that tackles any form of DDoS attacks; and SYN-specific defense, fighting off TCP SYN flood in particular. Generic defense mechanisms remove, filter, or redirect malicious traffic through techniques such as blackholing of address ranges or per-flow tracking of traffic statistics [92]. The SYN-specific approaches use stateless client puzzles like SYN cookies or SYN authentication [222], [223]. These require the client to behave correctly beyond the initial SYN segment. The strength of the SYN-specific defense capabilities relies on the available performance to enforce and check correct TCP behavior before finishing a TCP handshake.

In this section, we investigate powerful off-the-shelf data planes to mitigate SYN flood attacks in the 10 Gbit/s range without overloading. We use the software-based DPDK and hardware platforms such as NPU and FPGAs that can be programmed using the P4 DSL, leveraging our cryptographic hash integrations introduced in Section 5.2. Specifically, we focus on SYN-specific solutions, protecting entire networks that provide high service guarantees with low latency to legitimate clients while under SYN flood attack. We discuss the benefits and challenges in regard to flexibility and usability of implementing these strategies using the P4 DSL compared to traditional software packet processing frameworks with kernel-bypass and raw packet handling. We provide insights regarding portability and target-specific code adaptions and use measurements to highlight connection success probability and latency.

5.3.1 SYN Flood Mitigation

SYN flood mitigation can be separated into generic and SYN-specific approaches.

Generic Defense

SYN flooding is most successful when utilizing spoofed source addresses. Applying ingress filtering [224] would reduce effectiveness of SYN flooding and other DDoS attacks relying on spoofed addresses. As RFC 2827 states, "by restricting transit traffic which originates from a downstream network to known, and intentionally advertised, prefix(es), the problem of source address spoofing can be virtually eliminated" [224]. However, due to the distributed characteristic of the Internet, this is unlikely to happen [225].

Active or passive traffic monitoring of the total traffic in the network can be used to detect anomalies, specifically to detect higher than usual volumes of SYN segments, at the network's edge [226], [227]. Machine learning or other means, e.g., CUSUM [228], [229], can be used to discern abnormal traffic patterns. As result, the attack can be thwarted or throttled before the traffic reaches the targeted server.

The simplest mitigation approach is blackholing all attack traffic, including SYN flood, during an ongoing attack, for instance, by filtering based on the source subnet. Unfortunately, this approach also rejects any legitimate connection attempts from these subnets. Another approach, using IP anycast to spread the load over multiple networks, increasing network resilience and the attack surface used to mitigate the attack, is an improvement [230]. However, an anycast network is challenging to implement [230] and during large attacks, collateral damage, impairing other services in the network, is possible [231]. Lastly, tracking per-flow statistics and using thresholds can be used to distinguish legitimate from suspected attack traffic [92].

The mentioned generic approaches are simple and effective, but highly unspecific. While these techniques achieve the goal of mitigating SYN floods and other attacks, they likely cause undesired effects, i.e., discard legitimate traffic. Consequently, this reduces the service quality for legitimate clients.

SYN-specific Defense

SYN-specific defenses have a narrow, highly specific focus to improve on the generic approaches: they improve the behavior of the network stack and its data structures or employ sophisticated challenges for TCP clients to specifically target malicious SYN flood traffic.

Parameter Optimizations: As the attack targets the TCP/IP stack of the server, parameter optimizations of the stack can be utilized on the target server itself. Tweaking parameters of the architecture or the application results in only minimal, if at all, any success in throttling the attack as these adjustments only affect the overall performance of the network stack and application [232], [233].

RFC 4987 describes several SYN flood-specific parameters that can be adjusted. This includes increasing the *backlog*, the buffer for unfinished TCP connections, or reducing the SYN/ACK retransmission times, thereby reducing the timeout per Transmission Control Block (TCB). However, considering bandwidths of 1 Gbit/s and more, these parameters only marginally delay the success of the attack by using more memory resources. In fact, the *backlog* structure might not be designed to surpass a certain size, as data structures or search algorithms used might become inefficient. This may worsen performance during normal operation without an ongoing attack. Reducing timeouts might lead to legitimate connections not being fast enough to finish the handshake before the entry gets deleted. [190]

Replacing TCB entries either randomly or by choosing the oldest entry is another approach to modify the *backlog*. While this works for SYN flood rates of up to approximately 500 SYN segments per second [234], for packet rates possible in today's networks, legitimate connections are evicted with a probability of more than 99 % before they get established.

SYN Cache: The SYN cache approach only stores essential data, which can be as few as 16 B [235] in the TCB. Secret bits of the SYN segment, in combination with the addresses and ports, are hashed, to determine the bucket in a hash map [190], [225]–[227], [235]. In FreeBSD the size of such an entry is reduced to 160 B, compared to 736 B for a TCB [226] in Linux, and can contain more than 15 000 entries. Furthermore, the linked-list of each bucket has a limited size, i.e., if a bucket would overflow, the oldest entry gets deleted [236]. The idea is that the attacker does not know how the hash is calculated, hence, cannot attack the hash structure. Appropriate hash function choice ensures that memory and computational resources required are limited [190]. A downside is the restricted or not existing support for TCP options, which worsens the performance of the connection as the MSS option is required to cope with speeds encountered in modern networks. The attack will eventually be successful when surpassing a high enough SYN flood rate to exhaust the data structure.

SYN Cookie: For SYN cookies [222], state, usually kept by the server, is encoded and put into the initial server-side sequence number y, which can be freely chosen. A legitimate client will finish the handshake sending an ACK segment, setting the acknowledgment number to $y + 1$. The server only accepts the connection and creates the necessary state, if the received sequence number can be decoded and verified. Using standard SYN cookies, the 32 bit initial sequence number is made up of three different values: a timestamp value to prevent the collection and re-injection of old cookies [222]; the Maximum Segment Size option as it is essential for TCP performance; and a cryptographic hash of the connection 4-tuple, consisting of source and destination IP addresses and ports, and the timestamp value. This way, it is infeasible for an attacker to create a valid SYN cookie by itself.

SYN cookies perform a trade-off: instead of consuming memory resources, CPU resources are exhausted [237]. This allows for connection depletion attacks, exploiting the hash verification mechanism by performing unresolved handshakes with the server [237]. Even an attack with low bandwidth can shut down a high-performance server, exploiting the asymmetric nature of the attack [238]. The server not only has to calculate the cookie, but also verify it in the received ACK segment. In theory, this makes SYN cookies susceptible to ACK floods, however, the computational effort is the same as calculating the cookie during a SYN flood. Similar to the SYN cache, a drawback of this approach is the restricted support of TCP options. An updated SYN cookie layout uses the TCP timestamp field to also encode other options that are widely used in modern networks [226], including window scaling and selective acknowledgments [239].

SYN Authentication: SYN authentication [234], [240] aims to further reduce the resources required to verify the client's legitimacy. The server expects a certain response to a triggered, unusual event, which can be compared to the challenge in client puzzles [223], [237], [241]. If the client reacts accordingly, any further connection attempt is whitelisted.

The simplest idea is to ignore or reset the initial connection attempt, however, an attacker can simply resend the SYN segment and is whitelisted. More advanced, the server sends, e.g., a SYN/ACK segment with an invalid sequence number. Clients are expected to respond with a RST segment [235], referred to as $\text{Auth}_{\text{Invalid}}$. An attacker can circumvent this by including a flood of RST segments within their attack. Other SYN authentication approaches only whitelist once the client has demonstrated its intent to finish a complete handshake (referred to as $\text{Auth}_{\text{full}}$) or even by means of a higher layer protocol, e.g., an HTTP GET request is received. Fingerprinting techniques can be used to increase the probability that SYN and respective reply segments for a handshake were sent by the same client. For instance, the IPv4 TTL [240] or TCP options signature can be compared, referred to as Auth_{TTL}. All SYN authentication approaches can be further enhanced by including a cookie value as sequence number, similar to SYN cookies [234], referred to as $\text{Auth}_{\text{cookie}}$.

During the initial connection attempt, no TCB state is created. Instead, only when the client reacts accordingly, it gets whitelisted, which can be accomplished by storing one bit per flow, source address, or even source subnet. In particular, no information about TCP options, or only a signature when fingerprinting, is stored. Successive connection attempts do not have to be modified or inspected beyond the SYN flag, until the whitelist entry is invalidated.

A drawback is the interruption of the regular protocol flow through a connection reset, increasing the delay for the initial connection. Furthermore, if the attacker manages to get whitelisted, a subsequent SYN flood is considered legitimate traffic.

Comparison: We compare the introduced approaches based on metrics also discussed in literature [234] in Table 5.6. The weakness of the SYN cache is the memory exhaustion, while SYN cookies shift possible attack vectors towards CPU exhaustion attacks. SYN authentication neither requires extensive memory nor CPU resources. The exception are Auth_{TTL} to store TTL values and $\text{Auth}_{\text{cookie}}$ to calculate cryptographic hashes. Calculating cookie values is the limiting factor for efficiency, i.e., how much SYN flood can be processed. None of the SYN authentication strategies are transparent for the client application, as they reset the initial connection. It has to be noted, that, for instance, for modern browser implementations, an automatic retry is performed in the case of a reset, resulting in no significant service downgrade for the user.

A downside of SYN cookies is the limited support for TCP options. Robustness specifies whether the signaling capabilities of TCP are compromised [234]. This is the case for all SYN-specific strategies besides the SYN cache, as no initial state is kept, wherefore a retransmission of a SYN/ACK segment is not possible. This is more severe for SYN cookies than for SYN authentication, as the latter works on the assumption that the client retries failed connection attempts anyway.

			SYN authentication			
	SYN cache	SYN cookie	Invalid	Full	TTL	Cookie
Memory immunity	−−	+	+	+	−	+
CPU immunity	+	−	+	+	+	−
Efficiency	−	+	++	++	++	+
Transparency	+	+	−	−	−	−
Option support	+	−/o	+	+	+	+
Robustness	+	−	o	o	o	o
Classified as legitimate traffic						
False Positive	+	+	−	−	o	+
False Negative	+	+	+	+	+	+

Table 5.6: Comparison of SYN flood mitigation strategies

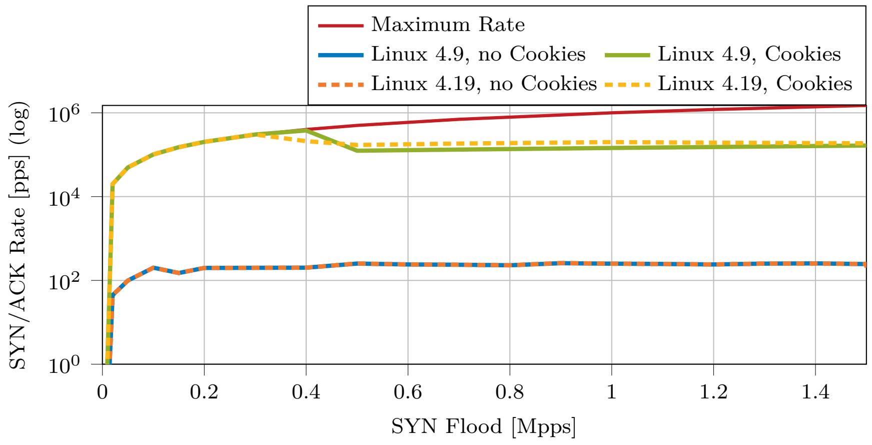
Lastly, we evaluate the techniques by whether they correctly classify legitimate traffic. Assuming all packets are received correctly and within the time frame of the calculated cookie, no technique produces false negatives, i.e., legitimate traffic classified as malicious traffic. Only the SYN cache, SYN cookies and $\text{Auth}_{\text{cookie}}$ cannot be circumvented by malicious traffic, i.e., only a legitimate connection is passed to the server. No other SYN authentication strategy fulfills this requirement, as a RST or ACK flood can achieve whitelisting for malicious SYN segments as no cryptographic cookie is used.

SYN cookies and SYN authentication violate the end-to-end principle of TCP. However, only during attacks, where a TCP SYN flood would likely cause a service downtime, we deem the application of these approaches justified. The assumption is that the server is under severe load caused by a SYN flood, while only a small percentage of the traffic is legitimate TCP traffic. Instead of no service, the goal is to provide a best-effort approach of service quality for legitimate flows, i.e., higher delay or minor connection disruptions are acceptable. None of the listed methods reliably shuts down the attack, instead the success of the attack is delayed by having to use more resources, either memory or computational, for the attack. Furthermore, no strategy is part of the official TCP specification or standardized by a committee [226]. The different strategies offer a trade-off between efficiency (performance) and correct classification (false positives): only the methods using a cryptographic hash reliably protect against the attack, however, at the cost of reduced performance.

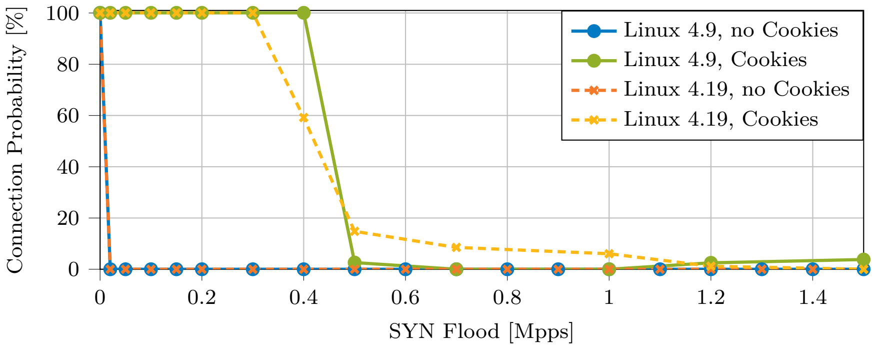
Case Study: SYN Cookies in Linux

Mere parameter changes to the TCP stack cannot effectively defend against SYN floods. Therefore, the Linux TCP/IP stack has SYN cookies enabled by default. However, they are only used when the *backlog* of a socket is already full [242]. Linux follows the proposed SYN cookie layout, encoding the MSS and a timestamp value with 60 s resolution. If the client sends a timestamp option it is used to encode further TCP options.

As a case study, we analyze two different Linux versions when subjected to a SYN flood, while serving a static website: 4.9.0 and 4.19.0, using the SHA1 and SipHash hash functions for



(a) Processed SYN flood



(b) Served HTTP requests. 100 requests per second offered

Figure 5.8: Performance of Linux during SYN flood

cookie calculation, respectively. Figure 5.8 shows the amount of processed SYN flood and the probability of successfully serving 100 HTTP requests per second for an increasing SYN flood. When disabling SYN cookies, Linux can only process up to 250 SYN packets per second. Worse, the connection success probability is close to zero, as no HTTP requests are served successfully. Profiling of the network stack's functions shown in Figure 5.9 reveals that this behavior is not due to CPU exhaustion. The CPU is only fully utilized after 0.4 Mpps, as indicated by the decline of idle CPU cycles. This leads to the conclusion that the TCP backlog is the limiting factor.

When enabling SYN cookies, both Linux versions behave similarly, i.e., process up to 0.4 Mpps of SYN flood, while all HTTP requests are served. When further increasing the SYN flood, the probability of serving any legitimate requests successfully approaches zero. For Linux 4.19, it is notable that a percentage of less than 10 % of requests is served successfully, even for rates of up to 1 Mpps of SYN flood traffic. The profiling reveals that, in this setting with SYN cookies enabled, the CPU utilization is the limiting factor. A clear difference between the Linux versions is the number of CPU cycles used for the hash calculation. Cookie calculation using SHA1 requires up to 15 % of the cycle budget, while SipHash with only 2.5 % is more

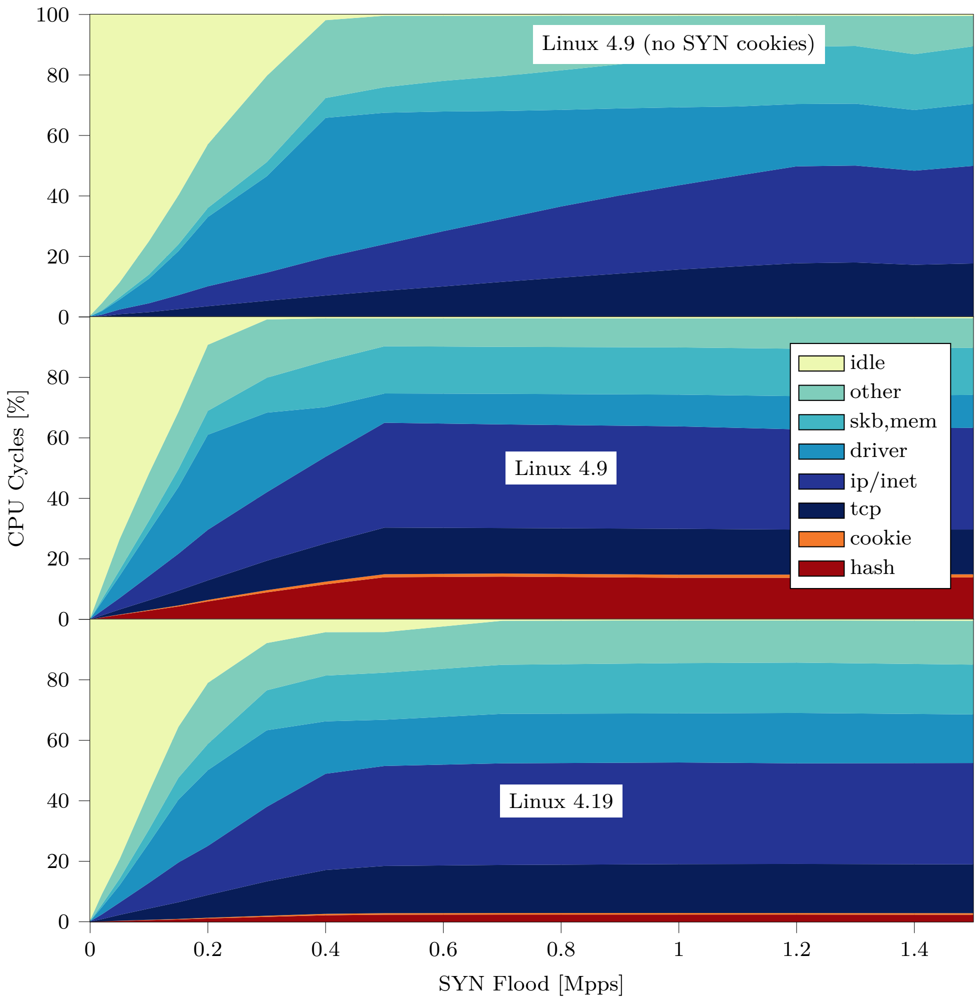


Figure 5.9: Profiling of different Linux versions during a SYN flood

efficient. This performance difference also supports our findings presented in Section 5.2. Given the cheaper hash function, one would expect an increased performance for Linux 4.19 instead of the measured decrease. We attribute this development to other networking related changes between these Linux versions, which are out of scope of this study.

Profiling also reveals common performance limiting factors of the Linux network stack. The majority of CPU cycles are spent handling packet buffer and memory referred to as skb and mem in Figure 5.9, driver related processing (driver), IP layer processing (ip/inet), and the TCP stack itself (tcp). The numbers are in the expected range and comparable to other Linux studies [3].

Primary goal of the network stack is to support as many protocols as possible, providing a reliable, stable, and robust interface for user space applications. Thus, raw performance is secondary, wherefore Linux should not be used for mitigation of large-scale SYN floods. To

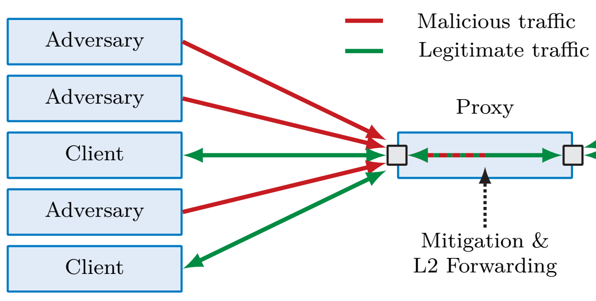


Figure 5.10: SYN proxy deployment scenario

perform the attack mitigation directly on the end host, optimized solutions are required. One approach to increase performance is to perform filtering operations like DoS mitigation before the network stack, e.g., by using the eBPF- and P4-programmable XDP.

5.3.2 Deployment Scenario

The aforementioned generic and SYN-specific strategies, targeting transport or application layer, can be implemented directly on the end host or within the network. A special case of the latter is deploying SYN-specific mitigation approaches as separate node, called SYN proxy.

On the End Host

All of the parameter optimizations, targeting the application or network stack, are deployed exclusively on the end host. All other strategies may be deployed on the device being the target of the attack. However, in this case, mitigation does not scale as it only protects this particular node. For instance, in a cluster of nodes, each node has to perform its own SYN flood mitigation. Further, the mitigation wastes CPU cycles that could otherwise be used for running actual applications. Our case study has shown that when leaving this task up to the network stack, in our case the Linux TCP/IP stack, in the best-case scenario, the node can cope with a maximum attack rate of 0.4 Mpps.

Network-wide as SYN Proxy

Generic approaches, including traffic monitoring, statistics gathering, and blackholing of volumetric attacks, can be deployed anywhere in the network, typically close to the network edge to detect malicious traffic early on.

SYN-specific attack mitigation strategies are commonly deployed as SYN proxy. This proxy can be used to protect multiple servers as shown in Figure 5.10, or, as part of a traffic scrubbing center or in the cloud, even multiple sites. The concept of a SYN proxy is based on the idea of intercepting potentially harmful traffic. Instead of the server, the proxy will answer to the initial SYN segment [190], [225], [226] using, for instance, SYN cookies or SYN authentication and only forward authenticated packets to the server. SYN proxies can be combined with monitoring services, i.e., only when an attack is detected, traffic gets routed through the proxy.

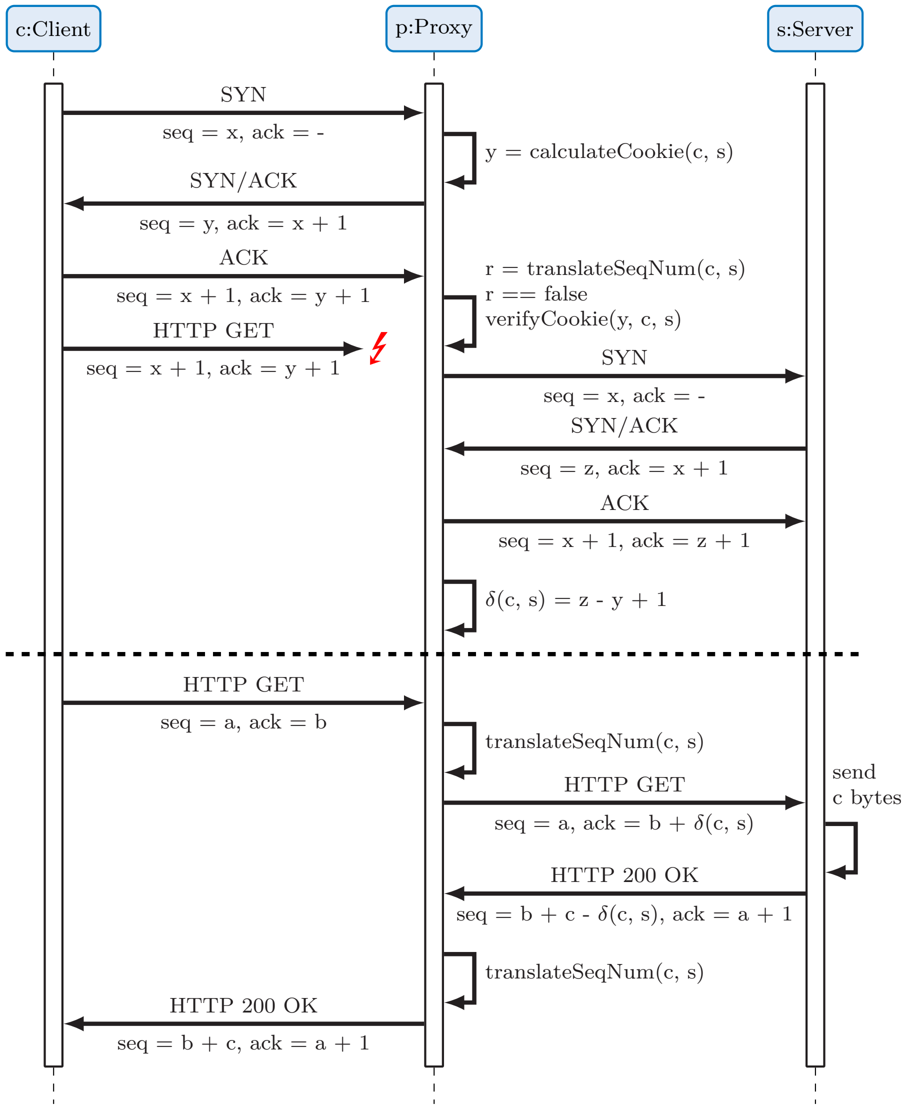


Figure 5.11: Message Exchange for SYN cookie strategy

In the following, we explain the actions performed by a SYN proxy. We focus on using SYN cookies and SYN authentication as they show the highest potential as discussed in Section 5.3.1.

SYN Cookies: SYN cookies used in a proxy setup as displayed in Figure 5.11 raise several issues. Only after the verification of the cookie, the full state object for an established TCP connection is created by the proxy. As the server is unaware of the connection and to uphold the transparency provided by SYN cookies, the proxy has to perform a second handshake with the server. While the proxy can reuse the first handshake's client-side sequence number in its SYN segment, the server chooses an initial sequence number z at random. As this won't match the first handshake's server-side sequence number, the proxy has to store the difference δ between

	SYN Cookies	SYN Authentication
Packet modification	every segment	handshake
Transparent	yes	no
Option support	limited (encoded)	full
State		
State per	flow	flow or subnet
Lookup key	4-tuple	4-tuple or source IP
Memory required	> 32 bit/flow	2 bit/flow
Lookup for	not SYN	every segment

Table 5.7: Comparison of SYN cookie and SYN authentication in proxy setup

the client- and server-side number. For all subsequent segments of the connection, the proxy has to modify sequence and acknowledgment numbers accordingly.

Another issue is that the first data segment of the client is sent directly after finishing the first handshake. This poses a problem for the proxy, as its handshake with the server is likely not completed yet. If the proxy drops this data segment, the client retransmits it after a timeout, for which 200 ms is a common time period. Considering that typical RTTs are only a hundredth of this, this retransmission causes a significant delay penalty for the connection attempt. A solution for the proxy is to temporarily store the client's initial data segment instead of dropping it. Once the second handshake is completed, the stored segment can be translated and forwarded. Alternatively, the proxy can actively notify the client once the second handshake is completed, by resending the SYN/ACK segment, triggering a retransmission of the data segment. Further improvements include setting a window size of zero, called zero window, in the original SYN/ACK segment, indicating that the server cannot process any more data. Until the SYN/ACK is resent with a non-zero window size, the client will not send the initial data segment, reducing bandwidth wasted for a segment that will be dropped.

SYN Authentication: SYN authentication is efficient due to its simplicity when used in the proxy setup. Figure 5.12 shows $\text{Auth}_{\text{cookie}}$, i.e., SYN authentication using a full TCP handshake with cryptographic cookie. The first connection attempt is interrupted after verifying the legitimacy of the client, i.e., the client is willing to create its own TCP connection state. By whitelisting all further attempts for this client, the proxy does not have to keep a separate connection with the server or perform sequence number translation. No connection state has to be stored by the proxy, wherefore, a simple bitmap representing the whitelist is sufficient. However, as ACK segments of established connections cannot be distinguished from the third segment of the handshake, the proxy has to check every segment against the whitelist. If the origin is not whitelisted, the segment is assumed to be the third segment of the handshake and, when using $\text{Auth}_{\text{cookie}}$, the cookie hash is verified.

$\text{Auth}_{\text{full}}$ is the same procedure as $\text{Auth}_{\text{cookie}}$, however, skipping the hash calculation and verification. Consequently, $\text{Auth}_{\text{full}}$ is easier to circumvent.

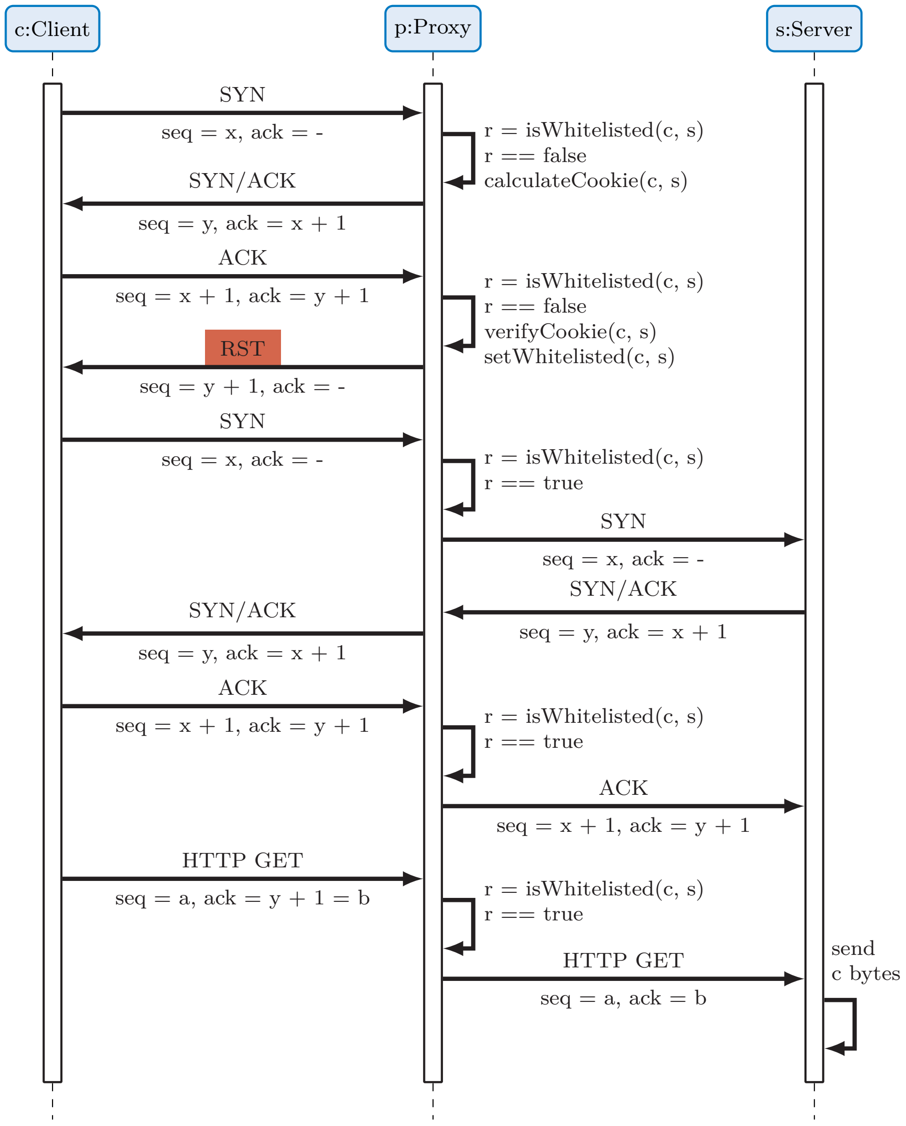


Figure 5.12: Message Exchange for SYN authentication strategy

Comparison: SYN cookie and SYN authentication, when used in the proxy setup, retain their properties regarding transparency and option support as highlighted in Table 5.7. SYN cookie has to modify every segment, while SYN authentication only modifies segments during the handshake. For the latter, the size of the kept state depends on the whitelisting granularity, e.g., per flow, per source address, or per subnet. However, state is reduced to a few bit per entry compared to keeping the sequence number difference per flow for SYN cookies. Both strategies need to perform a lookup for every segment, aside from SYN segments for SYN cookie, to determine the action. However, with P4 devices using match-action pipelines, this can be done efficiently.

5.3.3 Related Work

Related work on SYN flood attacks and mitigation is broad. It covers detecting and modeling of attacks, implementing SYN flood mitigation procedures, and doing so using high-performance frameworks.

Modeling SYN Flood Attacks: Maregeli analyzes of the impact of SYN flood attacks on servers and the network and contributes a model of the probability of successfully establishing a TCP connection during a SYN flood attack [225]. The measurements show that, while shutting down a server is already successful with less than 100 SYN segments per second, even for larger backlogs, a large-scale volumetric SYN flood similar to a UDP flood can successfully deprive the resources of networking devices, affecting the complete network. Compared to other approaches that use complex mathematical operations like Markov models or solving of differential equations to model buffer occupancy and residual waiting times [243], [244], Maregeli proposes a simplified and reduced model based on queuing theory, focusing on the service quality for legitimate clients. A comparison with other approaches shows that the model is not accurate for low traffic rates. However, for higher rates the results coincide, especially considering the drastically reduced mathematical effort of the model [225].

SYN Cookie Implementations: Besides Linux, other major operating systems like Windows and FreeBSD utilize TCP SYN cookies as preferred mitigation method, enabled by default during periods with high traffic volumes [245]. However, as our performance evaluation of the Linux network stack in Section 5.3.1 shows, these network stacks typically do not focus on performance. Their primary goal is usability and flexibility, supporting as many protocols and scenarios as possible.

SYN cookies, as part of a SYN proxy, have been implemented by multiple projects. The *SYNPROXY* module for the netfilter framework is available in Linux [246]. Managing the SYN flood, *SYNPROXY* forwards only legitimate traffic to the Linux kernel. To do so, the initial SYN segment is intercepted by netfilter, calculating a SYN cookie. Once the client finishes the connection establishment with a verified cookie, the proxy sends a SYN segment to the original server destination, using the initially negotiated options. After finishing the second handshake, the proxy is only involved in sequence number and timestamp translation [246]. This approach enables mitigation of a 2 Mpps SYN flood using only 7 % CPU utilization on an eight-core test system [246]. The disadvantage is that it is integrated with a Linux end host, i.e., it cannot be deployed as proxy or ported to different target platforms. Our P4 implementations improve on this as they are portable and can be deployed as SYN proxy.

Zhang et al. propose Poseidon [92], a DDoS defense framework that maps customizable mitigation strategies to programmable data planes in the network. The use-case is as part of a scrubbing center, cleaning the traffic from not only SYN flood traffic, but general DoS traffic of customer networks. Poseidon uses generic defenses like packet counting and probabilities to mitigate most attack vectors, including SYN floods. For HTTP floods, Poseidon does not create client puzzles in P4, but creates them in a separate DPDK proxy. This comes at the cost of higher end-to-end latency of approximately 70 ms, similar to existing commercial solutions. We improve on Poseidon's SYN flood mitigation by performing puzzles in the data plane without the need for probabilities or other generic approaches. In fact, our solution could be integrated as part of Poseidon's network orchestration framework.

Large-scale commercial solutions exist, however, due to the closed-source nature, implementation details are rare. For instance, the Arbor Threat Mitigation System [247] provides middleboxes of different scale for traffic scrubbing, including unspecified SYN flood mitigation. Cloudflare offloads the TCP handshake to the cloud using an IP anycast network [230], [248]. Only once the handshake completes, it is forwarded to the target server.

Other proposed techniques and implementations produce unsatisfying performance with less than 0.1 Mpps mitigated SYN flood [249], break TCP, require specialized clients [250], or only mitigate attacks in particular scenarios, thus interfering with deployment [251].

Exploiting TCP Simultaneous Open for SYN Flood Mitigation: Two clients can simultaneously open a TCP connection with each other using TCP simultaneous open [252]. Consequently, a total of four messages is used during the handshake. Zuquete et al. propose to exploit this for SYN flood mitigation with SYN cookies [253]: they send the SYN cookie without ACK flag, simulating a simultaneous open. A legitimate client will respond with a SYN/ACK segment. The advantages are two-fold: the server can include its TCP options in the initial SYN; and the client will send its TCP options again in the SYN/ACK segment. Consequently, neither side's options are lost compared to the normal SYN cookie approach.

However, there are severe downsides. First, implementing TCP simultaneous open is uncommon and requires that the end-points use a well-known source port [254]. Further, this exploit depends on firewalls allowing inbound SYN segments to the client. This is further complicated if the client is located behind a device performing network address translation. Zuquete et al. tested TCP simultaneous open support of operating systems. Their results show that none of the evaluated Windows and Cisco IOS versions supported the feature. Further, for other OSes like Linux and MacOS only selected versions supported it or the behavior was not correctly implemented [253]. Our tests using Linux 3.16 and 4.6 show that for both versions, the client and server properly implemented TCP simultaneous open.

Zuquete et al. propose two improvements to tackle these problems. First, using regular SYN cookies as fallback in case no response is received to trying TCP simultaneous open [253]. However, this approach can be considered even worse as it creates new attack vectors. In particular, the server itself can be used to perform an amplification attack, generating both a potentially harmful SYN and a SYN/ACK segment as response to a single connection attempt. Zuquete et al. counteract this by storing clients without suspected support for TCP simultaneous open in a cache [253].

High-Performance Attack Mitigation in Software: As standard network stack implementations are not capable of high-performance packet processing, dedicated frameworks have been created over recent years [25], [26], [28], [37], [255]. With kernel bypass, memory pre-allocation, parallelization and batch processing, among others, deficiencies of common bottlenecks are negated, enabling packet processing beyond 10 Gbit/s on commodity hardware.

DDoStop [256] is an application based on the Snabb virtual switch [147], a framework for fast packet networking. This NFV-like app offers zombie detection, a term coined by Arbor Networks, i.e., blocking of "source hosts that exceed certain thresholds" [256]. Subject to the complexity of the rule-set and required packet parsing, 1 Mpps to 10 Mpps of general DoS traffic can be mitigated per CPU core [256].

5.3.4 Requirements for SYN Proxy Implementation

We have identified the following requirements for a flexible, open-source and high-performance SYN proxy:

SYN Flood Mitigation: No traffic part of the SYN flood should reach the protected servers. The proxy should filter and drop malicious traffic while correctly detecting and forwarding legitimate connection attempts.

Performance: The proxy should operate at line-rate of 10 GbE, i.e., it should be able to process 14.88 Mpps of SYN flood. Furthermore, it should be scalable to allow for even higher data rates, e.g., 40 or 100 GbE.

Service Quality: The services provided by the protected servers should be reachable for as many legitimate clients as possible, reducing service downtime. As connection attempts occur during a DoS attack period, a best effort approach regarding service quality for legitimate traffic is chosen. Furthermore, the behavior of the proxy should be transparent and adhere to standards whenever possible to enable interaction with different client and server implementations.

Portability: The implementation should be platform-independent, facilitating deployments on a wide range of different software and hardware targets. Additionally, fine-grained configurability should be possible, avoiding setup dependencies whenever possible.

Extensibility: The proxy should be modularized to serve as starting point for further SYN flood mitigation strategies and mitigation of other types of DoS attacks, e.g., UDP or DNS floods.

While the proxy should perform well without any malicious traffic, it is typically only deployed during an attack. Therefore, the proxy is designed with the assumption that more than 90 % of all received traffic is malicious, i.e., only a minority of traffic is considered legitimate.

For the remainder of this section, we focus on three techniques: regular SYN cookies, as well as SYN authentication performing a full handshake with and without including a cryptographic hash as cookie ($\text{Auth}_{\text{cookie}}$ and $\text{Auth}_{\text{full}}$). Only the strategies using a cryptographic hash offer adequate protection while also achieving high service quality. We also discuss not using the cookie calculation to highlight the impact of performing the cryptographic hash.

We implement these mitigation methods in SYN proxies using two different approaches: based on kernel-bypass and raw packet handling with the libmoon packet processing framework; and using the P4 language to create platform-independent programs for software and hardware

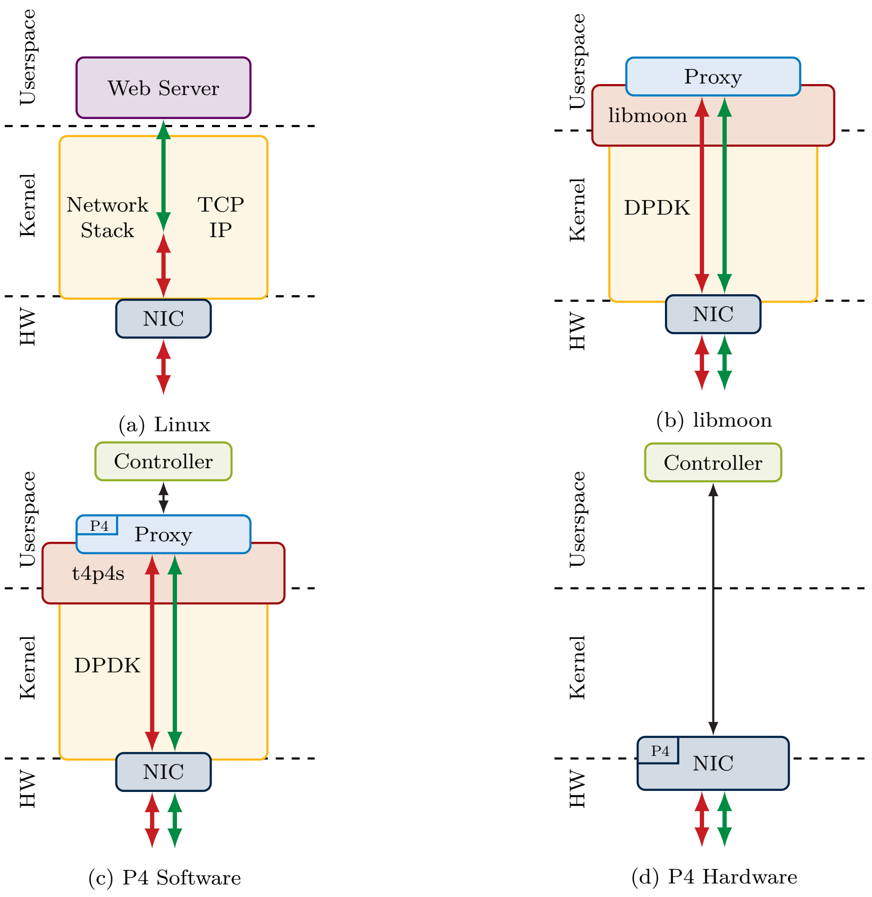


Figure 5.13: Architecture of SYN flood mitigation in different data planes

targets. The different architectures for these prototypes are shown in Figure 5.13. Further, we compare our implementations with SYN cookies in Linux' TCP/IP stack.

5.3.5 Software Packet Processing Prototype using libmoon

Section 5.3.5 is based on a collaboration between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle [19]. This publication itself is based on the Master's Thesis [20] by the author.

We use libmoon [4], [149] to implement our SYN proxy prototype in software targeting COTS hardware. The proxy is running as user space program, bypassing the regular network stack of the OS as shown in Figure 5.13b. libmoon does not offer a TCP stack, therefore, the handling of the TCP handshake has to be done by the proxy application itself. Utility functions for packet templating are available due to libmoon's dynamic protocol stack presented in Section 3.3, simplifying the implementation of the TCP handshake. libmoon receives and processes packets as batches. For each packet of a batch, depending on the mitigation strategy used and TCP flags set, a response is crafted and sent after all packets of the batch have been processed. We use the pseudo-cryptographic SipHash hash function for the calculation of cookie values for the reasons discussed in Section 5.2.

Connection State Tracking

A crucial part of state maintenance is correct tracking of TCP connection state, i.e., adding new connections and to regularly remove dead connections. Removing state of finished connections serves two purposes: freeing memory space and allowing the same 4-tuple to initiate a new connection. Approaches to state removal are either active or passive. State tracking aims at actively reacting to FIN and RST segments of the connection to determine when the connection concludes. For each flow, state kept by the proxy has to reflect whether such a segment has been seen. If state tracking concludes that the connection has been terminated, i.e., both sides sent and acknowledged a FIN segment or a RST has been sent, the corresponding state entry can be removed immediately. This approach is difficult to implement as reordering of segments, packet loss, retransmissions, or connection timeouts may result in state being deleted too early or not at all.

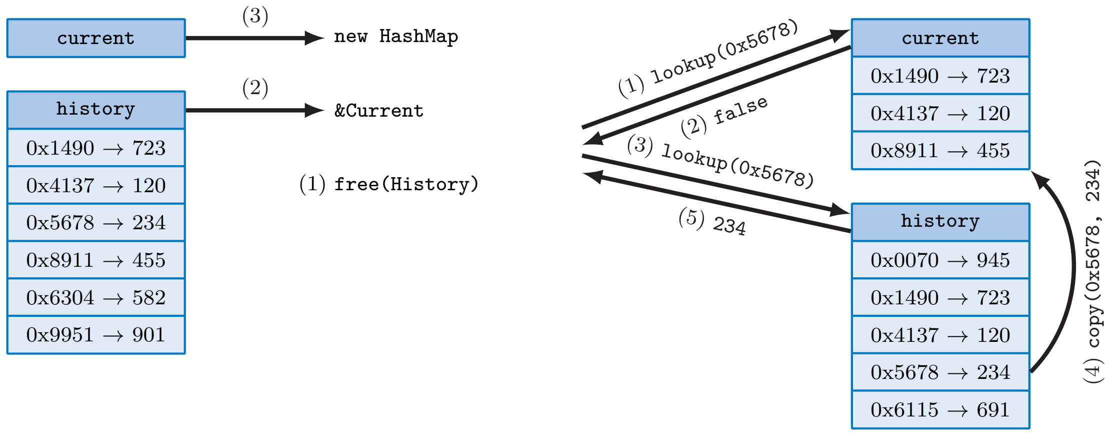
A passive approach is to use garbage collection, e.g., applying a second-chance page replacement algorithm [257]. For this, each state entry is extended with two bits, which are set every time the entry is looked up. A periodic background process iterates over all entries and unsets one of the bits. If both bits are unset, the entry is deemed inactive and removed. Common timeouts for inactive connections, for instance, in Linux are at least 10 min [242], wherefore we chose a similar timeout for garbage collection.

SYN Authentication

SYN authentication requires no state information besides the previously described connection state tracking. We chose a fixed-size bitmap as data structure representing the whitelist. For each entry, two bits are used to implement the second-chance page replacement algorithm. This results in a total of approximately 1 GB of memory being used when representing the complete IPv4 address space and whitelisting based on individual source addresses. As the location of the bits for a particular entry is known, no hash function for the lookup has to be used. Due to the size of the IPv6 address space, this approach is not possible for IPv6 addresses. Alternative ideas use either whitelisting based on subnets or a hash-based data structure.

SYN Cookies

The SYN cookie approach requires additional state information besides the connection state tracking. Increased memory consumption makes the bitmap impractical. Instead, this approach uses concurrent hash maps, as the garbage collection process has to run as separate process to not interfere and block time-sensitive lookups. When using a normal hash map implementation, every insert operation would invalidate the iterator object used to traverse the map. A concurrent hash map implementation has the drawback that it is more complex, resulting in reduced performance. Furthermore, when using only this mechanism to remove invalid entries, the reuse of ports by the client for future connections is infeasible due to the long timeout.



(a) Swap-out (b) Lookup

Figure 5.14: Swap-out hash map

To leverage the advantages of both approaches—immediate removal of connections and garbage collection of left-over entries—we use a hybrid approach, coined swap-out hash maps. Instead of a concurrent hash map and timeout bits, two normal hash maps, called `active` and `history`, are used in conjunction. Rather than periodically unsetting bits, `history` is deleted and replaced by the `active` map and a new empty `active` hash map is initialized as shown in Figure 5.14a.

The lookup procedure consists of the two stages visualized in Figure 5.14b. First, the `active` map is checked. If the entry exists, it is returned, otherwise, the same lookup is performed in the `history` map. If no entry is found, the connection is new. Otherwise, the entry is returned and copied to the `active` map, employing the second chance mechanism. Inactive entries are eventually swapped out to the `history` map before being deleted with the next swap. Insert operations are performed exclusively on the `active` hash map. With this scheme, no hash map is traversed, removing the need of a concurrent hash map.

Because of the primary assumption of the proxy, i.e., only a margin of all traffic is legitimate and does not belong to the SYN flood, merely a small number of entries is expected to be maintained in the hash map. Therefore, we chose Google's sparse hash map implementation [258]. As each value object consists of approximately 40 bit, no bottlenecks regarding required memory space, when keeping thousands of inactive entries up to ten minutes, are expected. The performance impact of a swap-out and consequent copy-up operations for active connections is reduced by the long timeout interval.

The second issue of SYN cookies in the proxy setup is receiving data from the client before the second connection between proxy and server is completed. We implemented two strategies to delay this initial TCP data: storing the initial request and using active notification after completing the handshake with the server. Storing the segment at the proxy requires the complete data to be copied into a buffer, which is stored with the remaining metadata for the connection. This increases the amount of data stored per connection by up to 1500 B. In particular, it requires proper garbage collection of left-over stored packet buffers to prevent memory leaks. Furthermore, the added complexity makes this approach become infeasible if the client sends more than one data segment before the second handshake is finished or segments are received out of order. For both strategies, our evaluation has shown that latency can be reduced to 10 ms or lower, implying that no retransmission after a timeout is required. This is a significant improvement compared to waiting for a timeout of typically 200 ms.

Optimizations

To reduce computational complexity, offloading features of modern NICs are used for checksum calculation.

For every TCP segment received by the proxy, exactly one action has to be performed resulting in one outgoing packet, e.g., sending a SYN/ACK as response to a SYN. Parallelizing the processing by performing all possible actions for each packet of a batch is infeasible as it generates a lot of potentially harmful traffic, even increasing the flood attack. Instead, the proxy either answers with a specific templated segment, where only addresses, etc., have to be updated, or forwards the received segment with slight changes. Reusing RX buffers is only beneficial for the latter case. However, this concerns only a small amount of the overall received traffic, assuming that the majority is SYN flood traffic. Denoting which buffer can be reused and which buffer should be replaced with a templated packet using “dropout batching” [259] would require a bit-mask. As this cannot be efficiently done using Lua, calling respective functionality in C using the FFI would generate additional overhead per batch [259]. Furthermore, RX buffers cannot be reused when being transmitted on a different port. Instead, RX buffers are never re-used and we utilize on-demand allocation of reduced batch sizes. For buffers to be forwarded, the complete content is copied, while for all other replies we use templated TX buffers. For packets that are less likely to be used, for instance, RST segments after successful whitelisting, buffers are only allocated on-demand. This reduces overhead when the buffers are not actually used.

5.3.6 Programmable Data Plane Prototype in P4

P4’s match-action-based paradigm makes it an excellent language to realize packet filtering and DoS mitigation applications. Opposed to raw packet handling demonstrated in the previous section using DPDK, packets can only be modified through constructs allowed by the language. Furthermore, P4 has a clear separation between data and control plane as shown in Figures 5.13c and 5.13d: while the data plane can match the entries of P4 tables to incoming packets and perform respective actions, only the control plane can insert new entries into a P4 table. The data plane communicates with the control plane, which in our case runs on the same node that also runs the P4 data plane, using digest messages.

Realized Programs and Targets

We implemented P4 programs for SYN cookie and SYN authentication strategies and tested their functionality using the Mininet-based bmv2 P4 switch target. For all platforms, the core P4 program, available as open source [21], remains the same. However, due to using different P4 architecture models and offering different extern interfaces, all platforms require small, target-dependent modifications to the P4 program. To reduce overhead, we only ported the implementations requiring simpler state maintenance, $\text{Auth}_{\text{full}}$ and $\text{Auth}_{\text{cookie}}$, to multiple P4 targets: the DPDK-based t4p4s [69] (see Figure 5.13c); the NFP-4000 Agilio SmartNIC [45] NPU (see Figure 5.13d); and the NetFPGA SUME [72]. We chose t4p4s as it uses the same underlying framework as our libmoon implementation and offers a comparison between implementing raw packet processing to using a DSL for data plane programming.

Program Core

The P4 implementations for the mentioned strategies follow the same structure independent of the used strategy. At first, packets are parsed up to and including the TCP header. The following match-action pipeline at its core works as an L2 forwarder. Depending on the determined outgoing port, Ethernet addresses are updated using a table lookup. As P4 cannot generate new packets, the received packet is modified according to the strategy used, TCP flags set in the received packet, and the state kept by the proxy. State—whitelist or sequence number difference—is maintained as match-action table, requiring one lookup for every segment. No changes to the IP layer are performed besides exchanging IP addresses, requiring only an update of the TCP checksum before the packet is transmitted.

Target-specific Changes

Aside from the core logic that is portable between different P4 targets, few target-specific changes are required. Individual targets use different P4 architecture models, i.e., the order or number of pipeline stages and the extern interfaces differ. For the former, this only requires changes to the structure of the program depending on the concrete model, e.g., v1model for t4p4s and SimpleSumeSwitch for the NetFPGA. The extern interfaces are challenging for SYN mitigation as our solutions require a cryptographic hash function.

Cookie Calculation: To calculate a standard-conform cookie, the P4 target needs to offer functionality for generating a timestamp used for replay protection and hash calculation. We use the insights gained in Section 5.2 for the choice of hash function. Integration of cryptographic hash functions in P4 data planes is currently possible for software, NPU, and FPGA targets [7] as discussed in Section 5.2. The integration and use of, for instance, SipHash as extern on the software target is straight forward as it can be added as library to the hardware-dependent t4p4s code. We choose SipHash as it is designed for short inputs like packet data [200], while achieving good performance when integrated into P4 targets [7]. Although the NPU target includes a crypto accelerator for SHA1 and SHA2, it was unavailable on our card, wherefore we opted to integrate a SipHash function as extern. The NPU allows to add extern functions written in a variation of C used to program its processing cores. We are not aware of a P4 ASIC that supports cryptographic functions.

When raw timestamp access is not possible, replay protection can also be achieved by using a table containing a counter, which is updated by the control plane.

Whitelisting: The easiest approach to perform whitelisting in a P4 program is to use a match-action table. The data plane informs the control plane through a digest message whenever a flow or IP address should be whitelisted and the control plane inserts an according table entry. The disadvantage is that this communication results in delay until the rule takes effect in the data plane.

An alternative approach that does not include a control plane is to use a Bloom filter data structure built with registers. This precludes additional communication delays, however, depending on the target and implementation of the register extern, registers might require more resources or have slower read/write access times compared to match-action table entries. Furthermore, additional complexity is required to maintain the state in the Bloom filter, in particular, evicting outdated entries. As both methods have trade-offs, which method to use depends on the concrete target device and deployment requirements, e.g., if the delay to insert table entries is acceptable.

Buffering Packets: Stalling the initial TCP data segment when using the SYN cookie strategy is not possible with P4, as P4 has no construct to write entire packets to memory. Stalling is only possible, if the target provides an extern for this task. To perform this operation at line-rate in a programmable ASIC, FPGA, or NPU, however, is unlikely due to the memory capacity and memory bandwidth required.

An alternative is to use a secondary COTS device as storage server, programmed using, for instance, a framework like DPDK. In this *slow-path* scheme, if a packet needs to be stalled, the proxy forwards this segment to the storage server. Once the handshake is finished, the proxy informs the storage server, e.g., via the controller, to transmit the stalled packet. The downsides are the increased complexity for the hardware setup and the necessary controller logic.

As the underlying problem can be circumvented by using a TCP zero window or active notification by resending the SYN/ACK as presented in Section 5.3.2, we did not implement this slow-path solution.

5.3.7 Evaluation

Software packet processing frameworks allow high flexibility through raw packet handling and easy addition of complex data structures. However, they require careful development and are limited to software platforms. Using a standardized DSL like P4 makes program development simpler and portable to ASIC, FPGA, and NPU devices, but comes at the cost of flexibility as the set of offered functions is limited. The following evaluation uses empirical measurements to compare performance indicators of the discussed implementations. As our software implementation uses the same underlying framework as t4p4s, we compare the P4 solution to a complex implementation using raw packet handling and kernel-bypass techniques. We compare our implementations regarding latency with the network stack of a standard Linux 4.19 with SYN cookies enabled.

Key Performance Indicators

The primary performance indicator for a SYN proxy is the total SYN flood processed. From a user perspective, e.g., the number of HTTP requests served without packet loss and the overall

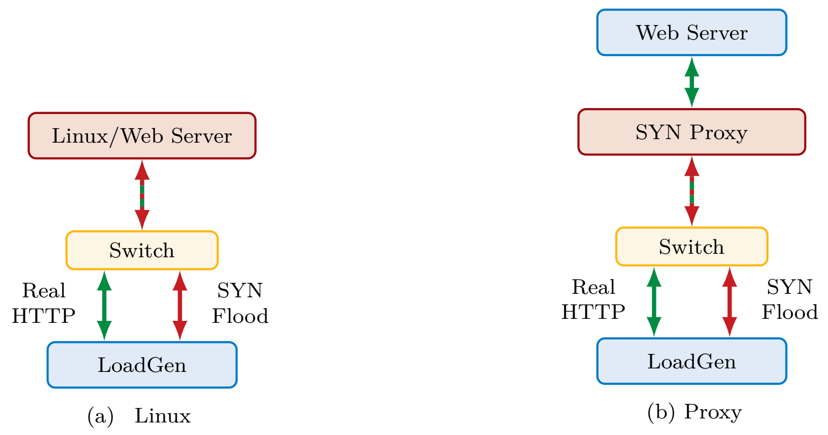


Figure 5.15: Measurement setup for SYN flood mitigation mechanisms

latency are a concern. While, in general, the latency of a device or application when operating in an overload scenario is not of interest, in the case of a SYN proxy, it is highly likely that the proxy will reach an overload state during a high volume attack. We, therefore, analyze latency values in low—no SYN flood,—middle—50 % of total processed SYN flood,—and overload scenarios.

Measurement Setup

As shown in Figure 5.15, the load generating host sends malicious SYN flood and legitimate HTTP traffic via two separate links to the DuT. Using a 10 GbE switch, the traffic is mixed so that malicious and legitimate traffic arrives at the DuT at the same port, i.e., are indistinguishable based on the ingress port. We use MoonGen [33] as load generator for the SYN flood traffic. A constant load of HTTP queries is generated using wrk2 [260]. All measurements run for 30 s, allowing for accurate latency results up to the 99.99 %-ile.

Depending on the DuT, the topology slightly differs. Linux as DuT cannot be configured to run as SYN proxy, wherefore, the web server is located on the same node as shown in Figure 5.15a. For all other scenarios using our SYN proxy implementations, the DuT forwards traffic classified as legitimate to a separate web server host as visualized in Figure 5.15b.

For measurements using software targets, the DuT is a COTS server, equipped with an Intel X722 NIC and an Intel Xeon Gold 6130 CPU clocked at 2.1 GHz. In other scenarios, the DuT is a server equipped with either a 10 GbE P4-programmable NPU or FPGA. We denote our libmoon-based software packet processing implementation as “lm”. For all measurements with the COTS system, we disabled turbo boost, set the CPU frequency to the maximum of 2.1 GHz, and pinned all traffic to one CPU core. When using Linux as DuT, the web server is pinned to a different CPU core as we are not interested in measuring the web server’s performance impact. The nginx web server (version 1.10.3) is limited to a single worker and CPU core and serves a 1 kB static website.

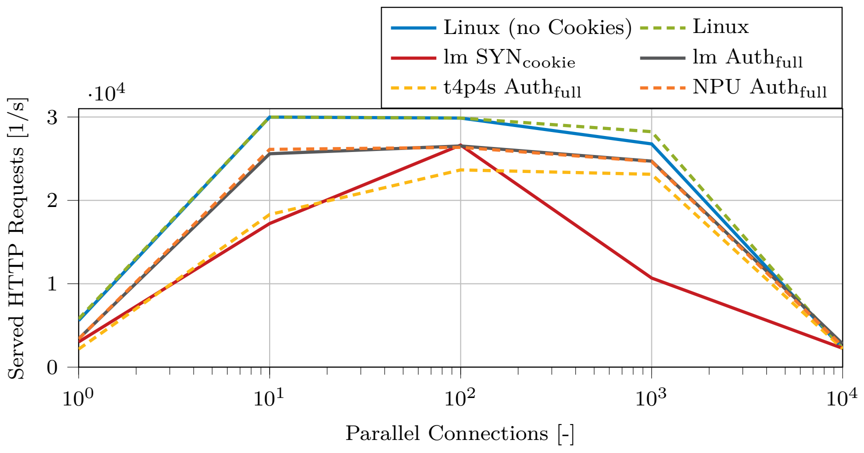


Figure 5.16: Maximum HTTP requests served for different number of parallel connections

Web Server Overload

Initial tests displayed in Figure 5.16 show that the web server is capable of processing up to 30 000 HTTP requests per second for 10 to 10 000 parallel connections. However, latency increases when using more than 1000 parallel connections or more than 4000 requests per second. We measured a reduced connection probability for more than 700 parallel connections, even when issuing only 100 HTTP requests per second. As we do not want to measure overload artifacts of the web server, we restrict our measurements to 100 parallel connections, issuing a total of 100 or 1000 HTTP requests per second.

Processed SYN Flood

Figure 5.17 shows the maximum SYN flood each implemented solution is able to process. For all implementations, the use of a cryptographic hash function is the limiting factor and reduces the maximum processed SYN flood by up to 50 %, which is comparable to our findings presented in Section 5.2 [7]. Due to the possibility to manually optimize and parallelize packet processing, the libmoon/DPDK implementation achieves up to 50 % better performance than the t4p4s implementation using P4. In contrast, the libmoon/DPDK implementation requires approximately 1000 lines of code and careful development and optimization. Only the hardware P4 targets are capable of processing up to 14 Mpps of SYN flood traffic when using the simpler $\text{Auth}_{\text{full}}$ strategy.

Batching

For the software implementations, batch processing has a significant influence on performance, shown in Figure 5.18. The raw packet handling of libmoon allows the complete processing of packets to be performed in batches. In particular, cookie values are calculated for the complete batch, reducing the overhead of C calls from libmoon using Lua's FFI. For t4p4s, the impact of batching is reduced, as each packet is processed by the P4 pipeline individually, although packets are received in batches. For all remaining measurements we have used the optimal batch sizes of 64 and 32 for libmoon and t4p4s, respectively.

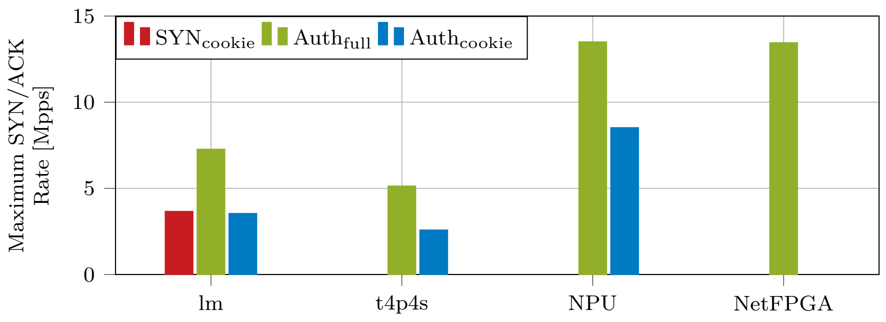


Figure 5.17: Maximum processed SYN flood traffic

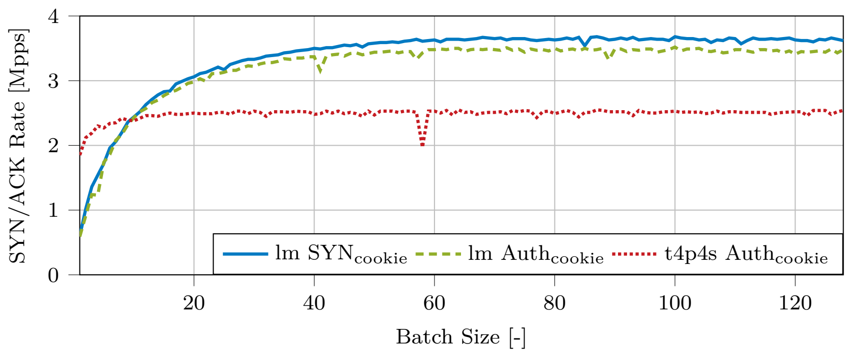


Figure 5.18: Processed SYN flood for different batch sizes on COTS system

CPU Core Scaling

The COTS proxy implementations do not have to share state between cores when using receive-side scaling to assign flows to CPU cores. This results in the linear scaling with CPU cores as demonstrated for the libmoon-based implementation in Figure 5.19. All other prototype implementations scale equally linearly with the number of cores or devices used (not shown). Consequently, when using enough cores or devices, throughput close to line-rate can be reached, even when cookies are calculated.

Quality of HTTP Requests

We evaluate the quality of HTTP connection attempts when traversing the SYN proxy prototypes in regard to the probability of successfully establishing the connection and elapsed time.

Connection Probability: For all implemented solutions, the connection probability for 100 HTTP requests per second is at 100 % until the point of overload as shown in Figure 5.20. After this point, the probability slowly drops. As the web server is not overloaded, all requests reaching the server are served. However, with increasing SYN flood, causing processing overload for the proxy, the chance that the proxy is able to process and forward legitimate traffic drops. As the NPU is able to process the SYN flood at almost line-rate, no HTTP message is lost.

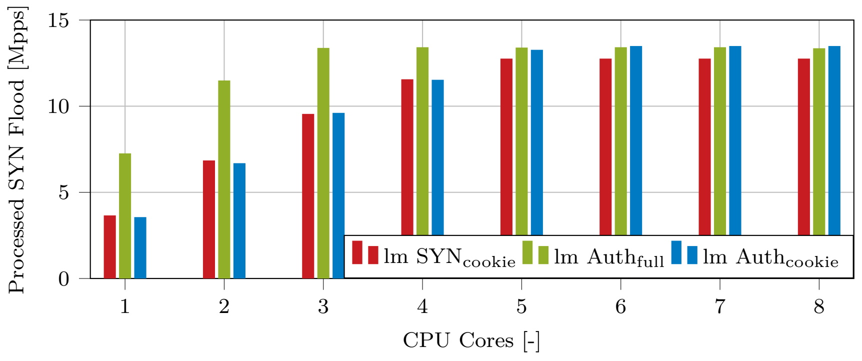


Figure 5.19: Multi-core scaling for software system

More requests per second reduce the connection probability during overload. This is due to the same number of parallel connections being used. The number of connections experiencing a timeout remains the same, however, in the case of having more requests per second for a given connection, one timeout has a larger impact. Increasing the number of parallel connections reverts this effect, as the impact of an individual connection experiencing a timeout is reduced.

SYN cookies are not able to process the increased amount of HTTP traffic even for a small SYN flood. This is due to the increased complexity of the state keeping, causing problems for rapid port reuse as previous state still persists.

To reduce clutter, we do not show $\text{Auth}_\text{cookie}$. For these strategies, the slope is the same as for their respective Auth_full counterparts, however, shifted to the left. This shift is correlated to the reduced maximum SYN flood that can be processed. For the NPU platform the probability starts dropping when reaching approximately 10 Mpps.

Connection Completion Latency: Connection latencies for the best case, i.e., no SYN flood, average case, i.e., 50 % SYN flood relative to maximum processed flood, and worst case, i.e., overload, are shown in Figure 5.21. For most scenarios, the median latency is between 1 and 1.4 ms, while for the no flood and 50 % cases a long-tail up to 4 ms is visible. Both implementations for the CPU target show sporadic outliers and a long-tail behavior with up to a second already for the 90 %-ile during overload. The long-tail is expected due to batch processing and operating system interrupts typical for software packet processing frameworks like DPDK [28]. Due to the lower probability when issuing 1000 HTTP requests per second for 100 parallel connections, the median latency during overload is above one second.

The exception is the NPU, showing latencies between 1 and 4 ms without outliers even in the worst case. Further, the latency for no flood is worse compared to when increasing the SYN flood load. We attribute this to specifics of the architecture, improving processing when increasing the backpressure on internal buffers or when reducing idle cycles caused by energy-saving features.

For the Auth_full implementation on the NetFPGA SUME, the latency for up to 1000 HTTP requests per second was stable between 1 to 4 ms without outliers up to a SYN flood of 1.5 Mpps (not shown). However, for reasons unknown to us, the program stopped working for higher

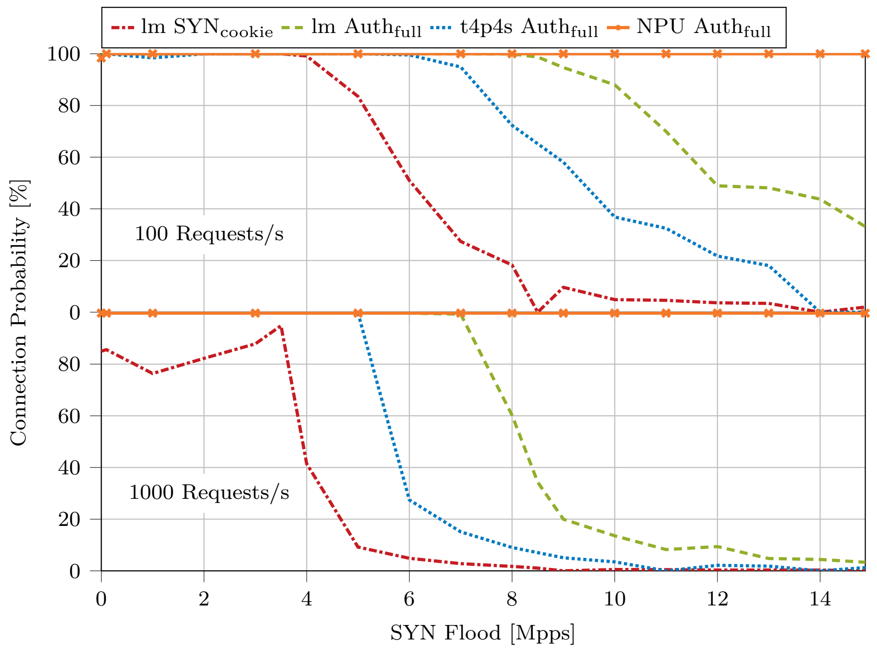


Figure 5.20: Connection probability for 100 and 1000 HTTP requests per second

loads of mixed traffic. Related work shows that the NetFPGA SUME, in general, is able to run P4 programs of higher complexity with latency below 10 μs and no long-tail. This is even possible when modifying the P4 architecture of the NetFPGA to integrate a SipHash or SHA3 core, which can be used to hash even complete packet data. [7]

Whitelisting—Scaling Match-Action Tables

As we have shown in Section 4.4.4, the number of entries in match-action tables on P4 targets scale in regard to performance and resource usage [6]. For instance, more than one million 32 B exact match entries for the t4p4s DPDK can be inserted. On a software target, the bottleneck becomes the CPU cache, quickly reducing the performance when exceeding L3 cache capacity. However, adding even more entries is still possible. On a hardware target, exhausting the available resources is a hard limit. While one entry depends on its match size, i.e., the key, action data, and action performed, sufficient resources for up to several hundreds of thousands of entries are available on, e.g., the commercial Intel Tofino ASIC, neglecting resources required by the rest of the P4 program. [6]

If the resources for whitelisting are restricted by the P4 target, whitelisting can also be aggregated. Instead of whitelisting individual flows identified by the 5-tuple, whitelisting can be performed based only on the source IP or even the complete source subnet. This results in a trade-off between whitelisting granularity and resource consumption.

FPGA Resource Consumption

For the NetFPGA SUME, the synthesized P4 proxy program only uses up to one third of the total resources. This leaves enough resources to further enhance the program to defend against other

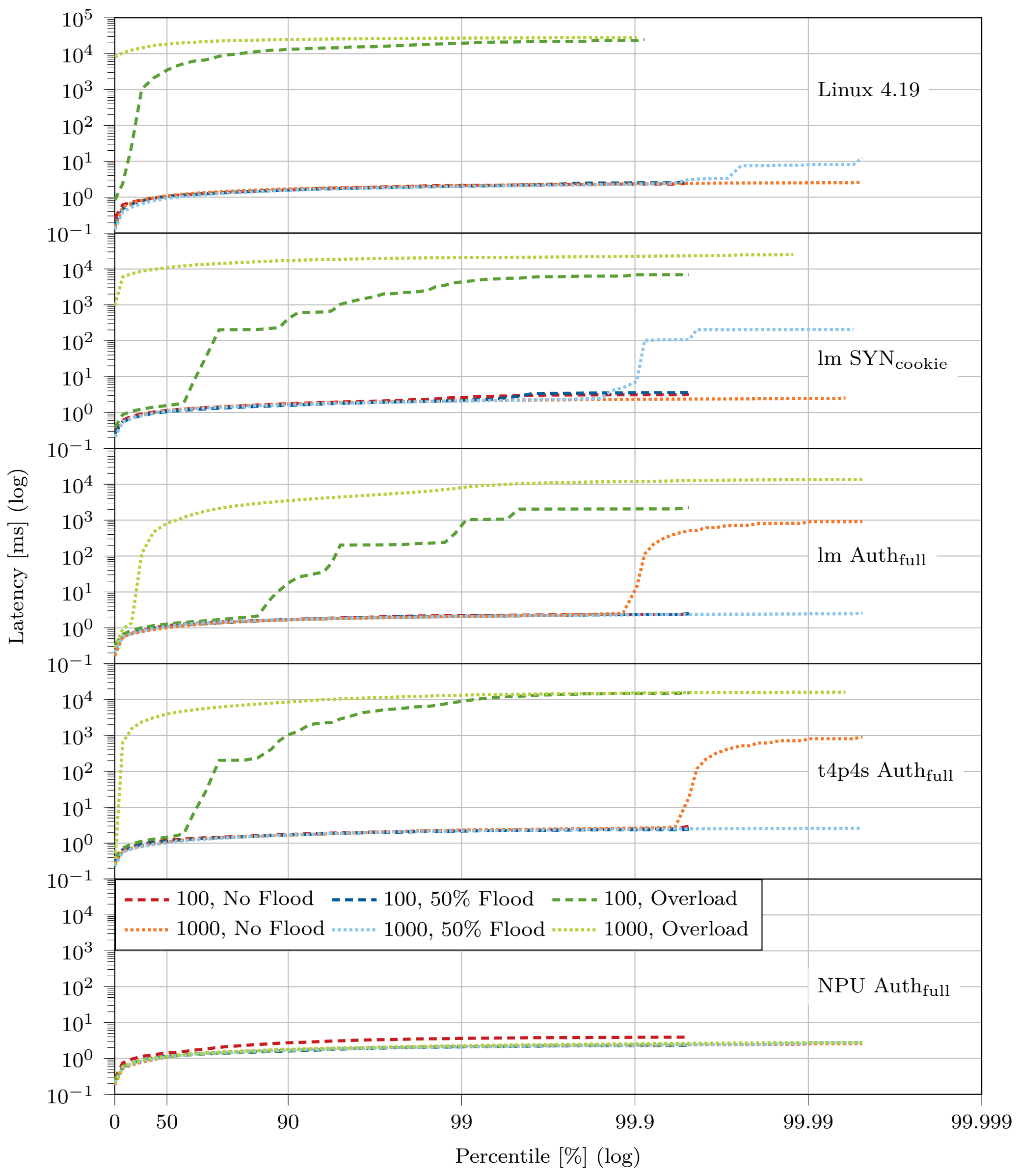


Figure 5.21: High Dynamic Range latency for 100 and 1000 HTTP requests per second

attacks. Further, it would allow to add the modified architecture model including a SipHash hashing module presented in Section 5.2.5 into the data plane [7]. As shown in Table 5.5, this would only require 2 % of the total resources and would allow to even perform Auth$_{cookie}$.

Programming Experience

Implementing SYN mitigation mechanisms in programmable data planes is drastically simplified using the P4 language compared to raw packet handling using, for instance, libmoon/DPDK. Its well-defined programming language, architecture model, and extern interfaces aid the im-

	LUTs		Registers		BRAM	
	Abs.	%	Abs.	%	Abs. [kB]	%
$\text{Auth}_{\text{full}}$	98 343	22.71	156 678	18.08	18 450	34.87
Data presented in Table 5.5 [7]						
Baseline	64 533	14.90	109 783	12.67	16 362	30.92
SipHash	66 380	15.32	114 282	13.19	17 460	32.99
SHA3-512	73 449	16.95	118 689	13.70	17 460	32.99

Table 5.8: Resource utilization of SYN proxy on NetFPGA SUME compared to P4 hash implementations presented in Section 5.2.5

plementation of the overall logic independent of the target platform, without the need to take care of memory and buffer management. Furthermore, P4 supports hardware targets, where implementing such complex processing was previously infeasible or required domain-specific knowledge. However, depending on the concrete target, it is subjective and domain-specific influences of the target device can cause challenges. For instance, when using t4p4s, knowledge about the code architecture, DPDK, and C is required, while debugging the NetFPGA SUME requires VHDL expertise. All targets provide different programming frontends, ranging from Linux command-line scripts for t4p4s and the P4→NetFPGA toolchain, to programmer studios for the NPU. Thereby, an expected difference in the user experience can be noted between research projects and commercial products. While using an DSL like P4 speeds up the prototyping, deployment, and adds portability, using a software packet processing framework allows for fine-grained optimizations. Explicitly managing buffers and performing only necessary operations on raw packets is time consuming, but the optimizations result in increased throughput and reduced latency and resource consumption.

5.3.8 Conclusion

SYN floods are still the predominant traffic for high-volume DDoS attacks on the Internet and will likely remain so in the future as the root-cause, a flaw in the TCP handshake, cannot be fixed. The client puzzle—including a cryptographic hash value—as part of SYN cookies or SYN authentication is the single effective SYN-specific defense strategy that is not based on blackholing or heuristics. It guarantees that no malicious connection attempts are successful, while not falsely rejecting legitimate requests. Due to the computational complexity of cryptographic hash functions, this is the bottleneck for these implementations. The SYN cookies and SYN authentication strategies both offer similar protection capabilities, with the former being fully transparent for TCP clients and the latter being simpler to implement. Performing the attack mitigation as a SYN proxy running on a dedicated node is scalable. This setup allows the protection of entire networks without consuming resources of the end hosts.

Our programmable data plane prototypes have shown that SYN authentication, when used in a proxy setup, can mitigate SYN floods at 10 Gbit/s line-rate. The P4 solutions are easy to implement and can be ported to different target platforms, in particular, hardware devices, which achieve end-to-end connection latencies below 5 ms with low jitter. The P4 software target achieves latencies comparable to implementations using the libmoon software packet processing framework. However, using raw packet handling allows for further optimizations compared to using a DSL for data plane programming, at the cost of added complexity and no portability to hardware devices.

We conclude that effective and efficient SYN flood mitigation on modern data planes is possible. With both mitigation strategies, SYN cookies and SYN authentication, performing equally well, we recommend SYN authentication, being the simpler one to implement. The crucial limiting factor for hardware data plane solutions is the availability of a suitable cryptographic hash function. However,we have shown that this is possible on different software and hardware targets.

5.4 Key Results

In this chapter, we have shown the value of performing cryptographic operations, in particular, hashing, in programmable data planes. These functions are essential for many modern protocols, especially when used in industrial applications. There, authenticated communication is already required at the data link layer, for instance, using IEEE 802.1AE MacSec. Other uses for cryptographic hashing include DoS mitigation approaches in the form of client puzzles.

Cryptographic hashing can be integrated into programmable data planes using different approaches, each with up- and downsides. For instance, as extern function or dedicated pipeline module that is a part of the architecture model. Alternatively, hashing or other functionality could in theory be implemented based on standard P4 constructs, e.g., using match-action tables. The best approach is highly target-dependent. Our prototype implementations show that integration is simple for the software target. Hardware targets offer less flexibility and extensibility. Processing pipelines might have restrictions for the amount of data that can be processed in externs. On the other hand, a pure P4 solution might limit the resources available for the remaining program. The most challenging platform are ASICs. They cannot be changed easily, however, efficient cryptographic operations in the data plane are possible as shown by bitcoin miners. It is merely a matter of integrating these hardware blocks with the remaining pipeline. The performance evaluation of our prototypes for software, NPU, and FPGA platforms have shown promising results.

We have used our prototypes to show the usability of hashing in programmable data planes by implementing high-performance SYN flood mitigation approaches. TCP SYN floods are still a difficult problem to handle as they are typically part of bigger DDoS floods. Many potential solutions have been proposed, however, our literature research shows that only a cryptographically secured client puzzle can reliably protect against SYN floods. All other approaches can be circumvented or only mitigate a certain percentage of the overall attack as they use heuristics to determine malicious traffic.

To highlight the simplicity of implementing SYN flood mitigation in programmable data planes, we also implemented the same approach using a classic software packet processing framework based on DPDK. Furthermore, this allows a closer performance comparison with the P4 program for the DPDK target. While the P4 programs were not entirely portable to different target platforms due to different architecture models and extern interfaces, the core of the program can be reused. Still, the lines of code for the P4 solutions were drastically less compared to the manually programmed DPDK solution. This allows rapid prototyping using a high-level programming language. The advantage of using the low-level packet processing framework lies in the performance as individual parts of the program can be optimized.

Summarizing, cryptographic operations integrated with programmable data planes opens up new and creative solutions for applications with security aspects. Therefore, we argue, that it is beneficial to include basic aspects of cryptographic externs using standardized interfaces in the P4 specification, for instance, as part of the PSA.

5.5 Statement on Author's Contributions

The motivational introduction in Section 5.1 is based on a collaboration between Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle [7]; and joint work between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, and Georg Carle [8]. The author significantly extended this section with background information on different categories of hash functions.

Section 5.2 is based on a collaboration between Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle [7]. The artificial performance benchmark of various hash functions presented in Section 5.2.3 was performed by the author. The benchmark is based on a framework by Fabien Geyer, which the author extended to include more tests with different hash functions. The discussion of hashing interfaces in P4 (Section 5.2.2), choice of hash functions (Section 5.2.3), and integration of hash functions (Section 5.2.4) was significantly extended and generalized by the author compared to the original publication. The integration of hash functions for the t4p4s target was initially performed by Philipp Hagenlocher as part of a research work, co-supervised by the author. The author further extended t4p4s by integrating additional hash functions. The performance evaluation of the integrated hash functions for the t4p4s and NFP-4000 targets presented in Section 5.2.6 was performed by the author, using a framework developed by Fabien Geyer and the author. The discussion about the FPGA-based target was shortened and simplified compared to the original publication, to better fit the focus on software targets in this thesis. Table 5.3 and Figures 5.4, 5.6, and 5.7 were extended for this work, providing further insights for the investigated target platforms.

Section 5.3 is based on a collaboration between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, and Georg Carle [8]; and a joint work between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle [19], which itself is based on the Master's Thesis [20] by the author. The background information on SYN flood mitigation (Section 5.3.1), deployment scenarios (Section 5.3.2), and related work (Section 5.3.3) was extended and unified for this work by the author and goes beyond the scope of the individual publications. Section 5.3.5 is based on a collaboration between Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle [19]. The complete implementation of the DPDK-based SYN proxy was performed by the author. The explanations in Section 5.3.5 were extended for this work. The prototype implementations for the t4p4s target presented in Section 5.3.6 were created by the author and are based on platform-independent P4 programs developed during an interdisciplinary project work by Bassam Jaber, which the author co-supervised. The author performed the performance measurements discussed in Section 5.3.7 using the mentioned programs and implementations contributed by Henning Stubbe and Sebastian Gallenmüller for the NetFPGA SUME and the NFP-4000 targets, respectively. Figures 5.14, 5.16, 5.18, and 5.19 were added and Figures 5.13, 5.15, 5.17, 5.20, and 5.21 were extended for this work by the author. Based on the addition of these Figures, the analysis goes significantly beyond the scope of the original work.

Chapter 6

Conclusion

In this chapter, we summarize the findings presented in this thesis. We reiterate our initially formulated research questions and outline our key results for the topics of reproducible testbed orchestration, automated modeling of data plane components, and the integration of cryptographic hashing into programmable data plane targets. Finally, we discuss areas with open questions, suitable for potential future work.

6.1 Contributions

The following summarizes the key contributions for the research questions introduced in Section 1.1.

RQ1: How can we perform reproducible measurements for a heterogeneous set of software and hardware data planes?

In Section 3.2, we have presented the pos testbed controller, which not only encourages, but enforces reproducible experiments. Each experiment consists of scripts for setup and measurement phases. These scripts provide the sequence of commands that should be executed, e.g., for configuration or execution of the measurement. Further, variables can be defined per measurement run, which provide further details for each individual measurement. pos explicitly supports the execution of measurement series, using the so-called loop mode. For every combination of loop variables, a measurement is executed, e.g., scaling the packet rate and packet size throughout all measurements. Fully specifying the experiment, including setup, variables, and measurements, is vital for achieving not just repeatability, but also reproducibility. Furthermore, all measurement artifacts are collected for further processing and evaluation. Bundling this data together for easy publishing results in replicable experiments.

The introduced testbed controller and enforced workflow are used for all experiments throughout this thesis. In Section 4.3.3, we extended the evaluation component of pos with a framework for automatically deriving mathematical models.

RQ2: How can we operate multi-user testbeds for reproducible experiments using heterogeneous hardware?

pos is not only designed for reproducible experiments, but also to manage heterogeneous testbeds, accessed by a variety of users with different backgrounds. Thereby, the testbeds are not only intended for performance measurements of wired devices. pos supports experiments using wireless devices, distributed setups, or pure computation tasks, using one or more nodes. All resources are shared by users. To avoid interferences, nodes are allocated only to one experiment at a time. This is automatically enforced by pos. The setup of pos is also automated and reproducible, decreasing the effort for testbed administrators.

We currently employ pos to manage more than 130 heterogeneous devices, including COTS servers, virtual machines, power distributions units, switches, and Raspberry PIs. Further, we have shown that pos can be used for hybrid teaching and measurement testbeds in Section 3.2.5.

RQ3: How can we model the performance of individual data plane components in an automated fashion?

We proposed a modeling approach to estimate the behavior of data plane components. As different target platforms require different models or new parameters for the same model, we automated the process of measuring and deriving the model. The framework has a target- and a testbed-dependent component. In this work, we implemented it for pos-controlled testbeds, achieving full automation and reproducibility. Our model-first approach is optimized for mathematically fitting the model for the measurement data using a curve fitting approach. Thereby, different metrics can be used as plugins to determine the quality of the resulting fitting. One important factor is the complexity of the model. We use metrics based on AIC, penalizing complex functions with multiple free fitting parameters. Typically, a simpler function better represents the actual system behavior. However, certain events, like overload or memory exhaustion, might drastically alter the behavior of a network device, requiring complexity in the model. Therefore, our derived models may consist of multiple partial models.The points that split the partial models are derived by calculating second derivatives and applying different algorithms to process the data.

We have implemented the target-specific component for the t4p4s P4 target presented in Section 4.4. We then applied the automated framework to model components and features typically found in data plane programs. The resulting models show that the key components of such programs on a software system are match-action tables. Thereby, we differentiated the match type, as each type is implemented using different data structures. The model for exact matches revealed multiple different events that alter the scaling of CPU cycles. Using a resource model we were able to predict the memory requirements of the data structure in relation to cache sizes. Similarly, for ternary and LPM matches, we could identify and model the impact of the underlying data structures and memory requirements.

RQ4: How can we determine the packet that takes the worst case path through the data plane?

Based on the models for individual data plane components, we proposed modeling the performance of entire processing paths through the data plane. Determining these paths is possible based on the data plane program's CFG. For the path models, we added up the costs of individual components as determined by our component models. Doing so for all paths allows to determine, e.g., the worst-case path through the data plane.

Generating traffic matching a specific path is a complex problem. This requires information provided by the control plane as, e.g., table entries are only known during runtime. Only this state information defines the criteria which packet chooses which path.

In Section 4.4.5, we used the calculated component models for the t4p4s target to derive models for each path through two sample programs. Using measurements, we generated packets that match only a single path and verified the accuracy of the path models. Thereby, the models overestimated between 20 and 30 % CPU cycles. Our approach of adding the cost for individual components does not take into account compiler optimizations. Furthermore, effects identified for components in isolation might be different for complete programs. Extern functionality further complicates the process of generating packets for a certain path, especially if language constructs change the state of the device.

RQ5: How can we extend programmable data planes with cryptographic hashing functionality?

By analyzing related work and current use cases, we have demonstrated the importance of cryptographic hash functions in today's networks. Consequently, this functionality is also important for applications based on programmable data planes. Supporting cryptographic hashing enables new protocols and applications, also used, e.g., in industrial networks, to be moved to the data plane, taking full advantage of P4's flexibility and portability. We have shown, that different approaches exist to integrate such functions, preferably either as extern or as part of the architecture model. Depending on the concrete target platform and underlying technology, different challenges may arise, resulting in advantages and disadvantages for either of the approaches. Our performance evaluation has demonstrated that our prototypes are capable of up to 10 Gbit/s line-rate. One unsolved area are ASICs, which do not allow the integration of additional functionality. For such targets, either the hash function has to be directly included in the design and manufacturing process, or the functionality can only be provided through a separate chip or processing node.

Based on the results presented in Section 5.2, we conclude that cryptographic hash functions can be integrated with programmable data planes. We argue, that they warrant a direct mention in the P4 or PSA specifications due to their importance in networking applications. This would encourage further efforts in simplifying the interfaces between hash functions and data planes and, ultimately, cause more chip manufacturers to include them in programmable data plane targets.

RQ6: How can we defend against TCP SYN floods from within the data plane?

In Section 5.3, we have discussed that SYN floods are and will remain a threat for computer networks. Thereby, only mitigation approaches that employ a cryptographically secured client puzzle can reliably filter such attack traffic. Other approaches either use heuristics or statistics, resulting in false positives or negatives, or can be circumvented by clever attackers adjusting the attack traffic. Using our presented prototypes for cryptographic hash functions, we implemented the most promising mitigation strategies for different P4 software and hardware platforms. Furthermore, we implemented the same defense approaches using a classical software packet processing system based on DPDK, based on the dynamic protocol stack presented in Section 3.3. Our discussion on implementation complexity and performance has shown that the SYN flood mitigation solutions can be implemented using both approaches, suitable for high-performance networks and large attack loads. However, the data planes programmed using a standardized DSL like P4 have advantages. The same attack mitigation program can be ported to hardware platforms, only requiring small changes for architecture-specific components. The upside of the handcrafted solution based on the packet processing framework is the potential for optimizations, leading to better performance.

The implemented SYN flood mitigation strategies presented in Section 5.3 provide another argument for the inclusion of cryptographic hash functions in programmable data planes. Our approach for filtering traffic using a client puzzle to differentiate malicious from legitimate traffic can be used for protecting against other, e.g., DoS threats.

6.2 Future Work

The pos testbed controller only provides limited access for device configuration outside of the OS. To fully control and configure all aspects of each testbed, for instance, BIOS or firmware settings have to be adjusted. Adding support for such interfaces would allow adding even more heterogeneous devices to pos-controlled testbeds. Furthermore, device configuration would be further automated and specified, getting closer to full reproducibility.

The component modeling framework does not yet support functions to model probabilistic distributions, like gaussian or trapezoid functions. However, this would be necessary to model, for instance, latency histograms. Adding these functions raises several challenges. Often, probabilistic functions have restrictions, either for their defined values or the range of single parameters. Specifying these is currently not supported. Further, the integral of a probabilistic modeling function has to be one by definition. This requires a new approach for splitting the overall model into multiple parts. For instance, instead of using multiple distinct models separated by splitting points, multiple probabilistic models could be added up using weights, such that the overall integral remains one. Lastly, better metrics to quantify the fit, like maximum likelihood, should be used. Integrating these metrics into the component modeling framework is trivial due to modularized plugin nature of metrics.

A related aspect are the ranges of free fitting parameters for modeling functions. Currently, a minimum value is used due to the measurement and modeling accuracy. An improvement would be individual per-parameter ranges. For instance, for a high degree polynomial, whereby the highest degree fitting parameter is smaller than 0.1, the polynomial could be reduced to the next lower degree polynomial. Essentially, these restrictions would provide further semantics to each functions' fitting parameters.

Our approach for analyzing and modeling paths of data plane programs is not fully automated as the necessary tools for deriving the CFGs do not yet support all architecture models and P4 language components. This problem extends towards the automated generation of packets that match a certain path. Further research and implementation effort is required for such tools to reliably generate packets for all paths, especially for cases that require control plane information or when packets modify the state of the device.

We have applied the automated component modeling framework to the software-based t4p4s target. In a next step, this framework could be applied to other, e.g., hardware-based P4 switch targets, like the Netronome NFP-4000 SmartNIC, the NetFPGA SUME, or the Intel Tofino chip. Depending on the concrete target, other metrics than packet rate and CPU cycles are of interest, like resource consumption for the ASIC platform. Investigating component model parameters for other available software targets, like P4 transpilers for eBPF or XDP, would allow a comparison with our presented results.

Our implementations for integrating cryptographic hash functions into programmable data planes are of prototype nature. In particular, they use open-source implementations of hash functions, which might not be optimized or ideal for a certain target platform. Similar to Malina et al. [217], future work includes further simplifying the interface to use cryptographic functions and integrating optimized hashing cores.

Chapter A

Appendix

A.1 List of Acronyms

$\mathbf{Auth}_{\mathbf{cookie}}$	$\text{Auth}_{\text{cookie}}$. SYN authentication strategy using a cryptographic cookie
$\mathbf{Auth}_{\mathbf{full}}$	$\text{Auth}_{\text{full}}$. SYN authentication strategy using a full handshake for authentication
$\mathbf{Auth}_{\mathbf{Invalid}}$	$\text{Auth}_{\text{Invalid}}$. SYN authentication strategy expecting a specific reaction from the client in response to invalid data for authentication
$\mathbf{Auth}_{\mathbf{TTL}}$	Auth_{TTL}. SYN authentication strategy using additional fingerprinting techniques for authentication
AFDX	Avionics Full-Duplex Switched Ethernet. Data network based on Ethernet
AIC	Akaike information criterion. Estimator for the relative quality of a set of statistical models
API	Application programming interface. Interface allowing multiple applications to interact
ASIC	Application-specific integrated circuit. Custom-made integrated circuit for particular use case
bmv2	Behavioral model version 2. Reference P4 software switch
BPF	Berkeley Packet Filter. Linux in-kernel virtual machine
BRAM	Block random access memory. Random access memory typically used in FPGAs to store large amounts of data
CDF	Cumulative distribution function. Plot variation displaying the empirical cumulative distribution function of the data

CFG	Control-flow graph. Presentation of all possible execution paths of a computer program using graph notation
CLI	Command-line interface. Input for a computer program in the form of lines of text
COTS	Commercial off-the-shelf. Commercially available product
CRC	Cyclic redundancy check. Checksum used to detect errors in digital data
DDoS	Distributed denial-of-service. Attacking a victim with traffic originating from several different sources
DMA	Direct memory access. Feature of operating systems to allow I/O devices to access main memory independent of the CPU
DoS	Denial-of-service. Attacking a victim server or network with the goal of making it unavailable for its intended use case
DPDK	Data Plane Development Kit. Framework for creating high-performance packet processing applications
DSL	Domain-specific language. Programming language designed for a specific application domain
DuT	Device under test. Device of interest in a measurement
eBPF	Extended Berkeley Packet Filter. Linux in-kernel virtual machine
EMD	Earth mover's distance. Quality metric to compare probability distributions
FFI	Foreign function interface. Interface of a programming language to call functions of another language
FPGA	Field-programmable gate array. Configurable integrated circuit consisting of programmable logic blocks and interconnects
GbE	Gigabit Ethernet. Transmitting Ethernet frames at Gbit/s rates
GRE	Generic Routing Encapsulation. Tunneling protocol developed by Cisco
HDL	Hardware description language. Computer language to describe the architecture and behavior of electronic circuits
HDR	High dynamic range. Variation of histograms, focusing on high percentiles
HMAC	Hash-based message authentication code. Message authentication code using a cryptographic hash function
IPMI	Intelligent Platform Management Interface. Interface to manage a computer remotely, out-of-band, and independent of the system's CPU

JIT	Just-in-time. Compilation of computer code during runtime of the application
KPI	Key performance indicator. Metric for a performance measurement
LPM	Longest prefix matching. Algorithm used by routers in Internet Protocol-based networks to select the best route from the forwarding table
LUT	Look-Up Table. Programmable block used typically in FPGAs for performing boolean algebra
MAC	Message authentication code. A cryptographic construct to verify a message's data integrity and authenticity
MAPE	Mean absolute percentage error. Estimator to predict statistical accuracy
NFP	Netronome Flow Processor. A network processing unit chip architecture sold, e.g., by Netronome
NIC	Network interface card. Network adapter connecting a computer to a computer network
NOS	Network operating system. Operating system for a network device
NPU	Network processing unit. Integrated circuit with feature-set targeted at network devices
OS	Operating system. Software performing basic tasks of a computing system
OvS	Open vSwitch. Virtual switch software
P4	Programming Protocol-independent Packet Processors. Domain-specific language to program network data planes
PISA	Protocol-Independent Switch Architecture. Single pipeline forwarding architecture used by the 2014 version of P4
POF	Protocol-oblivious Forwarding. Enhancement to OpenFlow-based SDN forwarding architectures
pos	plain orchestrating service. Testbed controller software for multi-user heterogeneous testbeds
PSA	Portable Switch Architecture. P4 target architecture describing common capabilities of network switches
SDN	Software-defined networking. Approach to programmatically and dynamically manage computer networks

sMAPE	Symmetric MAPE. Variation of MAPE estimator to predict statistical accuracy
SRAM	Static random-access memory. Volatile random-access memory
TCAM	Ternary content-addressable memory. Specialized high-performance memory using three inputs
TCB	Transmission Control Block. Data structure used by the TCP stack to track connection information
uBPF	Userspace eBPF. eBPF virtual machine running in Linux user space
VPP	Vector Packet Processing. Framework to build network switches and routers using vector processing
XDP	eXpress Data Path. High-performance data path based on eBPF in the Linux kernel

A.2 List of Figures

2.1 Protocol-Independent Switch Architecture model 18
2.2 Popularity of P4 and programmable data planes in related work 25

3.1 Abstract experimental process 30
3.2 Measurement setup used throughout this thesis 30
3.3 Common administration, user and node life cycles of experimental testbeds . . . 33
3.4 Steps performed during an experiment using the pos testbed controller 35
3.5 pos experimental workflow 38
3.6 pos configuration and initialization interfaces 41
3.7 iLab administration and usage life cycles 46
3.8 iLab infrastructure 47
3.9 Weekly isle usage during iLab1 course 52
3.10 Peak performance when encapsulating Ethernet frames using VXLAN 66
3.11 Performance when copying different amounts of consecutive bytes 67
3.12 Performance when using low-level DPDK structures versus libmoon's protocol stack 67

4.1 White-box view of a generic packet processing system s 72
4.2 Model for a packet processing system s 72
4.3 Modeling of an individual P4 data plane component 74
4.4 Modeling of an individual P4 data plane path 75
4.5 Automated measurement framework for data plane component modeling 82
4.6 Overfitting for a clearly linear dependency 86
4.7 Drastic change in device behavior requiring two individual fittings 88
4.8 t4p4s' internal traffic flow 92
4.9 t4p4s baseline scaling of packet rate with CPU frequency 92
4.10 Baseline latency of t4p4s for different CPU loads on a single CPU core 94
4.11 Scaling parsed header fields 95
4.12 Scaling field writes 96
4.13 Scaling added header bytes 97
4.14 Scaling number of match-action tables 98
4.15 Scaling number of exact match entries 100
4.16 Scaling number of ternary match entries 102
4.17 Scaling number of LPM match entries 103
4.18 Scaling the number of exact match keys 104
4.19 Scaling amount of action data 105
4.20 Influence of different data types to represent a 16 bit exact match key 105
4.21 Relative error of the model for scaling different numbers of tables 108

5.1 Performance benchmark results of selected hash functions on COTS CPU 123
5.2 Hashing functionality as extern 125
5.3 Hashing functionality integrated with architecture model 126

5.4 Throughput when using SipHash and SHA512-based functions for hashing full Ethernet frames, relative to 10 GbE line-rate 130
5.5 Median latency for NetFPGA SUME at 2.5 Gbit/s 131
5.6 Median latency for t4p4s-based CPU target 131
5.7 Latency distribution at 5 % and 80 % of respective maximum throughput using 64 B packets 132
5.8 Performance of Linux during SYN flood 140
5.9 Profiling of different Linux versions during a SYN flood 141
5.10 SYN proxy deployment scenario 142
5.11 Message Exchange for SYN cookie strategy 143
5.12 Message Exchange for SYN authentication strategy 145
5.13 Architecture of SYN flood mitigation in different data planes 149
5.14 Swap-out hash map 151
5.15 Measurement setup for SYN flood mitigation mechanisms 155
5.16 Maximum HTTP requests served for different number of parallel connections . . 156
5.17 Maximum processed SYN flood traffic 157
5.18 Processed SYN flood for different batch sizes on COTS system 157
5.19 Multi-core scaling for software system 158
5.20 Connection probability for 100 and 1000 HTTP requests per second 159
5.21 High Dynamic Range latency for 100 and 1000 HTTP requests per second 160

A.3 List of Tables

2.1 Comparison of techniques used by different high-performance software packet processing frameworks . . . 14
2.2 Comparison of different P4 target architectures . . . 23
2.3 Overview of different P4 targets investigated in this thesis . . . 24

3.1 Statistics about testbeds orchestrated using pos . . . 43
3.2 iLab lab room hardware resources . . . 51

4.1 Parameterized fittings showing overfitting . . . 86

5.1 Properties of investigated hash functions . . . 122
5.2 Hash function performance benchmark on COTS CPU . . . 124
5.3 Throughput for hashing full Ethernet frames of different sizes in percent, relative to 10 GbE line-rate . . . 129
5.4 Memory hierarchy of the NFP-4000 architecture . . . 130
5.5 Resource utilization for the NetFPGA SUME . . . 133
5.6 Comparison of SYN flood mitigation strategies . . . 139
5.7 Comparison of SYN cookie and SYN authentication in proxy setup . . . 144
5.8 Resource utilization of SYN proxy on NetFPGA SUME . . . 161

Bibliography

Work with author's contribution

[1] D. Scholz, “A Look at Intel’s Dataplane Development Kit”, in *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2014*, G. Carle, D. Raumer, and L. Schwaighofer, Eds., ser. Network Architectures and Services (NET), vol. NET-2014-08-1, Munich, Germany: Chair for Network Architectures and Services, Department of Computer Science, Technische Universität München, Sep. 2014, pp. 115–122. DOI: `10.2313/NET-2014-08-1_15`. [Online]. Available: `http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2014-08-1/NET-2014-08-1_15.pdf`.

[2] ——, “Diving into Snabb”, in *Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2016*, G. Carle, D. Raumer, and L. Schwaighofer, Eds., ser. Network Architectures and Services (NET), vol. NET-2016-09-1, Munich, Germany: Chair for Network Architectures and Services, Department of Computer Science, Technische Universität München, Sep. 2016, pp. 31–38. DOI: `10.2313/NET-2016-09-1_05`.

[3] D. Raumer, F. Wohlfart, D. Scholz, P. Emmerich, and G. Carle, “Performance Exploration of Software-based Packet Processing Systems”, in *Proceedings of Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen, 6. GI/ITG-Workshop MMBnet 2015*, vol. 8, Hamburg, Germany, Sep. 2015.

[4] S. Gallenmüller, D. Scholz, F. Wohlfart, Q. Scheitle, P. Emmerich, and G. Carle, “High-performance Packet Processing and Measurements (Invited Paper)”, in *10th International Conference on Communication Systems & Networks, COMSNETS 2018, Bengaluru, India, January 3-7, 2018*, IEEE, 2018, pp. 1–8. DOI: `10.1109/COMSNETS.2018.8328173`. [Online]. Available: `https://doi.org/10.1109/COMSNETS.2018.8328173`.

[5] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, “Performance Implications of Packet Filtering with Linux eBPF”, in *30th International Teletraffic Congress, ITC 2018, Vienna, Austria, September 3-7, 2018 - Volume 1*, IEEE, 2018, pp. 209–217. DOI: `10.1109/ITC30.2018.00039`. [Online]. Available: `https://doi.org/10.1109/ITC30.2018.00039`.

[6] D. Scholz, H. Stubbe, S. Gallenmüller, and G. Carle, “Key Properties of Programmable Data Plane Targets”, in *32nd International Teletraffic Congress, ITC 2020, Osaka, Japan, September 22-24, 2020*, Y. Jiang, H. Shimonishi, and K. Leibnitz, Eds., IEEE, 2020, pp. 114–122. DOI: `10.1109/ITC3249928.2020.00022`. [Online]. Available: `https://doi.org/10.1109/ITC3249928.2020.00022`.

[7] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle, “Cryptographic Hashing in P4 Data Planes”, in *2019 ACM/IEEE Symposium on*

Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019, IEEE, 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901886. [Online]. Available: https://doi.org/10.1109/ANCS.2019.8901886.

[8] D. Scholz, S. Gallenmüller, H. Stubbe, and G. Carle, "SYN Flood Defense in Programmable Data Planes", in *EuroP4@CoNEXT 2020: Proceedings of the 3rd P4 Workshop in Europe, Barcelona, Spain, December 1, 2020*, ACM, 2020, pp. 13–20. DOI: 10.1145/3426744.3431323. [Online]. Available: https://doi.org/10.1145/3426744.3431323.

[9] S. Gallenmüller, D. Schöffmann, D. Scholz, F. Geyer, and G. Carle, "DTLS Performance - How Expensive is Security?", *CoRR*, vol. abs/1904.11423, 2019. arXiv: 1904.11423. [Online]. Available: http://arxiv.org/abs/1904.11423.

[10] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos Framework: a Methodology and Toolchain for Reproducible Network Experiments", in *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, G. Carle and J. Ott, Eds., ACM, 2021, pp. 259–266. DOI: 10.1145/3485983.3494841. [Online]. Available: https://doi.org/10.1145/3485983.3494841.

[11] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, "Towards a Deeper Understanding of TCP BBR Congestion Control", in *17th International IFIP TC6 Networking Conference, Networking 2018, Zurich, Switzerland, May 14-16, 2018*, C. Casetti, F. A. Kuipers, J. P. G. Sterbenz, and B. Stiller, Eds., IFIP, 2018, pp. 109–117. DOI: 10.23919/IFIPNetworking.2018.8696830. [Online]. Available: http://dl.ifip.org/db/conf/networking/networking2018/2B1-1570416152.pdf.

[12] B. Jaeger, D. Scholz, D. Raumer, F. Geyer, and G. Carle, "Reproducible Measurements of TCP BBR Congestion Control", *Comput. Commun.*, vol. 144, pp. 31–43, 2019. DOI: 10.1016/j.comcom.2019.05.011. [Online]. Available: https://doi.org/10.1016/j.comcom.2019.05.011.

[13] F. Geyer, H. Kinkelin, H. Leppelsack, S. Liebald, D. Scholz, G. Carle, and D. Schupke, "Performance Perspective on Private Distributed Ledger Technologies for Industrial Networks", in *2019 International Conference on Networked Systems, NetSys 2019, Munich, Germany, March 18-21, 2019*, IEEE, 2019, pp. 1–8. DOI: 10.1109/NetSys.2019.8854512. [Online]. Available: https://doi.org/10.1109/NetSys.2019.8854512.

[14] D. Scholz, H. Harkous, S. Gallenmüller, H. Stubbe, M. Helm, B. Jaeger, N. Deric, E. Goshi, Z. Zhou, W. Kellerer, and G. Carle, "A Framework for Reproducible Data Plane Performance Modeling", in *ANCS '21: Symposium on Architectures for Networking and Communications Systems, Layfette, IN, USA, December 13 - 16, 2021*, ACM, 2021, pp. 59–65. DOI: 10.1145/3493425.3502756. [Online]. Available: https://doi.org/10.1145/3493425.3502756.

[15] M. Helm, H. Stubbe, D. Scholz, B. Jaeger, S. Gallenmüller, N. Deric, E. Goshi, H. Harkous, Z. Zhou, W. Kellerer, and G. Carle, "Application of Network Calculus Models on Programmable Device Behavior", in *the 33rd International Teletraffic Congress (ITC 33)*, Avignon, France, Aug. 2021.

[16] D. Scholz, *Repository of P4 Data Plane Performance Modeling Framework*, Last accessed: 2022-07-02, 2021. [Online]. Available: https://github.com/p4-modeling.

[17] M. Simon, H. Stubbe, D. Scholz, S. Gallenmüller, and G. Carle, "High-Performance Match-Action Table Updates from within Programmable Software Data Planes", in *ANCS '21: Symposium on Architectures for Networking and Communications Systems, Layfette, IN, USA, December 13 - 16, 2021*, ACM, 2021, pp. 102–108. DOI: 10.1145/3493425.3502759. [Online]. Available: https://doi.org/10.1145/3493425.3502759.

[18] D. Scholz, F. Geyer, S. Gallenmüller, and G. Carle, "Rapid Prototyping of Avionic Applications Using P4", in *5th P4 Workshop*, Talk, Stanford, CA, USA, 2018. [Online]. Available: http:

//www.net.in.tum.de/fileadmin/bibtex/publications/papers/2018-P4-workshop-avionics-slides.pdf.

[19] D. Scholz, S. Gallenmüller, H. Stubbe, B. Jaber, M. Rouhi, and G. Carle, “Me Love (SYN-) Cookies: SYN Flood Mitigation in Programmable Data Planes”, *CoRR*, vol. abs/2003.03221, 2020. arXiv: 2003.03221. [Online]. Available: https://arxiv.org/abs/2003.03221.

[20] D. Scholz, “TCP SYN Flood Defense in High-Speed Networks”, M.S. thesis, Technical University of Munich, 2016.

[21] D. Scholz, B. Jaber, G. Sebastian, and H. Stubbe, *Repository of SYN Proxy Implementations*, Last accessed: 2021-08-23, 2020. [Online]. Available: https://github.com/syn-proxy.

References

[22] I. Hadzic and J. M. Smith, “P4: A platform for FPGA implementation of protocol boosters”, in *Field-Programmable Logic and Applications, 7th International Workshop, FPL ’97, London, UK, September 1-3, 1997, Proceedings*, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds., ser. Lecture Notes in Computer Science, vol. 1304, Springer, 1997, pp. 438–447. DOI: 10.1007/3-540-63465-7_249. [Online]. Available: https://doi.org/10.1007/3-540-63465-7_249.

[23] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming Protocol-Independent Packet Processors”, *Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. DOI: 10.1145/2656877.2656890. [Online]. Available: https://doi.org/10.1145/2656877.2656890.

[24] J. H. Salim, “When NAPI comes to town”, in *Linux 2005 Conf*, 2005.

[25] J. L. García-Dorado, F. Mata, J. Ramos, P. M. S. del Río, V. Moreno, and J. Aracil, “High-Performance Network Traffic Processing Systems Using Commodity Hardware”, in *Data Traffic Monitoring and Analysis - From Measurement, Classification, and Anomaly Detection to Quality of Experience*, ser. Lecture Notes in Computer Science, E. Biersack, C. Callegari, and M. Matijasevic, Eds., vol. 7754, Springer, 2013, pp. 3–27. DOI: 10.1007/978-3-642-36784-7_1. [Online]. Available: https://doi.org/10.1007/978-3-642-36784-7_1.

[26] R. Bolla and R. Bruschi, “Linux Software Router: Data Plane Optimization and Performance Evaluation”, *J. Networks*, vol. 2, no. 3, pp. 6–17, 2007. DOI: 10.4304/jnw.2.3.6-17. [Online]. Available: https://doi.org/10.4304/jnw.2.3.6-17.

[27] W. Wu, M. Crawford, and M. Bowden, “The performance analysis of linux networking - Packet receiving”, *Comput. Commun.*, vol. 30, no. 5, pp. 1044–1057, 2007. DOI: 10.1016/j.comcom.2006.11.001. [Online]. Available: https://doi.org/10.1016/j.comcom.2006.11.001.

[28] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, “Comparison of Frameworks for High-Performance Packet IO”, in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7-8, 2015*, G. J. Brebner, A. Bachmutsky, and C. R. Das, Eds., IEEE Computer Society, 2015, pp. 29–38. DOI: 10.1109/ANCS.2015.7110118. [Online]. Available: https://doi.org/10.1109/ANCS.2015.7110118.

[29] T. Meyer, F. Wohlfart, D. Raumer, B. E. Wolfinger, and G. Carle, “Measurement and Simulation of High-Performance Packet Processing in Software Routers”, in *Proceedings of Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen, 5. GI/ITG-Workshop MMBnet 2013*, Hamburg, Germany, Sep. 2013.

[30] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, “Assessing Soft- and Hardware Bottlenecks in PC-based Packet Forwarding Systems”, in *Fourteenth International Conference on Networks (ICN 2015)*, Barcelona, Spain, Apr. 2015.

[31] R. Bolla and R. Bruschi, “Pc-based software routers: high performance and application service support”, in *Proceedings of the ACM SIGCOMM 2008 Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO 2008, Seattle, WA, USA, August 22, 2008*, J. Rexford, J. E. van der Merwe, and T. V. Lakshman, Eds., ACM, 2008, pp. 27–32. DOI: `10.1145/1397718.1397725`. [Online]. Available: `https://doi.org/10.1145/1397718.1397725`.

[32] *Intel DPDK: Data Plane Development Kit*, Last accessed: 2021-08-23, Intel Corporation. [Online]. Available: `http://dpdk.org/`.

[33] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, “MoonGen: A Scriptable High-Speed Packet Generator”, in *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*, K. Cho, K. Fukuda, V. S. Pai, and N. Spring, Eds., ACM, 2015, pp. 275–287. DOI: `10.1145/2815675.2815692`. [Online]. Available: `https://doi.org/10.1145/2815675.2815692`.

[34] P. Emmerich, S. Gallenmüller, R. Schönberger, D. Raumer, and G. Carle, “Architectures for Fast and Flexible Software Routers”, Technical University of Munich, Garching near Munich, Germany, Tech. Rep., 2015. [Online]. Available: `https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/MoonRoute_draft1.pdf`.

[35] T. F. D. Project, *The Vector Packet Processor (VPP)*, Last accessed: 2021-08-23, The Fast Data Project, 2019. [Online]. Available: `https://fd.io/vppproject/vpptech/`.

[36] Gorrie, Luke, *SIMD FTW, or, Straightline redux*, Last accessed: 2021-08-23, Snabb, Mar. 2015. [Online]. Available: `https://groups.google.com/g/snabb-devel/c/aez4pEnd4ow/m/WrXi5N7nxfkJ`.

[37] L. Rizzo, “netmap: A Novel Framework for Fast Packet I/O”, in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds., USENIX Association, 2012, pp. 101–112. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rizzo`.

[38] E. Kohler, R. T. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router”, *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000. DOI: `10.1145/354871.354874`. [Online]. Available: `https://doi.org/10.1145/354871.354874`.

[39] K. R. Fall, G. Iannaccone, M. Manesh, S. Ratnasamy, K. J. Argyraki, M. Dobrescu, and N. Egi, “RouteBricks: enabling general purpose network infrastructure”, *ACM SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 112–125, 2011. DOI: `10.1145/1945023.1945037`. [Online]. Available: `https://doi.org/10.1145/1945023.1945037`.

[40] L. Rizzo and G. Lettieri, “VALE, a switched ethernet for virtual machines”, in *Conference on emerging Networking Experiments and Technologies, CoNEXT ’12, Nice, France - December 10 - 13, 2012*, C. Barakat, R. Teixeira, K. K. Ramakrishnan, and P. Thiran, Eds., ACM, 2012, pp. 61–72. DOI: `10.1145/2413176.2413185`. [Online]. Available: `https://doi.org/10.1145/2413176.2413185`.

[41] Ntop, *PF_RING*, Last accessed: 2021-08-23. [Online]. Available: `http://www.ntop.org/products/packet-capture/pf_ring/`.

[42] T. Zhang, L. Linguaglossa, P. Giaccone, L. Iannone, and J. Roberts, “Performance Benchmarking of State-of-the-Art Software Switches for NFV”, *CoRR*, vol. abs/2003.13489, 2020. arXiv: `2003.13489`. [Online]. Available: `https://arxiv.org/abs/2003.13489`.

[43] N. Bonelli, S. Giordano, and G. Procissi, "Network Traffic Processing With PFQ", *IEEE J. Sel. Areas Commun.*, vol. 34, no. 6, pp. 1819–1833, 2016. DOI: 10.1109/JSAC.2016.2558998. [Online]. Available: https://doi.org/10.1109/JSAC.2016.2558998.

[44] M. Attig and G. J. Brebner, "400 Gb/s Programmable Packet Parsing on a Single FPGA", in *2011 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Brooklyn, NY, USA, October 3-4, 2011*, IEEE Computer Society, 2011, pp. 12–23. DOI: 10.1109/ANCS.2011.12. [Online]. Available: https://doi.org/10.1109/ANCS.2011.12.

[45] "NFP-4000 Theory of Operation", Netronome Systems Inc., Tech. Rep., Jan. 2016, Last accessed: 2021-08-23. [Online]. Available: https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf.

[46] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers", in *Symposium on Architecture for Networking and Communications Systems, ANCS '13, San Jose, CA, USA, October 21-22, 2013*, IEEE Computer Society, 2013, pp. 13–24. DOI: 10.1109/ANCS.2013.6665172. [Online]. Available: https://doi.org/10.1109/ANCS.2013.6665172.

[47] N. McKeown, T. E. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner, "OpenFlow: enabling innovation in campus networks", *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008. DOI: 10.1145/1355734.1355746. [Online]. Available: https://doi.org/10.1145/1355734.1355746.

[48] G. J. Brebner, "Packets everywhere: The great opportunity for field programmable technology", in *Proceedings of the 2009 International Conference on Field-Programmable Technology, FPT 2009, Sydney, Australia, December 9-11, 2009*, N. W. Bergmann, O. Diessel, and L. Shannon, Eds., IEEE Computer Society, 2009, pp. 1–10. DOI: 10.1109/FPT.2009.5377604. [Online]. Available: https://doi.org/10.1109/FPT.2009.5377604.

[49] R. Duncan and P. Jungck, "packetC Language for High Performance Packet Processing", in *11th IEEE International Conference on High Performance Computing and Communications, HPCC 2009, 25-27 June 2009, Seoul, Korea*, IEEE, 2009, pp. 450–457. DOI: 10.1109/HPCC.2009.89. [Online]. Available: https://doi.org/10.1109/HPCC.2009.89.

[50] M. K. Chen, X. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju, "Shangri-La: achieving high performance from compiled network applications while enabling ease of programming", in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds., ACM, 2005, pp. 224–236. DOI: 10.1145/1065010.1065038. [Online]. Available: https://doi.org/10.1145/1065010.1065038.

[51] NPL, "NPL: Open, High-Level language for developing feature-rich solutions for programmable networking platforms", 2019, Last accessed: 2021-03-31. [Online]. Available: https://nplang.org/.

[52] H. Song, "Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane", in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*, N. Foster and R. Sherwood, Eds., ACM, 2013, pp. 127–132. DOI: 10.1145/2491185.2491190. [Online]. Available: https://doi.org/10.1145/2491185.2491190.

[53] S. Li, D. Hu, W. Fang, S. Ma, C. Chen, H. Huang, and Z. Zhu, "Protocol Oblivious Forwarding (POF): Software-Defined Networking with Enhanced Programmability", *IEEE Network*, vol. 31, no. 2, pp. 58–66, 2017. DOI: 10.1109/MNET.2017.1600030NM. [Online]. Available: https://doi.org/10.1109/MNET.2017.1600030NM.

[54] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", in *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego,*

California, USA, January 1993, USENIX Association, 1993, pp. 259–270. [Online]. Available: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[55] J. Corbet, *A JIT for packet filters*, Last accessed: 2021-08-23, LWN.net, 2011. [Online]. Available: https://lwn.net/Articles/437981/.

[56] J. Corbet, "BPF: the universal in-kernel virtual machine", Last accessed: 2021-08-23, LWN.net, 2014. [Online]. Available: https://lwn.net/Articles/599755.

[57] IO Visor Project, *bcc Reference Guide*, Last accessed: 2021-08-23, IO Visor Project, 2017. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md.

[58] M. Rybczyńska, *Bounded loops in BPF for the 5.3 kernel*, Last accessed: 2021-08-23, LWN.net, 2019. [Online]. Available: https://lwn.net/Articles/794934/.

[59] Cilium, *BPF and XDP Reference Guide*, Last accessed: 2021-08-23, Cilium, 2020. [Online]. Available: https://docs.cilium.io/en/v1.9/bpf/.

[60] J. Corbet, *Unprivileged bpf()*, Last accessed: 2021-08-23, LWN.net, 2015. [Online]. Available: https://lwn.net/Articles/660331/.

[61] "eBPF Offload Getting Started Guide - Netronome CX SmartNIC", Netronome Systems Inc., Tech. Rep., 2018, Last accessed: 2021-08-23. [Online]. Available: https://www.netronome.com/documents/305/eBPF-Getting_Started_Guide.pdf.

[62] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hXDP: Efficient Software Packet Processing on FPGA NICs", in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, USENIX Association, 2020, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/brunella.

[63] C. Cascaval and D. Daly, "P4 Architectures", 2017, Last accessed: 2021-03-31. [Online]. Available: https://p4.org/assets/p4-ws-2017-p4-architectures.pdf.

[64] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster, "Composing Dataplane Programs with μP4", in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 329–343. DOI: 10.1145/3387514.3405872. [Online]. Available: https://doi.org/10.1145/3387514.3405872.

[65] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu, "Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs", in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 435–450. DOI: 10.1145/3387514.3405879. [Online]. Available: https://doi.org/10.1145/3387514.3405879.

[66] P4lang, *P4lang Behavioral Model (bmv2)*, Last accessed: 2021-08-23, P4lang, 2021. [Online]. Available: https://github.com/p4lang/behavioral-model.

[67] IO Visor Project, *eXpress Data Path*, Last accessed: 2021-08-23, IO Visor Project, 2016. [Online]. Available: https://www.iovisor.org/technology/xdp.

[68] VMware, *p4c-xdp*, Last accessed: 2021-08-23, VMware, 2021. [Online]. Available: https://github.com/vmware/p4c-xdp.

[69] P. Vörös, D. Horpácsi, R. Kitlei, D. Leskó, M. Tejfel, and S. Laki, "T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors", in *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20,*

2018, IEEE, 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850752. [Online]. Available: https://doi.org/10.1109/HPSR.2018.8850752.

[70] S. Laki, D. Horpácsi, P. Vörös, M. Tejfel, P. Hudoba, G. Pongrácz, and L. Molnár, "The Price for Asynchronous Execution of Extern Functions in Programmable Software Data Planes", in *23rd Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN 2020, Paris, France, February 24-27, 2020*, IEEE, 2020, pp. 23–28. DOI: 10.1109/ICIN48450.2020.9059357. [Online]. Available: https://doi.org/10.1109/ICIN48450.2020.9059357.

[71] H. Wang, R. Soulé, H. T. Dang, K. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A Rapid Prototyping Framework for P4", in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, ACM, 2017, pp. 122–135. DOI: 10.1145/3050220.3050234. [Online]. Available: https://doi.org/10.1145/3050220.3050234.

[72] S. Ibanez, G. J. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA Workflow for Line-Rate Packet Processing", in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, K. Bazargan and S. Neuendorffer, Eds., ACM, 2019, pp. 1–9. DOI: 10.1145/3289602.3293924. [Online]. Available: https://doi.org/10.1145/3289602.3293924.

[73] S. Ibanez, G. Antichi, G. J. Brebner, and N. McKeown, "Event-Driven Packet Processing", in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*, ACM, 2019, pp. 133–140. DOI: 10.1145/3365609.3365848. [Online]. Available: https://doi.org/10.1145/3365609.3365848.

[74] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research", *CoRR*, vol. abs/2101.10632, 2021. arXiv: 2101.10632. [Online]. Available: https://arxiv.org/abs/2101.10632.

[75] B. Mihai. "Compiling P4 to EBPF". Last accessed: 2021-08-23. (2015), [Online]. Available: https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.

[76] Osiński, Tomasz, *p4c-ubpf: a New Back-end for the P4 Compiler*, Last accessed: 2021-08-23, p4.org, Jun. 2020. [Online]. Available: https://p4.org/p4/p4c-ubpf.

[77] T. Osinski, H. Tarasiuk, P. Chaignon, and M. Kossakowski, "P4rt-OVS: Programming Protocol-Independent, Runtime Extensions for Open vSwitch with P4", in *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, IEEE, 2020, pp. 413–421. [Online]. Available: https://ieeexplore.ieee.org/document/9142794.

[78] T. K. Dangeti, V. K. S, and R. Upadrasta, "P4LLVM: An LLVM Based P4 Compiler", in *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, IEEE Computer Society, 2018, pp. 424–429. DOI: 10.1109/ICNP.2018.00059. [Online]. Available: https://doi.org/10.1109/ICNP.2018.00059.

[79] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A Programmable, Protocol-Independent Software Switch", in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, Eds., ACM, 2016, pp. 525–538. DOI: 10.1145/2934872.2934886. [Online]. Available: https://doi.org/10.1145/2934872.2934886.

[80] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, and C. Kim, "PVPP: A Programmable Vector Packet Processor", in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, ACM, 2017, pp. 197–198. DOI: 10.1145/3050220.3060609. [Online]. Available: https://doi.org/10.1145/3050220.3060609.

[81] Y. Yan, T. Shen, A. Beldachi, K. Rajkumar, R. Wang, R. Nejabati, and D. Simeonidou, "P4-enabled smart NIC: Architecture and technology enabling sliceable optical DCS", in *European Conference on Optical Communications (ECOC)*, 2019, pp. 1–3.

[82] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers", in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2016, pp. 148–155.

[83] H. T. Dang, H. Wang, T. Jepsen, G. J. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 Language Benchmark Suite", in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, ACM, 2017, pp. 95–101. DOI: 10.1145/3050220.3050231. [Online]. Available: https://doi.org/10.1145/3050220.3050231.

[84] H. Harkous, M. Jarschel, M. He, R. Pries, and W. Kellerer, "Towards Understanding the Performance of P4 Programmable Hardware", in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*, IEEE, 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901881. [Online]. Available: https://doi.org/10.1109/ANCS.2019.8901881.

[85] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends", *CoRR*, 2021. arXiv: 2102.00643. [Online]. Available: https://arxiv.org/abs/2102.00643.

[86] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes", in *ACM SIGCOMM*, 2015.

[87] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "OmniMon: Re-architecting Network Telemetry with Resource Efficiency and Full Accuracy", in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 404–421. DOI: 10.1145/3387514.3405877. [Online]. Available: https://doi.org/10.1145/3387514.3405877.

[88] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic In-band Network Telemetry", in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 662–680. DOI: 10.1145/3387514.3405894. [Online]. Available: https://doi.org/10.1145/3387514.3405894.

[89] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu, "Flow Event Telemetry on Programmable Data Plane", in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 76–89. DOI: 10.1145/3387514.3406214. [Online]. Available: https://doi.org/10.1145/3387514.3406214.

[90] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane", in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, ACM, 2017, pp. 164–176. DOI: 10.1145/3050220.3063772. [Online]. Available: https://doi.org/10.1145/3050220.3063772.

[91] J. Kucera, D. A. Popescu, G. Antichi, J. Korenek, and A. W. Moore, "Seek and Push: Detecting Large Traffic Aggregates in the Dataplane", *CoRR*, vol. abs/1805.05993, 2018. arXiv: 1805.05993. [Online]. Available: http://arxiv.org/abs/1805.05993.

[92] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches", in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/poseidon-mitigating-volumetric-ddos-attacks-with-programmable-switches/.

[93] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin, "NetLock: Fast, Centralized Lock Management Using Programmable Switches", in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 126–138. DOI: 10.1145/3387514.3405857. [Online]. Available: https://doi.org/10.1145/3387514.3405857.

[94] R. Das and A. C. Snoeren, "Enabling Active Networking on RMT Hardware", in *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020*, B. Zhao, H. Zheng, H. V. Madhyastha, and V. N. Padmanabhan, Eds., ACM, 2020, pp. 175–181. DOI: 10.1145/3422604.3425934. [Online]. Available: https://doi.org/10.1145/3422604.3425934.

[95] D. Gonçalves, S. Signorello, F. M. V. Ramos, and M. Médard, "Random Linear Network Coding on Programmable Switches", in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*, IEEE, 2019, pp. 1–6. DOI: 10.1109/ANCS.2019.8901883. [Online]. Available: https://doi.org/10.1109/ANCS.2019.8901883.

[96] R. Kundel, L. Nobach, J. Blendin, H. Kolbe, G. Schyguda, V. Gurevich, B. Koldehofe, and R. Steinmetz, "P4-BNG: Central Office Network Functions on Programmable Packet Pipelines", in *15th International Conference on Network and Service Management, CNSM 2019, Halifax, NS, Canada, October 21-25, 2019*, H. Lutfiyya, Y. Diao, A. N. Zincir-Heywood, R. Badonnel, and E. R. M. Madeira, Eds., IEEE, 2019, pp. 1–9. DOI: 10.23919/CNSM46954.2019.9012666. [Online]. Available: https://doi.org/10.23919/CNSM46954.2019.9012666.

[97] A. Aghdai, M. Huang, D. Dai, Y. Xu, and H. J. Chao, "Transparent Edge Gateway for Mobile Networks", in *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, IEEE Computer Society, 2018, pp. 412–417. DOI: 10.1109/ICNP.2018.00057. [Online]. Available: https://doi.org/10.1109/ICNP.2018.00057.

[98] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári, "The Price for Programmability in the Software Data Plane: The Vendor Perspective", *IEEE J. Sel. Areas Commun.*, vol. 36, no. 12, pp. 2621–2630, 2018. DOI: 10.1109/JSAC.2018.2871307. [Online]. Available: https://doi.org/10.1109/JSAC.2018.2871307.

[99] Shahzad and E. Jung, "Flexible IoT Datapath Programming using P4", *CoRR*, vol. abs/2006.15782, 2020. arXiv: 2006.15782. [Online]. Available: https://arxiv.org/abs/2006.15782.

[100] F. Geyer and M. Winkel, "Towards embedded packet processing devices for rapid prototyping of avionic applications", in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018.

[101] P. Wintermeyer, M. Apostolaki, A. Dietmüller, and L. Vanbever, "P2GO: P4 Profile-Guided Optimizations", in *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020*, B. Zhao, H. Zheng, H. V. Madhyastha, and V. N. Padmanabhan, Eds., ACM, 2020, pp. 146–152. DOI: 10.1145/3422604.3425941. [Online]. Available: https://doi.org/10.1145/3422604.3425941.

[102] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, “Switch Code Generation Using Program Synthesis”, in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 44–61. DOI: `10.1145/3387514.3405852`. [Online]. Available: `https://doi.org/10.1145/3387514.3405852`.

[103] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, “bf4: towards bug-free P4 programs”, in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 571–585. DOI: `10.1145/3387514.3405888`. [Online]. Available: `https://doi.org/10.1145/3387514.3405888`.

[104] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford, “Tracking P4 Program Execution in the Data Plane”, in *SOSR '20: Symposium on SDN Research, San Jose, CA, USA, March 3, 2020*, A. Wang, E. Rozner, and H. Zeng, Eds., ACM, 2020, pp. 117–122. DOI: `10.1145/3373360.3380843`. [Online]. Available: `https://doi.org/10.1145/3373360.3380843`.

[105] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, “TEA: Enabling State-Intensive Network Functions on Programmable Switches”, in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 90–106. DOI: `10.1145/3387514.3405855`. [Online]. Available: `https://doi.org/10.1145/3387514.3405855`.

[106] A. Nötzli, J. Khan, A. Fingerhut, C. W. Barrett, and P. Athanas, “p4pktgen: Automated Test Case Generation for P4 Programs”, in *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*, ACM, 2018, 5:1–5:7. DOI: `10.1145/3185467.3185497`.

[107] B. O'Connor, T. Madejski, J. Wanderer, A. Vahdat, Y. Tseng, M. Pudelko, C. Cascone, A. Endurthi, Y. Wang, A. Ghaffarkhah, D. Gopalpur, and T. Everman, “Using P4 on Fixed-Pipeline and Programmable Stratum Switches”, in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*, IEEE, 2019, pp. 1–2. DOI: `10.1109/ANCS.2019.8901885`. [Online]. Available: `https://doi.org/10.1109/ANCS.2019.8901885`.

[108] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. M. Parulkar, “ONOS: towards an open, distributed SDN OS”, in *Proceedings of the third workshop on Hot topics in software defined networking, HotSDN '14, Chicago, Illinois, USA, August 22, 2014*, A. Akella and A. G. Greenberg, Eds., ACM, 2014, pp. 1–6. DOI: `10.1145/2620728.2620744`. [Online]. Available: `https://doi.org/10.1145/2620728.2620744`.

[109] L. Yu, J. Sonchack, and V. Liu, “Mantis: Reactive Programmable Switches”, in *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 296–309. DOI: `10.1145/3387514.3405870`. [Online]. Available: `https://doi.org/10.1145/3387514.3405870`.

[110] K. Zhang, D. Zhuo, and A. Krishnamurthy, “Gallium: Automated Software Middlebox Offloading to Programmable Switches”, in *SIGCOMM '20: Proceedings of the 2020 Annual conference of*

the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020, H. Schulzrinne and V. Misra, Eds., ACM, 2020, pp. 283–295. DOI: `10.1145/3387514.3405869`. [Online]. Available: `https://doi.org/10.1145/3387514.3405869`.

[111] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4DNS: In-Network DNS", in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*, IEEE, 2019, pp. 1–6. DOI: `10.1109/ANCS.2019.8901896`. [Online]. Available: `https://doi.org/10.1109/ANCS.2019.8901896`.

[112] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: consensus at network speed", in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, J. Rexford and A. Vahdat, Eds., ACM, 2015, 5:1–5:7. DOI: `10.1145/2774993.2774999`. [Online]. Available: `https://doi.org/10.1145/2774993.2774999`.

[113] H. T. Dang, J. Hofmann, Y. Liu, M. Radi, D. Vucinic, R. Soulé, and F. Pedone, "Consensus for Non-volatile Main Memory", in *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, IEEE Computer Society, 2018, pp. 406–411. DOI: `10.1109/ICNP.2018.00056`. [Online]. Available: `https://doi.org/10.1109/ICNP.2018.00056`.

[114] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. H. Clausen, "Stateless Load-Aware Load Balancing in P4", in *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*, IEEE Computer Society, 2018, pp. 418–423. DOI: `10.1109/ICNP.2018.00058`. [Online]. Available: `https://doi.org/10.1109/ICNP.2018.00058`.

[115] L. Zeno, D. R. K. Ports, J. Nelson, and M. Silberstein, "SwiShmem: Distributed Shared State Abstractions for Programmable Switches", in *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020*, B. Zhao, H. Zheng, H. V. Madhyastha, and V. N. Padmanabhan, Eds., ACM, 2020, pp. 160–167. DOI: `10.1145/3422604.3425946`. [Online]. Available: `https://doi.org/10.1145/3422604.3425946`.

[116] S. Bradner and J. McQuaid, *Benchmarking Methodology for Network Interconnect Devices*, RFC 2544 (Informational), RFC, Updated by RFCs 6201, 6815, Fremont, CA, USA: RFC Editor, Mar. 1999. DOI: `10.17487/RFC2544`. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc2544.txt`.

[117] ACM, *Artifact Review and Badging Version 1.1*, Last accessed: 2021-08-23, 2020. [Online]. Available: `https://www.acm.org/publications/policies/artifact-review-and-badging-current`.

[118] V. Bajpai, A. Brunström, A. Feldmann, W. Kellerer, A. Pras, H. Schulzrinne, G. Smaragdakis, M. Wählisch, and K. Wehrle, "The Dagstuhl beginners guide to reproducibility for experimental networking research", *Comput. Commun. Rev.*, vol. 49, no. 1, pp. 24–30, 2019. DOI: `10.1145/3314212.3314217`. [Online]. Available: `https://doi.org/10.1145/3314212.3314217`.

[119] L. Nussbaum, "Testbeds Support for Reproducible Research", in *Proceedings of the Reproducibility Workshop, Reproducibility@SIGCOMM 2017, Los Angeles, CA, USA, August 25, 2017*, ACM, 2017, pp. 24–26. DOI: `10.1145/3097766.3097773`. [Online]. Available: `https://doi.org/10.1145/3097766.3097773`.

[120] R. Bolze, F. Cappello, E. Caron, M. J. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E. Talbi, and I. Touche, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed", *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 4, 2006. DOI: `10.1177/1094342006070078`.

[121] *Fed4Fire*, Last accessed: 2021-08-23. [Online]. Available: https://www.fed4fire.eu.

[122] L. Nussbaum, "An overview of Fed4FIRE testbeds–and beyond?", in *GEFI-Global Experimentation for Future Internet Workshop*, 2019.

[123] *OneLab*, Last accessed: 2021-08-23. [Online]. Available: https://onelab.eu/.

[124] L. Baron, F. Boubekeur, R. Klacza, M. Y. Rahman, C. Scognamiglio, N. Kurose, T. Friedman, and S. Fdida, "Demo: OneLab: Major Computer Networking Testbeds for IoT and Wireless Experimentation", in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom 2015, Paris, France, September 7-11, 2015*, S. Fdida, G. Pau, S. K. Kasera, and H. Zheng, Eds., ACM, 2015, pp. 199–200. DOI: 10.1145/2789168.2789180. [Online]. Available: https://doi.org/10.1145/2789168.2789180.

[125] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbah, A. Rocha, and J. Stubbs, "Lessons Learned from the Chameleon Testbed", in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds., USENIX Association, 2020.

[126] *Geni*, Last accessed: 2021-08-23. [Online]. Available: https://www.geni.net/.

[127] B. N. Chun, D. E. Culler, T. Roscoe, A. C. Bavier, L. L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: an overlay testbed for broad-coverage services", 3, vol. 33, 2003. DOI: 10.1145/956993.956995.

[128] Y. Zhuang, C. Matthews, S. Tredger, S. Ness, J. Short-Gershman, L. Ji, N. Rebenich, A. French, J. Erickson, K. Clarkson, Y. Coady, and R. McGeer, "Taking a walk on the wild side: teaching cloud computing on distributed research testbeds", in *The 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Atlanta, GA, USA - March 05 - 08, 2014*, J. D. Dougherty, K. Nagel, A. Decker, and K. Eiselt, Eds., ACM, 2014. DOI: 10.1145/2538862.2538931.

[129] T. Rakotoarivelo, G. Jourjon, and M. Ott, "Designing and orchestrating reproducible experiments on federated networking testbeds", *Comput. Networks*, vol. 63, pp. 173–187, 2014. DOI: 10.1016/j.bjp.2013.12.033. [Online]. Available: https://doi.org/10.1016/j.bjp.2013.12.033.

[130] O. S. Sella, A. W. Moore, and N. Zilberman, "FEC Killed The Cut-Through Switch", in *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, NEAT@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*, 2018, pp. 15–20. DOI: 10.1145/3229574.3229577. [Online]. Available: https://doi.org/10.1145/3229574.3229577.

[131] Molex, *Molex PXC systems*, Last accessed: 2021-08-23. [Online]. Available: http://www.oplink.com/uploads/files/520cea9a8ecbcceb5d156a1be86b02a1.pdf.

[132] M. Pahl, "The iLab Concept: Making Teaching Better, at Scale", *IEEE Commun. Mag.*, vol. 55, no. 11, pp. 178–185, 2017. DOI: 10.1109/MCOM.2017.1700394. [Online]. Available: https://doi.org/10.1109/MCOM.2017.1700394.

[133] E. de Britto e Silva, N. Slamnik-Krijestorac, S. A. Hadiwardoyo, and J. M. Márquez-Barja, "Bringing 4G LTE closer to students: A low-cost testbed for practical teaching and experimentation", in *GoodTechs '20: 6th EAI International Conference on Smart Objects and Technologies for Social Good, Antwerp, Belgium, September 14-16, 2020*, C. Prandi and J. Márquez-Barja, Eds., ACM, 2020.

[134] P. Celeda, J. Vykopal, V. Svábenský, and K. Slávicek, "KYPO4INDUSTRY: A Testbed for Teaching Cybersecurity of Industrial Control Systems", in *SIGCSE '20: The 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA, March 11-14, 2020*, J. Zhang, M. Sherriff, S. Heckman, P. A. Cutter, and A. E. Monge, Eds., ACM, 2020. DOI: 10.1145/3328778.3366908.

[135] S. S. Hanna, A. Guirguis, M. A. Mahdi, Y. A. El-Nakieb, M. A. Eldin, and D. M. Saber, "CRC: collaborative research and teaching testbed for wireless communications and networks", in *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization, WiNTECH@MobiCom 2016, New York City, New York, USA, October 3-7, 2016*, D. Koutsonikolas and P. Patras, Eds., ACM, 2016. DOI: `10.1145/2980159.2980161`.

[136] L. Yan and N. McKeown, "Learning Networking by Reproducing Research Results", 2, vol. 47, 2017. DOI: `10.1145/3089262.3089266`.

[137] J. Schönwälder, T. Friedman, and A. Pras, "Using networks to teach about networks (report on dagstuhl seminar nr. 17112)", *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, 2017.

[138] V. Bajpai, M. Kühlewind, J. Ott, J. Schönwälder, A. Sperotto, and B. Trammell, "Challenges with Reproducibility", in *Proceedings of the Reproducibility Workshop, Reproducibility@SIGCOMM 2017, Los Angeles, CA, USA, August 25, 2017*, ACM, 2017. DOI: `10.1145/3097766.3097767`.

[139] M. von Maltitz and G. Carle, "A Performance and Resource Consumption Assessment of Secret Sharing Based Secure Multiparty Computation", in *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings*, J. García-Alfaro, J. Herrera-Joancomartí, G. Livraga, and R. Rios, Eds., ser. Lecture Notes in Computer Science, vol. 11025, Springer, 2018, pp. 357–372. DOI: `10.1007/978-3-030-00305-0_25`. [Online]. Available: `https://doi.org/10.1007/978-3-030-00305-0_25`.

[140] D. J. Law, A. Healey, P. Anslow, S. B. Carlson, and V. Maguire, *IEEE 802.3bm-2015*, Last accessed: 2021-08-23, 2015.

[141] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems", in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, R. Mahajan and I. Stoica, Eds., USENIX Association, 2014, pp. 489–502. [Online]. Available: `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong`.

[142] K. Wiles, *The Pktgen Application*, Last accessed: 2021-08-23. [Online]. Available: `http://pktgen.readthedocs.org/en/latest/index.html`.

[143] N. Bonelli, *PFQ i/o*, Last accessed: 2021-08-23. [Online]. Available: `http://www.pfq.io/`.

[144] L. Rizzo, *The netmap project*, Last accessed: 2021-08-23. [Online]. Available: `http://info.iet.unipi.it/~luigi/netmap/`.

[145] P. Srivats, *Ostinato – Packet Traffic Generator and Analyzer*, Last accessed: 2021-08-23. [Online]. Available: `http://ostinato.org/`.

[146] A. Turner, *Tcpreplay*, Last accessed: 2021-08-23. [Online]. Available: `https://github.com/synfinatic/tcpreplay`.

[147] L. Gorrie *et al.*, *Snabb: Simple and fast packet networking*, Last accessed: 2021-08-23. [Online]. Available: `https://github.com/snabbco/snabb`.

[148] G. Luke, *Packet copies: Expensive or cheap?*, Last accessed: 2021-08-23. [Online]. Available: `https://github.com/snabbco/snabb/issues/648`.

[149] P. Emmerich. "libmoon git repository". Last accessed: 2021-08-23, Technische Universität München. (2017), [Online]. Available: `https://github.com/libmoon/libmoon`.

[150] DPDK Project, *DPDK Homepage*, Last accessed: 2021-08-23. [Online]. Available: `https://www.dpdk.org/`.

[151] M. Pall, *The LuaJIT Project*, Last accessed: 2021-08-23. [Online]. Available: http://luajit.org/.

[152] S. Gallenmüller, "Data-Driven Analysis and Modeling of Packet Processing Systems", Ph.D. dissertation, Technical University of Munich, Feb. 2021. DOI: 10.2313/NET-2021-02-1.

[153] T. S. Chis, "Performance Modelling with Adaptive Hidden Markov Models and Discriminatory Processor Sharing Queues", Ph.D. dissertation, Imperial College London, UK, 2016. [Online]. Available: http://hdl.handle.net/10044/1/39049.

[154] J. Liu, W. T. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Cascaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes", in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, S. Gorinsky and J. Tapolcai, Eds., ACM, 2018, pp. 490–503. DOI: 10.1145/3230543.3230582.

[155] M. C. Neves, L. Freire, A. E. S. Filho, and M. P. Barcellos, "Verification of P4 programs in feasible time using assertions", in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, X. A. Dimitropoulos, A. Dainotti, L. Vanbever, and T. Benson, Eds., ACM, 2018, pp. 73–85. DOI: 10.1145/3281411.3281421.

[156] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic Test Packet Generation", *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, 2014. DOI: 10.1109/TNET.2013.2253121. [Online]. Available: https://doi.org/10.1109/TNET.2013.2253121.

[157] H. Harkous, M. Jarschel, M. He, R. Pries, and W. Kellerer, "P8: P4 with Predictable Packet Processing Performance", *IEEE Transactions on Network and Service Management*, 2020.

[158] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading Latency for Compute in the Network", in *Proceedings of the 2020 Workshop on Network Application Integration/CoDesign, NAI@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, ACM, 2020, pp. 35–40. DOI: 10.1145/3405672.3405807. [Online]. Available: https://doi.org/10.1145/3405672.3405807.

[159] Y. Zhou, J. Bi, C. Zhang, M. Xu, and J. Wu, "FlexMesh: Flexibly Chaining Network Functions on Programmable Data Planes at Runtime", in *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, IEEE, 2020, pp. 73–81. [Online]. Available: http://ieeexplore.ieee.org/document/118453912.

[160] X. Chen, D. Zhang, and H. Zhou, "MATReduce: Towards High-Performance P4 Pipeline by Reducing Duplicate Match Operations", in *IEEE Global Communications Conference, GLOBECOM 2018, Abu Dhabi, United Arab Emirates, December 9-13, 2018*, IEEE, 2018, pp. 1–7. DOI: 10.1109/GLOCOM.2018.8647320. [Online]. Available: https://doi.org/10.1109/GLOCOM.2018.8647320.

[161] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation", in *Passive and Active Measurement - 13th International Conference, PAM 2012, Vienna, Austria, March 12-14th, 2012. Proceedings*, N. Taft and F. Ricciato, Eds., ser. Lecture Notes in Computer Science, vol. 7192, Springer, 2012, pp. 85–95. DOI: 10.1007/978-3-642-28537-0_9. [Online]. Available: https://doi.org/10.1007/978-3-642-28537-0_9.

[162] D. Lukács, G. Pongrácz, and M. Tejfel, "Performance guarantees for P4 through cost analysis", in *2019 IEEE 15th International Scientific Conference on Informatics*, IEEE, 2019, pp. 000 305–000 310.

[163] ——, "Control flow based cost analysis for P4", *Open Comput. Sci.*, vol. 11, no. 1, pp. 70–79, 2021. DOI: 10.1515/comp-2020-0131. [Online]. Available: https://doi.org/10.1515/comp-2020-0131.

[164] J. J. Moré, “The Levenberg-Marquardt algorithm: implementation and theory”, in *Numerical analysis*, Springer, 1978, pp. 105–116.

[165] Y. Rubner, C. Tomasi, and L. J. Guibas, “A Metric for Distributions with Applications to Image Databases”, in *Proceedings of the Sixth International Conference on Computer Vision (ICCV-98), Bombay, India, January 4-7, 1998*, IEEE Computer Society, 1998, pp. 59–66. DOI: `10.1109/ICCV.1998.710701`. [Online]. Available: `https://doi.org/10.1109/ICCV.1998.710701`.

[166] K. Ahnert and M. Abel, “Numerical differentiation of experimental data: local versus global methods”, *Comput. Phys. Commun.*, vol. 177, no. 10, pp. 764–774, 2007. DOI: `10.1016/j.cpc.2007.03.009`. [Online]. Available: `https://doi.org/10.1016/j.cpc.2007.03.009`.

[167] W. H. Press and S. A. Teukolsky, “Savitzky-Golay smoothing filters”, *Computers in Physics*, vol. 4, no. 6, pp. 669–672, 1990.

[168] P4ELTE, *T4P4S, a multitarget P4$_{16}$ compiler*, Last accessed: 2021-08-23, 2020. [Online]. Available: `https://github.com/P4ELTE/t4p4s%22`.

[169] S. Gallenmüller, J. Naab, I. Adam, and G. Carle, “5G QoS: Impact of Security Functions on Latency”, in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*, IEEE, 2020, pp. 1–9. DOI: `10.1109/NOMS47738.2020.9110422`. [Online]. Available: `https://doi.org/10.1109/NOMS47738.2020.9110422`.

[170] Huston, Geoff, *BGP in 2020 – The BGP Table*, Last accessed: 2021-08-23, APNIC, 2021. [Online]. Available: `https://blog.apnic.net/2021/01/05/bgp-in-2020-the-bgp-table/`.

[171] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 248966-042b, Last accessed: 2021-08-23, Intel, 2021. [Online]. Available: `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html`.

[172] P. Gupta, S. Lin, and N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds”, in *Proceedings IEEE INFOCOM ’98, The Conference on Computer Communications, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies, Gateway to the 21st Century, San Francisco, CA, USA, March 29 - April 2, 1998*, IEEE Computer Society, 1998, pp. 1240–1247. DOI: `10.1109/INFCOM.1998.662938`. [Online]. Available: `https://doi.org/10.1109/INFCOM.1998.662938`.

[173] DPDK Project, *`Rte_lpm.h` file reference*, Last accessed: 2021-08-23, 2019. [Online]. Available: `https://doc.dpdk.org/api-19.02/rte__lpm_8h_source.html`.

[174] A. Fog, “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”, *Copenhagen University College of Engineering*, p. 134, 2012.

[175] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, “Sonata: query-driven streaming network telemetry”, in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, S. Gorinsky and J. Tapolcai, Eds., ACM, 2018, pp. 357–371. DOI: `10.1145/3230543.3230555`. [Online]. Available: `https://doi.org/10.1145/3230543.3230555`.

[176] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, “NetCache: BalancinKey-Value Stores with Fast In-Network Caching”, in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, ACM, 2017, pp. 121–136. DOI: `10.1145/3132747.3132764`. [Online]. Available: `https://doi.org/10.1145/3132747.3132764`.

[177] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon”, in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, M. P. Barcellos, J. Crowcroft, A.

Vahdat, and S. Katti, Eds., ACM, 2016, pp. 101–114. DOI: 10.1145/2934872.2934906. [Online]. Available: https://doi.org/10.1145/2934872.2934906.

[178] A. Z. Broder and M. Mitzenmacher, "Survey: Network Applications of Bloom Filters: A Survey", *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2003. DOI: 10.1080/15427951.2004.10129096. [Online]. Available: https://doi.org/10.1080/15427951.2004.10129096.

[179] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters", *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004. DOI: 10.1109/MM.2004.1268997. [Online]. Available: https://doi.org/10.1109/MM.2004.1268997.

[180] G. Gui, M. Liu, F. Tang, N. Kato, and F. Adachi, "6G: Opening New Horizons for Integration of Comfort, Security, and Intelligence", *IEEE Wirel. Commun.*, vol. 27, no. 5, pp. 126–132, 2020. DOI: 10.1109/MWC.001.1900516. [Online]. Available: https://doi.org/10.1109/MWC.001.1900516.

[181] B. Preneel, "Cryptographic hash functions", *Eur. Trans. Telecommun.*, vol. 5, no. 4, pp. 431–448, 1994. DOI: 10.1002/ett.4460050406. [Online]. Available: https://doi.org/10.1002/ett.4460050406.

[182] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks", in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, USENIX Association, 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks.

[183] S. Goldberg and J. Rexford, "Security Vulnerabilities and Solutions for Packet Sampling", in *Proceedings of the 2007 IEEE Sarnoff Symposium*, Apr. 2007. DOI: 10.1109/SARNOF.2007.4567339.

[184] P. Hoffman and B. Schneier, *Attacks on Cryptographic Hashes in Internet Protocols*, RFC 4270 (Informational), RFC, Fremont, CA, USA: RFC Editor, Nov. 2005. DOI: 10.17487/RFC4270. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4270.txt.

[185] J. Choi, S. Min, and Y. Han, "MACsec Extension over Software-Defined Networks for in-Vehicle Secure Communication", in *Tenth International Conference on Ubiquitous and Future Networks, ICUFN 2018, Prague, Czech Republic, July 3-6, 2018*, IEEE, 2018, pp. 180–185. DOI: 10.1109/ICUFN.2018.8436963. [Online]. Available: https://doi.org/10.1109/ICUFN.2018.8436963.

[186] B. Carnevale, L. Fanucci, S. Bisase, and H. Hunjan, "MACsec-Based Security for Automotive Ethernet Backbones", *Journal of Circuits, Systems, and Computers*, vol. 27, no. 5, 1850082:1–1850082:17, 2018. DOI: 10.1142/S0218126618500822. [Online]. Available: https://doi.org/10.1142/S0218126618500822.

[187] E. Heidinger, C. Heller, A. Klein, and S. Schneele, "Quality of service IP cabin infrastructure", in *29th Digital Avionics Systems Conference*, IEEE, 2010.

[188] F. Hauser, M. Schmidt, M. Häberle, and M. Menth, "P4-MACsec: Dynamic Topology Monitoring and Data Layer Protection With MACsec in P4-Based SDN", *IEEE Access*, vol. 8, pp. 58 845–58 858, 2020. DOI: 10.1109/ACCESS.2020.2982859. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2982859.

[189] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, "P4-IPsec: Site-to-Site and Host-to-Site VPN With IPsec in P4-Based SDN", *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020. DOI: 10.1109/ACCESS.2020.3012738. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3012738.

[190] W. Eddy, *TCP SYN Flooding Attacks and Common Mitigations*, RFC 4987 (Informational), RFC, Fremont, CA, USA: RFC Editor, Aug. 2007. DOI: 10.17487/RFC4987. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4987.txt.

[191] T. Datta, N. Feamster, J. Rexford, and L. Wang, "SPINE: Surveillance Protection in the Network Elements", in *9th USENIX Workshop on Free and Open Communications on the Internet, FOCI*

2019, Santa Clara, CA, USA, August 13, 2019, S. E. McGregor and M. C. Tschantz, Eds., USENIX Association, 2019. [Online]. Available: `https://www.usenix.org/conference/foci19/presentation/datta`.

[192] U. Ben-Porat, A. Bremler-Barr, and H. Levy, "Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks", *IEEE Trans. Computers*, vol. 62, no. 5, pp. 1031–1043, 2013. DOI: `10.1109/TC.2012.49`. [Online]. Available: `https://doi.org/10.1109/TC.2012.49`.

[193] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data Plane Performance Diagnosis of TCP", in *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, ACM, 2017, pp. 61–74. DOI: `10.1145/3050220.3050228`. [Online]. Available: `https://doi.org/10.1145/3050220.3050228`.

[194] E. Cidon, S. Choi, S. Katti, and N. McKeown, "AppSwitch: Application-layer Load Balancing within a Software Switch", in *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*, K. Chen and J. Padhye, Eds., ACM, 2017, pp. 64–70. DOI: `10.1145/3106989.3106998`. [Online]. Available: `https://doi.org/10.1145/3106989.3106998`.

[195] M. Molina, S. Niccolini, and N. Duffield, "A Comparative Experimental Study of Hash Functions Applied to Packet Sampling", in *International Teletraffic Congress (ITC-19)*, 2005.

[196] C. Henke, C. Schmoll, and T. Zseby, "Empirical Evaluation of Hash Functions for PacketID Generation in Sampled Multipoint Measurements", in *Passive and Active Network Measurement, 10th International Conference, PAM 2009, Seoul, Korea, April 1-3, 2009. Proceedings*, S. B. Moon, R. Teixeira, and S. Uhlig, Eds., ser. Lecture Notes in Computer Science, vol. 5448, Springer, 2009, pp. 197–206. DOI: `10.1007/978-3-642-00975-4_20`. [Online]. Available: `https://doi.org/10.1007/978-3-642-00975-4_20`.

[197] R. Jenkins, *A Hash Function for Hash Table Lookup*, Last accessed: 2021-08-23, 1997. [Online]. Available: `http://www.burtleburtle.net/bob/hash/doobs.html`.

[198] A. Appleby, *MurmurHash Project Page*, Last accessed: 2021-08-23, 2016. [Online]. Available: `https://sites.google.com/site/murmurhash/`.

[199] F. Yamaguchi and H. Nishi, "Hardware-based hash functions for network applications", in *19th IEEE International Conference on Networks, ICON 2013, Singapore, December 11-13, 2013*, IEEE, 2013, pp. 1–6. DOI: `10.1109/ICON.2013.6781990`. [Online]. Available: `https://doi.org/10.1109/ICON.2013.6781990`.

[200] J. Aumasson and D. J. Bernstein, "SipHash: A Fast Short-Input PRF", in *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, S. D. Galbraith and M. Nandi, Eds., ser. Lecture Notes in Computer Science, vol. 7668, Springer, 2012, pp. 489–508. DOI: `10.1007/978-3-642-34931-7_28`. [Online]. Available: `https://doi.org/10.1007/978-3-642-34931-7_28`.

[201] J. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "BLAKE2: Simpler, Smaller, Fast as MD5", in *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, M. J. J. Jr., M. E. Locasto, P. Mohassel, and R. Safavi-Naini, Eds., ser. Lecture Notes in Computer Science, vol. 7954, Springer, 2013, pp. 119–135. DOI: `10.1007/978-3-642-38980-1_8`. [Online]. Available: `https://doi.org/10.1007/978-3-642-38980-1_8`.

[202] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code", in *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, H. Gilbert and H. Handschuh, Eds., ser. Lecture Notes in Computer Science, vol. 3557, Springer, 2005, pp. 32–49. DOI: `10.1007/11502760_3`. [Online]. Available: `https://doi.org/10.1007/11502760_3`.

[203] Y. Nir, *ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec*, RFC 7634 (Proposed Standard), RFC, Fremont, CA, USA: RFC Editor, Aug. 2015. DOI: 10.17487/RFC7634. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7634.txt.

[204] A. Langley, W. Chang, N. Mavrogiannopoulos, J. Strombergson, and S. Josefsson, *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*, RFC 7905 (Proposed Standard), RFC, Fremont, CA, USA: RFC Editor, Jun. 2016. DOI: 10.17487/RFC7905. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7905.txt.

[205] N. Hua, E. Norige, S. Kumar, and B. Lynch, "Non-crypto Hardware Hash Functions for High Performance Networking ASICs", in *2011 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Brooklyn, NY, USA, October 3-4, 2011*, IEEE Computer Society, 2011, pp. 156–166. DOI: 10.1109/ANCS.2011.32. [Online]. Available: https://doi.org/10.1109/ANCS.2011.32.

[206] L. Dadda, M. Macchetti, and J. Owen, "The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512)", in *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, IEEE Computer Society, 2004, pp. 70–75. DOI: 10.1109/DATE.2004.1269207. [Online]. Available: https://doi.org/10.1109/DATE.2004.1269207.

[207] X. Du and S. Li, "The ASIC Implementation of SM3 Hash Algorithm for High Throughput", *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. 99-A, no. 7, pp. 1481–1487, 2016. DOI: 10.1587/transfun.E99.A.1481. [Online]. Available: https://doi.org/10.1587/transfun.E99.A.1481.

[208] H. Cho, "ASIC-Resistance of Multi-Hash Proof-of-Work Mechanisms for Blockchain Consensus Protocols", *IEEE Access*, vol. 6, pp. 66 210–66 222, 2018. DOI: 10.1109/ACCESS.2018.2878895. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2878895.

[209] The P4 Language Consortium, *The P4 Language Specification, Version 1.0.5*, Last accessed: 2021-08-23, Nov. 2018. [Online]. Available: https://p4.org/specs/.

[210] P4.org Architecture Working Group, *$P4_{16}$ Portable Switch Architecture (PSA), Version 1.1*, Last accessed: 2021-08-23, Nov. 2018. [Online]. Available: https://p4.org/specs/.

[211] Z. Cao, Z. Wang, and E. W. Zegura, "Performance of Hashing-Based Schemes for Internet Load Balancing", in *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000*, IEEE Computer Society, 2000, pp. 332–341. DOI: 10.1109/INFCOM.2000.832203. [Online]. Available: https://doi.org/10.1109/INFCOM.2000.832203.

[212] Austin Appleby, *Poly1305 git repository*, Last accessed: 2021-08-23, 2016. [Online]. Available: https://github.com/floodyberry/poly1305-donna.

[213] Pedro Brito, *A VHDL implementation of SipHash*, Last accessed: 2021-08-23, 2015. [Online]. Available: https://github.com/pemb/siphash.

[214] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs", *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 368, 2012. [Online]. Available: http://eprint.iacr.org/2012/368.

[215] X. Chen, "Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables", in *Proceedings of the 2020 ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure, SPIN@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, A. Chen and L. Vanbever, Eds., ACM, 2020, pp. 8–14. DOI: 10.1145/3405669.3405819. [Online]. Available: https://doi.org/10.1145/3405669.3405819.

[216] S. Wray, *The Joy of Micro-C*, Last accessed: 2021-08-23, 2014. [Online]. Available: https://open-nfp.org/media/documents/the-joy-of-micro-c_fcjSfra.pdf.

[217] L. Malina, D. Smekal, S. Ricci, J. Hajny, P. Cíbik, and J. Hrabovsky, “Hardware-Accelerated Cryptography for Software-Defined Networks with P4”, in *Innovative Security Solutions for Information Technology and Communications - 13th International Conference, SecITC 2020, Bucharest, Romania, November 19-20, 2020, Revised Selected Papers*, D. Maimut, A. Oprina, and D. Sauveron, Eds., ser. Lecture Notes in Computer Science, vol. 12596, Springer, 2020, pp. 271–287. DOI: 10.1007/978-3-030-69255-1_18. [Online]. Available: https://doi.org/10.1007/978-3-030-69255-1_18.

[218] C. Osborne, *16 DDoS attacks take place every 60 seconds, rates reach 622 Gbps*, Last accessed: 2021-08-23, Feb. 2020. [Online]. Available: https://www.zdnet.com/article/16-ddos-attacks-take-place-every-60-seconds-rates-reach-622-gbps/.

[219] radware Inc, *DoS Cyber Attack Campaign Against Israeli Targets*, Last accessed: 2021-08-23, 2013. [Online]. Available: https://security.radware.com/ddos-threats-attacks/threat-advisories-attack-reports/dos-cyber-campaign-against-israeli-targets/.

[220] ——, *Operation Ababil*, Last accessed: 2021-08-23, 2013. [Online]. Available: https://security.radware.com/WorkArea/DownloadAsset.aspx?id=848.

[221] O. Kupreev, E. Badovskaya, and A. Gutnikov, *DDoS attacks in Q1 2020*, Last accessed: 2021-08-23, May 2020. [Online]. Available: https://securelist.com/ddos-attacks-in-q1-2020/96837/.

[222] D. J. Bernstein, *SYN cookies*, Last accessed: 2021-08-23, 1996. [Online]. Available: http://cr.yp.to/syncookies.html.

[223] T. Aura, P. Nikander, and J. Leiwo, “DOS-Resistant Authentication with Client Puzzles”, in *Security Protocols, 8th International Workshop, Cambridge, UK, April 3-5, 2000, Revised Papers*, B. Christianson, B. Crispo, and M. Roe, Eds., ser. Lecture Notes in Computer Science, vol. 2133, Springer, 2000, pp. 170–177. DOI: 10.1007/3-540-44810-1_22. [Online]. Available: https://doi.org/10.1007/3-540-44810-1_22.

[224] P. Ferguson and D. Senie, *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*, RFC 2827 (Best Current Practice), RFC, Updated by RFC 3704, Fremont, CA, USA: RFC Editor, May 2000. DOI: 10.17487/RFC2827. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2827.txt.

[225] C. N. Maregeli, “A study on TCP-SYN attacks and their effects on a network infrastructure”, M.S. thesis, TU Delft, Delft University of Technology, 2010.

[226] Wesley M. Eddy, *Defenses Against TCP SYN Flooding Attacks - The Internet Protocol Journal - Volume 9, Number 4*, Last accessed: 2021-08-23. [Online]. Available: https://web.archive.org/web/20190619064221/http://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-34/syn-flooding-attacks.html.

[227] Cisco Security, *A Cisco Guide to Defending Against Distributed Denial of Service Attacks*, Last accessed: 2021-08-23. [Online]. Available: https://tools.cisco.com/security/center/resources/guide_ddos_defense.

[228] E. Brodsky and B. S. Darkhovsky, *Nonparametric methods in change point problems*. Springer Science & Business Media, 2013, vol. 243.

[229] W. Chen and D. Yeung, “Defending Against TCP SYN Flooding Attacks Under Different Types of IP Spoofing”, in *Fifth International Conference on Networking and the International Conference on Systems (ICN / ICONS / MCL 2006), 23-29 April 2006, Mauritius*, IEEE Computer Society, 2006, p. 38. DOI: 10.1109/ICNICONSMCL.2006.72. [Online]. Available: https://doi.org/10.1109/ICNICONSMCL.2006.72.

[230] Prince, Matthew, *A Brief Primer on Anycast*, The Cloudflare Blog, Last accessed: 2021-08-23, 2011. [Online]. Available: `https://blog.cloudflare.com/a-brief-anycast-primer/`.

[231] G. C. M. Moura, R. de Oliveira Schmidt, J. S. Heidemann, W. B. de Vries, M. Müller, L. Wei, and C. Hesselman, "Anycast vs. DDoS: Evaluating the November 2015 Root DNS Event", in *Proceedings of the 2016 ACM on Internet Measurement Conference, IMC 2016, Santa Monica, CA, USA, November 14-16, 2016*, P. Gill, J. S. Heidemann, J. W. Byers, and R. Govindan, Eds., ACM, 2016, pp. 255–270. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2987446`.

[232] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help", *ACM Trans. Internet Techn.*, vol. 6, no. 1, pp. 20–52, 2006. DOI: `10.1145/1125274.1125276`. [Online]. Available: `https://doi.org/10.1145/1125274.1125276`.

[233] M. Aron and P. Druschel, "TCP implementation enhancements for improving Webserver performance", *Technical Report TR99-335*, 1999.

[234] L. Ricciulli, P. Lincoln, and P. Kakkar, "TCP SYN flooding defense", in *CNDS*, CNDS, 1999.

[235] Imperva, Inc., *TCP SYN Flood*, Last accessed: 2021-08-23. [Online]. Available: `https://www.incapsula.com/ddos/attack-glossary/syn-flood.html`.

[236] J. Lemon, "Resisting SYN Flood DoS Attacks with a SYN Cache", in *Proceedings of BSDCon 2002, San Francisco, California, USA, February 11-14, 2002*, S. J. Leffler, Ed., USENIX, 2002, pp. 89–97. [Online]. Available: `http://www.usenix.org/publications/library/proceedings/bsdcon02/lemon.html`.

[237] A. Juels and J. G. Brainard, "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks", in *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999, San Diego, California, USA*, The Internet Society, 1999. [Online]. Available: `https://www.ndss-symposium.org/ndss1999/cryptographic-defense-against-connection-depletion-attacks/`.

[238] D. Karig and R. Lee, "Remote Denial of Service Attacks and Countermeasures", *Princeton University Department of Electrical Engineering Technical Report CE-L2001-002*, vol. 17, 2001.

[239] K. Pentikousis and H. G. Badr, "Quantifying the deployment of TCP options - a comparative study", *IEEE Commun. Lett.*, vol. 8, no. 10, pp. 647–649, 2004. DOI: `10.1109/LCOMM.2004.835308`. [Online]. Available: `https://doi.org/10.1109/LCOMM.2004.835308`.

[240] R. Oncioiu and E. Simion, "Approach to Prevent SYN Flood DoS Attacks in Cloud", in *2018 International Conference on Communications (COMM)*, IEEE, 2018.

[241] X. Wang and M. K. Reiter, "Defending Against Denial-of-Service Attacks with Puzzle Auctions", in *2003 Symposium on Security and Privacy, 2003.*, IEEE, 2003, pp. 78–92.

[242] *tcp(7) - Linux man page*, Last accessed: 2021-08-23. [Online]. Available: `https://linux.die.net/man/7/tcp`.

[243] H. Wang, D. Zhang, and K. G. Shin, "Detecting SYN Flooding Attacks", in *Proceedings IEEE INFOCOM 2002, The 21st Annual Joint Conference of the IEEE Computer and Communications Societies, New York, USA, June 23-27, 2002*, IEEE Computer Society, 2002, pp. 1530–1539. DOI: `10.1109/INFCOM.2002.1019404`. [Online]. Available: `https://doi.org/10.1109/INFCOM.2002.1019404`.

[244] A. Aissani, "Queueing Analysis for Networks Under DoS Attack", in *Computational Science and Its Applications - ICCSA 2008, International Conference, Perugia, Italy, June 30 - July 3, 2008, Proceedings, Part II*, O. Gervasi, B. Murgante, A. Laganà, D. Taniar, Y. Mun, and M. L. Gavrilova, Eds., ser. Lecture Notes in Computer Science, vol. 5073, Springer, 2008, pp. 500–513. DOI: `10.1007/978-3-540-69848-7_41`. [Online]. Available: `https://doi.org/10.1007/978-3-540-69848-7_41`.

[245] Microsoft TechNet, *Syn attack protection on Windows Vista, Windows 2008, Windows 7, Windows 2008 R2, Windows 8/8.1, Windows 2012 and Windows 2012 R2*, Last accessed: 2021-08-23. [Online]. Available: https://blogs.technet.microsoft.com/nettracer/2010/06/01/syn-attack-protection-on-windows-vista-windows-2008-windows-7-windows-2008-r2-windows-88-1-windows-2012-and-windows-2012-r2/.

[246] P. McHardy, *netfilter: implement netfilter SYN proxy*, Last accessed: 2021-08-23, 2013. [Online]. Available: https://lwn.net/Articles/563151/.

[247] Netscout Systems, Inc, *Arbor Threat Mitigation System (TMS)*, Data Sheet, Last accessed: 2021-08-23, 2018. [Online]. Available: https://www.netscout.com/product/arbor-threat-mitigation-system.

[248] Cloudflare, *SYN Flood Attack*, Last accessed: 2021-08-23, 2011. [Online]. Available: https://www.cloudflare.com/en-gb/learning/ddos/syn-flood-ddos-attack/.

[249] H. Kabir, J. L. Santos, and R. Kantola, "Securing the Private Realm Gateway", in *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*, IEEE Computer Society, 2016, pp. 243–251. DOI: 10.1109/IFIPNetworking.2016.7497199. [Online]. Available: https://doi.org/10.1109/IFIPNetworking.2016.7497199.

[250] L. Kavisankar and C. Chellappan, "CNoA: Challenging Number Approach for uncovering TCP SYN flooding using SYN spoofing attack", *CoRR*, vol. abs/1110.1753, 2011. arXiv: 1110.1753. [Online]. Available: http://arxiv.org/abs/1110.1753.

[251] S. Kumarasamy and A. Gowrishankar, "An Active Defense Mechanism for TCP SYN flooding attacks", *CoRR*, vol. abs/1201.2103, 2012. arXiv: 1201.2103. [Online]. Available: http://arxiv.org/abs/1201.2103.

[252] J. Postel, *Transmission Control Protocol*, RFC 793 (Internet Standard), RFC, Updated by RFCs 1122, 3168, 6093, 6528, Fremont, CA, USA: RFC Editor, Sep. 1981. DOI: 10.17487/RFC0793. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.txt.

[253] A. Zuquete, "Improving the functionality of syn cookies", in *Advanced Communications and Multimedia Security, IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security, September 26-27, 2002, Portoroz, Slovenia*, B. Jerman-Blazic and T. Klobucar, Eds., ser. IFIP Conference Proceedings, vol. 228, Kluwer, 2002, pp. 57–77. DOI: 10.1007/978-0-387-35612-9_6. [Online]. Available: https://doi.org/10.1007/978-0-387-35612-9_6.

[254] C. M. Kozierok, *TCP Connection Establishment Process: The Three-Way Handshake*, Last accessed: 2021-08-23, The TCP/IP Guide, Sep. 2005. [Online]. Available: http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentProcessTheThreeWayHandsh-4.htm.

[255] M. Dobrescu, N. Egi, K. J. Argyraki, B. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: exploiting parallelism to scale software routers", in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds., ACM, 2009, pp. 15–28. DOI: 10.1145/1629575.1629578. [Online]. Available: https://doi.org/10.1145/1629575.1629578.

[256] K. Larsson, *DDoStop app: a basic DDoS mitigation application*, Last accessed: 2021-08-23. [Online]. Available: https://github.com/plajjan/snabbswitch/tree/snabbddos/src/apps/ddos.

[257] S. Y. Cheung, *The Second Chance Page Replacement Policy*, Last accessed: 2021-08-23. [Online]. Available: http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/9-virtual-mem/SC-replace.html.

[258] *Google sparsehash*, Last accessed: 2021-08-23, 2010. [Online]. Available: https://github.com/sparsehash/sparsehash.

[259] S. Gallenmüller, P. Emmerich, R. Schönberger, D. Raumer, and G. Carle, “Building Fast but Flexible Software Routers”, in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2017, Beijing, China, May 18-19, 2017*, IEEE Computer Society, 2017, pp. 101–102. DOI: `10.1109/ANCS.2017.21`. [Online]. Available: `https://doi.org/10.1109/ANCS.2017.21`.

[260] G. Tene, *wrk2 - a HTTP benchmarking tool based mostly on wrk*, Last accessed: 2021-08-23, 2012. [Online]. Available: `https://github.com/giltene/wrk2`.