國立清華大學電機資訊學院資訊安全研究所
碩士論文
Institute of Information Security
College of Electrical Engineering and Computer Science
National Tsing Hua University
Master Thesis

CUCKOOGUARD：智慧網路卡上的節約記憶體SYN洪水攻擊防禦架構

CuckooGuard: A Memory-Efficient SYN Flood Defense Architecture for SmartNICs

崔斯坦
Tristan Döring
Student ID: 112164421

指導教授：徐正炘 博士; 孫宏民 博士
Advisors: Cheng-Hsin Hsu, Ph.D.; Hung-Min Sun, Ph.D.

中華民國 114 年 06 月
June, 2025

國立清華大學學位論文授權書

Authorized Agreement for Thesis/Dissertation

(裝訂於紙本論文書名頁之次頁 For author to bind after the title page)

● 立書人（即論文作者）Author: 崔斯坦 （以下稱本人 hereinafter referred to as " I "）

● 授權標的 Authorized subject：本人於 國立清華大學 （以下稱學校）電機資訊學院 資訊安全研究所（研究所、學位學程）113學年度 第2學期之 ■碩士 □博士 學位論文
I hereby authorize that my ■ Master’s □ Doctoral thesis submitted in the 2 semester of the 113 academic year at National Tsing Hua University (hereinafter referred to as "the University"), College of Electrical Engineering and Computer Science College, Institute of Information Security Department / Graduate Institute / Degree Program,
論文題目 Title：CUCKOOGUARD：智慧網路卡上的節約記憶體SYN洪水攻擊防禦架構
指導教授 Advisor：孫宏民,徐正炘

（以下稱**本著作**，本著作並包含論文全部、摘要、目錄、圖檔、影音以及相關書面報告、技術報告或專業實務報告等，以下同
Hereinafter referred to as "the publication", which contains all thesis/dissertation, abstracts, catalogues, graphic documents, audiovisual reports, technical reports or professional practice reports, etc.）

緣依據**學位授予法**等相關法令，對於本著作及其電子檔，學校圖書館得依法進行保存等利用，而國家圖書館則得依法進行保存、以紙本或讀取設備於館內提供公眾閱覽等利用。此外，為促進學術研究及傳播，本人在此並進一步同意授權(1)**國立清華大學**與(2)**國家圖書館對**本著作進行以下各點所定之利用：

In accordance with the Degree Conferral Act and other relevant laws and regulations, for this publication and its electronic file, the school library can be preserved and used according to the law. The National Central Library must preserve it in accordance with the law and permit public access in the library with paper or reading equipment. In addition, in order to promote academic research and scholarly communication, I hereby agree to authorize (1)**National Tsing Hua University** and the (2)**National Central Library** to use this publication for the following purposes

一、授權部分：

本人**同意**授權國立清華大學與國家圖書館，無償、不限期間與次數重製本著作並得為教育、科學及研究等**非營利用途之利用**，其包括得將本著作之電子檔收錄於數位資料庫，並透過自有或委託代管之伺服器、網路系統或網際網路向位於全球之使用者(包含但不限於國立清華大學校園內、校園外及國家圖書館館內、館外)公開傳輸，以供使用者為非營利目的之檢索、閱覽、下載及/或列印。
I hereby agree to authorize National Tsing Hua University and the National Central Library to reproduce this publication free of charge, without

limitation on time or the number of reproductions, and for education, science, research, and other non-profit use. This involves recording the electronic file of this publication into a digital database and the public transmission of them via self-owned or entrusted servers, network systems or the Internet to the user for the purpose of search, reading, downloading or printing for non-profit purposes.

二、本授權書第一點所定授權，均為非專屬且非獨家授權之約定，本人仍得自行或授權任何第三人利用本著作。
The authorized agreement is non-exclusive permission. I retain the right to use the publication myself or to authorize any third party to use it.

三、本授權書第一點所定授權對象，依各該點授權利用本著作時，均應尊重本人著作人格權及權利管理電子資訊等相關權利，不得以任何方式省略、增修或變更本人署名、本著作名稱、本著作內容及相關資料（包括本人原記載取得學位論文之學校全銜、書目等詮釋資料等）。
The authorized parties specified in this authorized agreement, when using this publication, shall respect my moral rights and the rights related to the management of electronic information. They shall not omit, alter, or modify my authorship, the title of the publication, the content of the publication, or related information (including the full name of the school where I obtained my degree, bibliographic details, and other interpretative materials) in any way.

四、依本授權書第一點將本著作及其電子檔對外公開之時間 Public date of the publication：

☐ 於完成審核後立即對外公開 Public immediately after completion of the review

■ 於完成審核之六個月後對外公開 Public six months after completion of the review

☐ 本人要求本著作應自 Public access after 民國　　　年　　月　　日起始得對外公開，故因本授權書第一點所定授權亦應自該日起始生效力。

五、本授權書第一點所定各該授權對象，均應各自遵守其授權範圍及相關約定。如有違反，由該違反之行為人自行承擔一切法律責任
The authorized party of this authorized agreement shall comply with the scope of authorization and relevant agreements. In case of any violation, the person responsible for the violation shall bear all legal responsibilities."

六、本人擔保本著作為本人創作而無侵害他人著作權或其他權利。如有違反，本人願意自行承擔一切法律責任
I guarantee that this publication was created by me without infringing copyright or other rights of others. In case of any violation, I am willing to assume full legal responsibility.

七、個資利用同意條款 Consent for the Use of Personal Data：
本人同意，學校及國家圖書館為本授權書所定各授權事項目的範圍內得蒐集、處理及利用本人所提供之個人資料，學校並可將該等個人資料提供給包括國家圖書館在內之相關第三人在同一目的範圍內處理及利用
I agree that National Tsing Hua University and the National Central Library may collect, process, and use the personal data I provide within the scope

of the authorized matters specified in this authorized agreement. National Tsing Hua University may also provide such personal data to relevant third parties, including the National Central Library, for processing and use within the same purpose scope.

立授權書人 Signature of the author ： [illegible]

民國 114 年 6 月 30日 (Date in ROC :Year/MM/DD)

國立清華大學碩士學位論文
指導教授推薦書

National Tsing Hua University Master Thesis
Advisor Approval Form

資訊安全研究所
崔斯坦 君（學號112164421）所提之論文

CUCKOOGUARD：智慧網路卡上的節約記憶體SYN洪水攻擊防禦架構

經由本人指導撰述，同意提付審查。

Institute of Institute of Information Security
Mr.Tristan Subal Doring（Student ID: 112164421）who has submitted the Thesis/Dissertation

CUCKOOGUARD: A Memory-Efficient SYN Flood Defense Architecture for SmartNICs

under my guidance, I approve for the submission to the oral defense committee.

☑ 論文題目與內容符合本系(所、班、學位學程)專業領域
The subject and content of the thesis/dissertation are conformed to the professional field of the department (institute, class or program).

☑ 符合本系(所、班、學位學程)論文相似度比對標準
The thesis/dissertation meets the standard for the thesis/dissertation originality check set by the department (institute, class or program).

指導教授 (Advisor) [illegible] [illegible] (簽章) (Signature)

中華民國 114 年 6 月 11 日
2025 / 6 / 11 (YYYY/MM/DD)

國立清華大學碩士學位論文
考試委員審定書

National Tsing Hua University
Final Thesis/Dissertation Review Form

資訊安全研究所
崔斯坦 君（學號112164421）所提之論文

CUCKOOGUARD：智慧網路卡上的節約記憶體SYN洪水攻擊防禦架構

經本委員會審查，符合碩士資格標準。

Institute of Institute of Information Security
Mr.Tristan Subal Doring（Student ID: 112164421）who has submitted the Thesis/Dissertation

CUCKOOGUARD: A Memory-Efficient SYN Flood Defense Architecture for SmartNICs

has passed the oral defense and has met the qualifications to be awarded the degree of Master of Science.

學位考試委員會 (Oral defense Committee)	主持人 (Chair)	林靖茹	（簽章） (Signature)
	委　員 (Members)	孫宏民	
		黃俊穎	
		[illegible]	

中華民國　114　年　6　月　11　日
2025　/　6　/　11　(YYYY/MM/DD)

Acknowledgments

I am deeply grateful to my advisor, Professor Cheng-Hsin Hsu, whose insightful guidance, unwavering support, and encouragement to push beyond my limits have profoundly shaped both this thesis and my development as a researcher. His mentorship not only cultivated my independent thinking but also nourished my curiosity and passion for research.

My sincere appreciation goes to all my friends and fellow National Tsing Hua University students for their openness, camaraderie, and the welcoming environment they created. Their support made my time here both enriching and memorable.

中文摘要

SYN 洪水攻擊持續挑戰伺服器的擴展性與可用性。現有的防禦方法不是加重主機 CPU 的負擔，就是超出可程式化硬體的記憶體限制。本文提出 CUCKOOGUARD：智慧網路卡上的節約記憶體 SYN 洪水攻擊防禦架構，透過分散式代理設計（split-proxy），將連線驗證卸載至資料平面，以提升效能並降低主機負載。CUCKOOGUARD 採用 Cuckoo 濾波器來精確追蹤 TCP 連線，並支援移除過時的連線項目，有效提升記憶體使用效率並支援高連線流動性。系統以 P4 語言實作，並與現有技術進行效能比較，CUCKOOGUARD 將流量過濾的誤判率由 7.66% 降低至 1.56%，使主機 CPU 負載減少達 79%。實驗結果顯示，在資源受限的可程式化平台上，亦能實現準確且低負擔的 SYN 洪水攻擊防禦。

Abstract

SYN flood attacks continue to challenge server scalability and availability. Existing defenses either burden the host CPU or exceed the memory limits of programmable hardware. We present CuckooGuard, a memory-efficient SYN flood mitigation architecture for SmartNICs that offloads connection validation to the data plane using a split-proxy design. At its core, CuckooGuard uses Cuckoo filters to enable precise TCP connection tracking, including the removal of stale connection entries. This enhances memory efficiency and supports high connection turnover. Implemented in P4 and benchmarked against the state-of-the-art, CuckooGuard reduces flow filtering false positive rates from 7.66% to 1.56%, yielding a 79% reduction in server CPU overhead. These results demonstrate that accurate, low-overhead SYN flood defense is achievable on resource-constrained programmable platforms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the context of escalating geopolitical tensions in recent years, cyberspace has increasingly become a battleground for state-sponsored and criminal actors alike, leading to a notable surge in cyber attacks targeting critical infrastructure and digital services [15, 31]. Among these, Distributed Denial-of-Service (DDoS) attacks have emerged as a particularly persistent threat. According to Cloudflare's Q4 2024 report, a substantial proportion of these attacks target the Layer 4 TCP protocol, with SYN flood attacks standing out as the most prevalent form of network-layer DDoS activity [20]. SYN flood attacks exploit the TCP handshake mechanism to overwhelm server resources, thereby impairing system availability. Although stateless countermeasures, such as SYN-Cookies [8], have been widely adopted, their reliance on host CPU resources renders them increasingly ineffective due to the scale and frequency of contemporary DDoS campaigns. This evolving threat landscape underscores the urgent need for more sophisticated and scalable defense frameworks that can operate effectively under high-load scenarios and adapt to the dynamic nature of modern cyber warfare.

Broader defenses, such as SDN-based filtering [91], offer coarse-grained protection, e.g., by blocking spoofed or known malicious IPs. However, they lack the per-flow granularity needed for effective SYN flood mitigation and are best viewed as complementary solutions. Recent work has thus focused on in-network defenses implemented entirely in the data plane [77]. One notable example is SMARTCOOKIE [92], which uses a split-proxy design to offload SYN-Cookie validation from the server to programmable network hardware. While effective, it relies on programmable switch hardware that has since been discontinued by the manufacturer [43], its scalability is limited, and it incurs significant server overhead due to a 7.66% false positive rate in flow filtering.

In this work, we present CUCKOOGUARD, a memory-efficient SYN flood defense architecture designed for deployment on Smart Network Interface Cards (SmartNICs). These widely used devices [33] offer line-rate packet processing near the server edge and support programmable data planes via P4 [39], making them ideal for handling fine-grained, low-latency traffic. We implement a proof-of-concept of CUCKOOGUARD using the P4 language [62] to validate our approach. As a widely adopted domain-specific language (DSL) for programmable data planes, P4 enables hardware-agnostic yet efficient implementations, making it well-suited for deployment across SmartNIC platforms.

Unlike prior SYN-Proxy approaches such as SMARTCOOKIE [92], which relies on time-decaying Bloom filters for flow filtering, CUCKOOGUARD aims to provide precise and memory-efficient flow filtering. While an array of set-membership data structures was considered, the Cuckoo filter, as introduced by Fan et al. [32], crystallized itself as the correct data structure for this role. However, implementing Cuckoo filters in programmable data planes presents notable challenges. Specifically, the element insertion procedure involves conditional loops, typically unsupported or discouraged in data plane programming due to pipeline constraints. Consequently, performance implications must be thoroughly evaluated, and the filter configuration must be carefully tuned to mitigate overhead and preserve line-rate processing. To address these challenges, CUCKOOGUARD employs a tailored Cuckoo filter design that balances precision and efficiency, enabling accurate flow tracking while remaining compatible with the constraints of programmable data planes. This enables precise per-connection tracking with support for dynamically removing flow entries, significantly reducing memory overhead. Our implementation demonstrates that CUCKOOGUARD offers both precision and practicality, particularly in SmartNIC environments where memory resources are limited due to hardware limitations and the need for multiple concurrent network functions [38] deployed on a single device.

1.1 Contributions

Compared to state-of-the-art SYN-Proxy solutions, CUCKOOGUARD offers the following contributions:

- **Improved CPU Offload:** Lower false positive rates at the flow filter reduce unnecessary cookie verifications at the server, decreasing server-side CPU overhead by up to 79%.

- **A Novel Memory-Efficient and Precise TCP-Connection Tracking Scheme:** Compact and precise connection tracking that uniquely supports the deletion of individual flow entries (corresponding to terminated connections) and sustains high connection churn, even when severely memory-constrained.

1.2 Thesis Organization

This thesis is structured to systematically introduce, design, implement, and evaluate CUCKOOGUARD a hardware-accelerated defense architecture against SYN flood attacks in programmable data planes.

Chapter 2 establishes the necessary background to understand the environment in which modern network functions operate. It begins by contextualizing DDoS attacks, focusing on SYN floods, and introducing the limitations of conventional mitigation techniques. The chapter then presents the principles of data plane programmability, highlighting P4 as a domain-specific language, and delves into P4-capable hardware targets, emphasizing Field-Programmable Gate Array (FPGA)-based SmartNICs. The background concludes by reviewing memory-efficient approximate filter data structures, which form a key component in the proposed design.

Chapter 3 surveys the state of the art in SYN flood mitigation within programmable network environments. Existing approaches are categorized into three main classes: Machine Learning (ML)-based solutions, rule-based defenses, and SYN-Cookie-based architectures. Their respective benefits and limitations are analyzed in preparation for the architectural contributions of this thesis.

Chapter 4 consolidates the background and related work to define the precise objective of this thesis: the development of a SYN flood defense architecture tailored for deployment on hardware-accelerated, programmable data planes—specifically FPGA-based SmartNICs programmed using P4. Two representative case studies are introduced to motivate this need, from which a set of architectural requirements is derived.

Chapter 5 presents the CUCKOOGUARD architecture. It identifies the targeted environment's most appropriate flow filter data structure. Then, it justifies the selection of the Cuckoo filter based on memory efficiency, precision, and deletion support. The chapter then introduces our CUCKOOGUARD architecture centered around SYN-Cookie validation and Cuckoo filter-based flow tracking. CUCKOOGUARD is subsequently evaluated against the previously defined requirements, demonstrating comprehensive alignment.

Chapter 6 discusses the implementation of CUCKOOGUARD and the technical challenges encountered throughout the development process. Special attention is given to implementing Cuckoo filter insertion logic within the constraints of programmable data planes, as it represents a non-trivial contribution rarely addressed in existing systems.

Chapter 7 provides an empirical assessment of CUCKOOGUARD. The experiments verify the functionality of the architecture and identify optimized configuration parameters. A comparative evaluation against the SMARTCOOKIE baseline highlights CUCKOOGUARD's superior performance in server offloading and filter precision, significantly reducing computational overhead.

Finally, Chapter 8 summarizes the key findings of this thesis, outlines the lessons learned, and discusses promising directions for future research and development.

Chapter 2

Background

This chapter provides the reader with the background information necessary to understand the following thesis.

2.1 DDoS Attacks in Modern Networks

DDoS attacks fall under the broader category of volumetric attacks [18], whose goal is to saturate bandwidth or exhaust system resources. Three common techniques form the basis of most DDoS operations: flooding, spoofing, and amplification. Flooding refers to overwhelming a target with high volumes of traffic, often crafted to trigger resource-intensive reactions. Spoofing involves forging packet headers—typically the source IP address—to conceal the origin of traffic or to bypass filtering mechanisms. Amplification, on the other hand, abuses legitimate services such as DNS or NTP to generate disproportionally large responses from small requests [46].

These techniques are not mutually exclusive and are frequently combined in real-world attacks. For instance, SYN flood attacks usually exploit the TCP handshake by initiating handshakes with spoofed source IP addresses, tricking the server into handling packets from the same source as if they came from different legitimate users. This causes the server to perform costly operations for each incoming SYN flood packet, thereby straining system resources.

2.2 SYN Flood Defenses

The primary scenario considered throughout this thesis involves a typical client-server architecture, in which a server passively awaits TCP connection attempts initiated by clients. Upon reception of a `SYN` packet, the server responds by sending a `SYN-ACK` packet. The server also temporarily allocates memory for storing metadata related to this incomplete, or *half-open*, connection. This metadata includes essential information such as connection tuples, sequence numbers, and timestamps, and is managed within a data structure commonly referred to as the SYN backlog. The allocated memory remains reserved until the handshake is successfully completed by the arrival of the final `ACK` packet.

If, however, this concluding `ACK` never arrives—characteristic of SYN flood attacks—the allocated backlog entry persists until a predefined timeout period, usually several seconds, expires. Consequently, in a SYN flood scenario, the attacker deliberately generates a high volume of spoofed or unresponsive `SYN` packets, rapidly saturating the backlog and exhausting the server's available memory resources. This effectively prevents legitimate users from establishing new TCP connections, causing a denial-of-service.

2.2.1 Conventional Methods

Several conventional methods attempt to mitigate this form of resource exhaustion. One such approach, the *SYN cache* [48], replaces comprehensive connection state entries with smaller, lightweight fingerprints to reduce memory usage per connection. While this increases an attacker's effort, it does not fundamentally prevent memory exhaustion, especially given increasingly large attack volumes. An alternative and widely adopted strategy is *rate limiting* [81], where the server imposes a hard limit on the number of connection attempts it processes within a certain time frame. Although effective in reducing resource usage, rate limiting does not distinguish between legitimate and malicious traffic, thus allowing attackers to cause a denial-of-service for legitimate clients through sheer request volume.

Due to these inherent limitations, more sophisticated solutions have been developed, among which SYN-Cookies have proven particularly effective.

2.2.2 SYN-Cookies

Initially introduced by Bernstein [8], SYN-Cookies represent a robust and widely adopted approach to defending against SYN flood attacks. Traditional defense mechanisms require the server to immediately allocate memory upon receiving the initial `SYN` packet and store metadata until the handshake is complete. SYN-Cookies, however, circumvent this early resource allocation by maintaining a completely stateless server during the initial stages of the TCP handshake. The protocol-level operations can be traced in Figure 2.1. Specifically, instead of storing connection details directly in memory, the server encodes critical connection parameters—such as source and destination IP addresses, port numbers, and timestamps—into the TCP sequence number field of the `SYN-ACK` response packet.

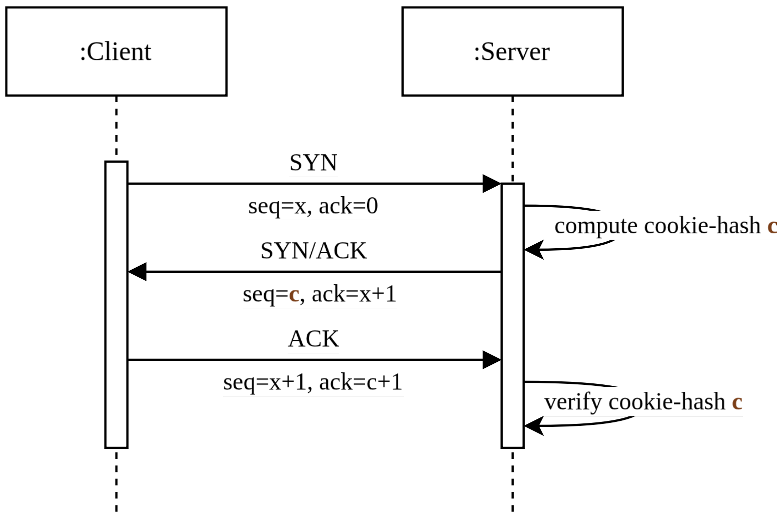

Figure 2.1: Visualization of SYN-Cookie TCP handshake and connection validation mechanism.

This encoded data, called a ”cookie”, effectively eliminates the need to maintain server-side state for incomplete connections. The cookie-hash c is protected from forgery by utilizing a cryptographic hash function (e.g., SipHash [21]) for encoding. Only after the client responds with a valid `ACK` packet containing this cookie does the server reconstruct and verify the embedded connection information. Upon successful verification, the server then formally allocates memory resources and establishes the connection. This strategy significantly reduces the server’s vulnerability to SYN flood attacks by ensuring resources are dedicated exclusively to verified, legitimate connection attempts. However, when SYN Cookies are deployed on commodity servers, this can lead to an exhaustion of computational resources, as each cookie generation and validation requires an expensive hash calculation. This computational expense can be offloaded as a network function to programmable data planes with abundant computational resources, as explored in the next section.

2.3 Programmable Data Planes

In contemporary networks, a clear separation exists between the control plane and the data plane. While the control plane manages network configuration, routing decisions, and policy enforcement, the data plane handles actual packet processing tasks, such as forwarding, filtering, and packet transformations. Traditionally, data plane functions in high-performance environments have been implemented using fixed-function hardware with limited to no programmability. However, recent technological developments in programmable data planes now allow network operators to dynamically develop, modify, and deploy custom network functions directly in the data plane while retaining line-rate performance [53].

Programmable data plane solutions are characterized by a combination of varying degrees of high performance, flexibility, and ease of programmability. They can be implemented either purely in software running on general-purpose hardware or on specialized hardware. Software-based implementations provide maximum flexibility but face challenges in achieving sufficiently high throughput. To tackle this, specific frameworks and languages have emerged. For instance, eBPF (extended Berkeley Packet Filter) [28] and DPDK (Data Plane Development Kit) [24] optimize software execution paths to minimize processing overhead, whereas the P4 language [62] was explicitly designed to leverage hardware acceleration in programmable network devices. The fundamental design differences and use cases of these programming paradigms will be elaborated in the following sections.

2.3.1 P4

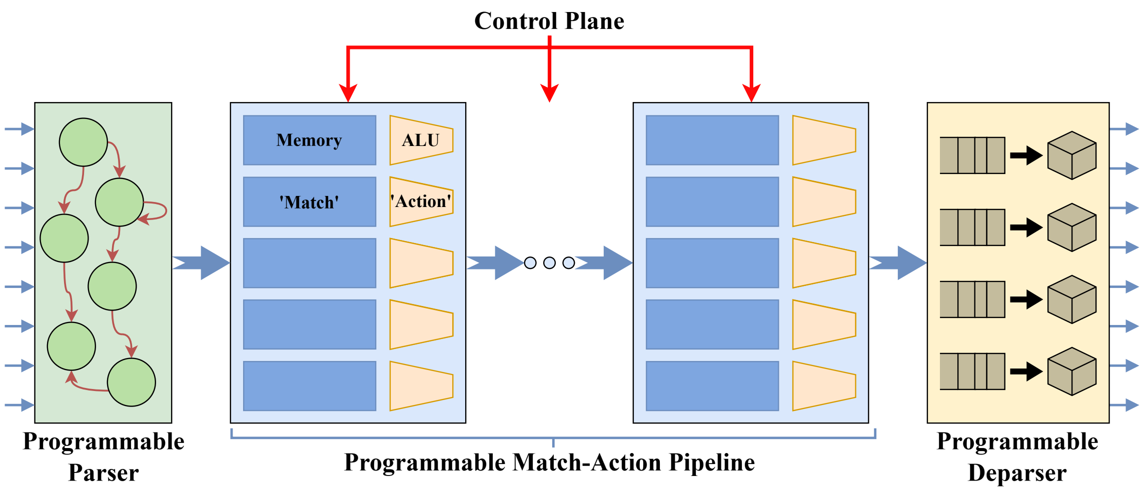

Figure 2.2: P4's match-action pipeline architecture.

The P4 language [12] is a DSL explicitly designed for programming packet-processing logic in programmable network devices such as programmable switches and SmartNICs [39]. Unlike general-purpose programming languages, P4 adheres to a structured *match-action* pipeline model (see Figure 2.2). In this model, packets are first parsed, i.e., their header fields are extracted in a protocol-layered fashion. This programmable parser is protocol-agnostic, allowing it to be adapted to existing or novel network protocols.

Following parsing, packets enter the programmable match-action pipeline, where they are repeatedly classified based on selected header fields. Predefined actions—such as forwarding, modifying, or dropping a packet—are then applied using so-called match-action tables. While the general structure of these tables (e.g., which packet fields to match and what set of actions is available) is defined statically in P4 code, their content can be dynamically configured at runtime by the control plane. This separation between static table layout and dynamic table population enables flexible and adaptive network functions. For instance, firewalls and access control mechanisms can update their behavior in response to changing traffic patterns without needing to modify the core P4 program. This control-plane-driven deployment model makes P4 particularly well-suited for dynamic and stateful network applications.

After passing through the match-action pipeline, packets are reassembled in the deparser based on the outcomes of prior processing and are then emitted from the device.

Despite its flexibility, P4's strict pipeline semantics constrain general-purpose programmability, most notably through the absence of native looping constructs, which renders the language not Turing-complete. Iterative behavior is only supported through controlled packet recirculation, where packets exiting the egress pipeline are redirected back to the ingress for repeated processing. While this mechanism introduces a throughput trade-off due to additional pipeline passes, it provides a practical solution for use cases that require limited iteration.

Importantly, these constraints are a deliberate design choice. P4's deterministic, hardware-conscious model is optimized for high-speed, line-rate processing and predictable execution behavior. Rather than limiting expressiveness, this focus on structured programmability ensures that network functions written in P4 are both efficient and portable across a range of modern, performance-critical network devices.

2.3.2 eBPF

The extended Berkeley Packet Filter (eBPF) allows executing user-defined programs directly within the Linux kernel [39]. eBPF programs, typically written in a restricted subset of C, are dynamically loaded into the kernel at runtime. By running in kernel space, eBPF efficiently intercepts and processes packets at various points in the networking stack, eliminating costly context switches. Two common hook points are the eXpress Data Path (XDP) and Traffic Control (TC), each suited to different types of network functionality:

- **XDP** [83] operates at the earliest possible point in the networking stack, directly after the NIC driver receives a packet but before it enters the conventional kernel pipeline. This early positioning allows XDP to drop, redirect, or modify packets with minimal overhead, making it ideal for high-speed processing.

- **TC** [27], on the other hand, is situated later in the stack, just before packets are handed to user space or forwarded by the kernel. TC programs are suitable for more complex operations that require awareness of routing and queuing decisions. Additionally, TC supports cloning packets and redirecting them to multiple destinations [45].

The primary benefit of eBPF is its flexibility and transparency to Linux user-space software, which allows for real-time modifications to packet processing logic without disrupting running services. However, because eBPF programs execute in kernel space, they are subject to strict resource constraints and security checks: unbounded loops, unsafe memory access, and blocking operations are disallowed. This limits programming expressiveness compared to general-purpose user-space environments, but ensures system stability and predictable performance. While eBPF may not reach the throughput of specialized hardware data planes, such as those written in P4, it offers practical advantages on general-purpose hardware, including access to ample memory resources and easier integration with existing systems. This makes it well-suited for high-speed network functions that require flexibility and adaptability.

2.3.3 DPDK

The Data Plane Development Kit (DPDK) consists of libraries and drivers designed to facilitate high-performance packet processing entirely in user space, bypassing the kernel networking stack [39]. By directly accessing hardware resources from user space, DPDK avoids the overhead associated with kernel interactions, enabling line-rate packet processing on general-purpose CPUs.

Compared to eBPF, DPDK offers greater programming flexibility, simpler debugging, and fewer kernel-level restrictions. DPDK applications are typically developed in general-purpose languages such as C. However, completely bypassing the kernel can increase the complexity of integration within existing Linux environments and introduce additional operational overhead. Unlike eBPF, DPDK-based applications are not transparent to the rest of the system, which may complicate coordination with other networking components or tools.

2.4 P4 Hardware Acceleration

Programmable data planes offer a flexible alternative to traditional fixed-function network devices. However, when implemented purely in software, they often face performance bottlenecks. The pace of improvement in general-purpose CPU throughput has not kept up with the growing demands of modern network traffic [39, 40], making the role of network domain-specific accelerators increasingly important. While frameworks such as eBPF and DPDK enable high-speed packet processing on commodity hardware, they are fundamentally limited by the computational capabilities of modern CPUs.

To address these limitations, modern network systems are increasingly turning to hardware-accelerated, programmable data planes. In this work, we focus on those based on P4. Devices such as programmable switches (e.g., Intel Tofino [42]) and SmartNICs are specifically designed to run data plane network functions at line rate. These platforms combine the flexibility of software programmability with the speed and determinism of specialized hardware, reaching processing rates of multiple terabits per second in some deployments [57].

To understand how P4 targets achieve this, it is first necessary to consider the architecture model they expose. Before discussing individual hardware platforms, the following subsection introduces the concept of P4 architectures, which serve as a bridge between high-level P4 programs and bare-metal hardware.

P4 Architectures

To bridge the gap between the abstract P4 language and the specific capabilities of target hardware, the P4$_{16}$ specification [62] introduces the concept of P4 architectures. A P4 architecture defines the structure of the programmable data plane, for example, specifying components such as parsers, match-action pipelines, and deparsers, and formalizes how P4 programs interact with these components through well-defined interfaces. Different targets implement different architectures to reflect their design and capabilities. While the core P4 language remains portable, each architecture exposes a specific logical view of the processing pipeline and may offer device-specific features through external components. These externs enable support for additional operations, such as hash calculations, counters, meters, or random number generators, which are, for example, implemented in hardware outside the core packet-processing logic. This architectural abstraction allows P4 to remain flexible across various targets while still enabling platform-specific optimizations. However, it also means that P4 programs are generally tied to a specific architecture and remain limited in their portability.

2.4.1 Hardware Targets

P4 programs can be deployed on different types of hardware, depending on where the processing is performed in the network. As illustrated in Figure 2.3, two prominent target classes are programmable switches, which operate in the network core and process high-throughput traffic across many ports, and SmartNICs, which are typically deployed at the edge and attached to individual servers, where they offload specific packet-processing

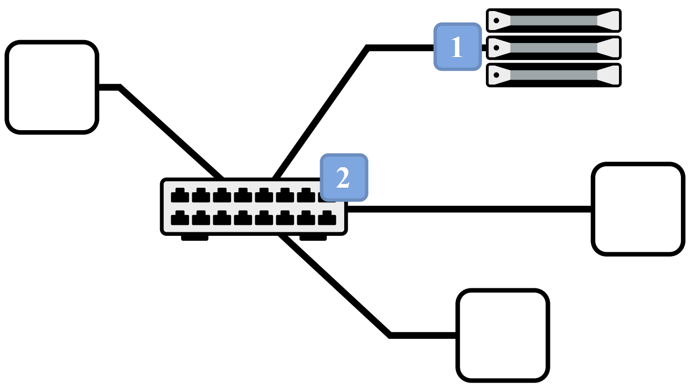

Figure 2.3: Locations of P4-programmable hardware in the network: SmartNICs and programmable switches.

tasks [79].

Programmable Switches

Among P4 hardware targets, programmable switches are designed to handle high-volume, multi-port traffic in parallel, making them especially well-suited for deployment in aggregation or backbone layers of the network. Although P4 is hardware-agnostic, the Intel Tofino is the only widely used and practically deployed programmable switch platform today. It is built around a specialized high-performance ASIC capable of full line-rate processing and has become the de facto standard for P4-based switch deployments [42]. However, Intel has recently announced the end of life for Tofino devices [43], raising concerns about the long-term sustainability of ASIC-based P4 switch hardware.

Tofino follows a highly parallel match-action pipeline architecture and supports the Tofino Native Architecture (TNA)—a custom P4 architecture that extends the open PSA (Portable Switch Architecture [60]) model [39]. TNA introduces a set of useful features, including multiple programmable pipeline components, advanced non-cryptographic hash functions, and registers for stateful processing. Notably, Tofino can support up to 12.8 Tbit/s total throughput, with multiple independent pipelines that can be run in parallel or chained together for more complex network functions [42]. The architecture also supports a wide range of built-in primitives for tasks such as packet recirculation and telemetry.

P4 development for Tofino is done via Intel's P4 Studio [41, 65], which provides a complete development toolchain, including a dedicated compiler, runtime SDKs, and P4i—a visualization tool that maps P4 logic onto hardware resources. These tools allow developers to optimize their programs for the chip's parallel architecture and memory layout.

Tofino-based switches are commercially available in white-box systems from vendors such as EdgeCore [29] and APS Networks [4], and are also embedded in proprietary systems from major network equipment manufacturers like Arista [5] and Cisco [19]. These devices typically feature a separate control-plane CPU running Linux, which handles device management, configuration, and interaction with the P4 runtime environment.

SmartNICs

SmartNICs (Smart Network Interface Cards) are programmable network adapters that offload packet-processing tasks from the host CPU to the NIC itself, offering increased performance, lower latency, and improved overall efficiency [23]. Unlike ASIC-based NICs, which are fixed-function by design, SmartNICs are built around reprogrammable architectures that expose varying degrees of flexibility and performance [39]. They are typically deployed at the edge of the network, attached to a single server, though recent trends have explored broader deployment models that integrate SmartNICs into distributed network infrastructures [79].

At a high level, SmartNICs can be categorized into two main architectural classes: those based on general-purpose processors (GPPs), and those built around field-programmable gate arrays (FPGAs). Each class presents different trade-offs in terms of programmability, performance, and development complexity.

GPP-based SmartNICs. GPP-based SmartNICs are centered around embedded processors such as ARM cores. They benefit from mature software ecosystems, high-level programming support, and relatively low barriers to development [23, 30]. These devices are well-suited for tasks that require tight integration with user-space applications, software-defined control logic, or flexible packet inspection. However, they inherit fundamental limitations from their CPU-centric design: packet processing is sequential, throughput scales poorly under load, and execution latency is variable and harder to predict. These constraints limit their applicability in latency-critical or throughput-intensive scenarios.

A notable extension of this design is the SoC-based SmartNIC, which integrates fixed-function accelerators, such as cryptographic engines or compression units—alongside the GPP cores. For example, NVIDIA's BlueField architecture combines ARM CPUs with dedicated units for in-line encryption or data processing [58, 49]. While such devices can reach line rate for specific tasks, the performance of fully custom network functions remains constrained. The accelerators are typically fixed-purpose, and the CPU cores are not designed to handle full-scale data plane workloads.

FPGA-based SmartNICs. FPGA-based SmartNICs, in contrast, offer a much higher degree of architectural flexibility and are particularly well-suited for fully customized,

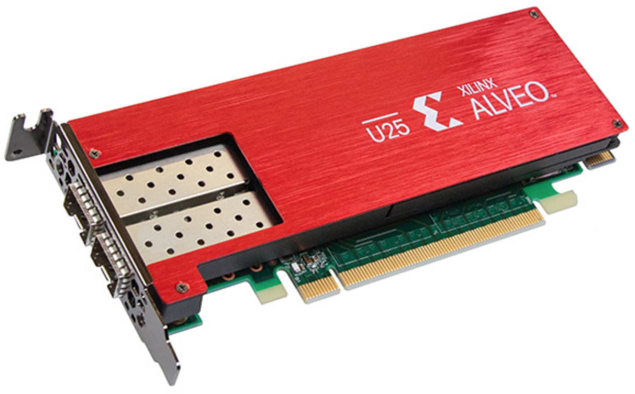

Figure 2.4: AMD (Xilinx) U25 SmartNIC [88].

high-performance data plane programming. These devices consist of reconfigurable logic blocks, programmable interconnects, and embedded hardware components such as Block RAM (BRAM) and Digital Signal Processing units (DSPs) [39]. The defining feature of FPGAs is that all processing logic executes in parallel, unlike the sequential execution model of CPUs. This enables FPGA-based SmartNICs to implement fine-grained, deeply pipelined network functions with consistent latency and true line-rate throughput.

Beyond parallel logic execution, FPGA architectures also support highly efficient memory access patterns. In particular, on-chip memories such as BRAM naturally enable fast, parallel lookup operations across multiple entries. It is common for FPGA-based SmartNIC architectures to implement Content Addressable Memories (CAMs) using BRAM resources, allowing packet header fields to be matched against a large number of flow entries simultaneously [84]. Unlike standard RAM, where an address must be provided to retrieve data, a CAM takes a search key as input and returns the matching memory address directly, facilitating constant-time lookup performance [1]. This memory model is particularly promising for network applications that require fast, scalable flow state management.

From a performance-per-watt perspective, FPGAs are significantly more efficient than CPUs and offer a compelling trade-off compared to ASICs. While they consume more power than fixed-function hardware, they remain fully reprogrammable, providing a cost-effective and flexible solution for evolving network applications [30, 33]. Their main drawback lies in development complexity: programming FPGAs typically requires specialized knowledge of hardware design and toolchains. However, the increasing adoption of high-level abstractions like P4 is gradually reducing these barriers, making FPGA-based platforms increasingly accessible for network function development.

Experience with the AMD Alveo U25 FPGA-based SmartNIC. While vendor toolchains have made strides in making FPGA-based SmartNICs more accessible, programming them remains a nontrivial task. For example, AMD (formerly Xilinx) provides VitisNetP4 [3], a P4 IP (Intellectual Property) block that integrates into the broader FPGA development environment. While this represents a significant step toward abstraction, deploying a functional P4 application still requires deep familiarity with the FPGA toolchain and architecture. As part of this thesis, the AMD (Xilinx) Alveo U25 [90] SmartNIC (shown in Figure 2.4) was thoroughly investigated as a potential P4 hardware target. However, support for this device appears to have been discontinued despite its release in 2020. Although documentation advertised P4 programmability, actual deployment involved manually integrating IP blocks into a larger hardware design—something not easily achieved without prior experience in FPGA development. Due to these complexities and limited vendor support, the Alveo U25 was ultimately dropped as a target for experimentation.

Newer initiatives, such as AMD's OpenNIC project [89], attempt to address these challenges by offering predefined FPGA architectures with well-defined interfaces into which custom P4 logic can be inserted. This significantly reduces the integration burden and is a step toward more accessible and modular development. Nonetheless, the overall state of tooling and documentation continues to vary widely between vendors and device generations.

Standardization remains a major challenge in the SmartNIC ecosystem. While the P4 community has proposed the Portable NIC Architecture (PNA) to promote cross-platform compatibility, major vendors such as AMD and Netronome have yet to adopt it in practice. Most SmartNICs are shipped with their own P4 architecture models and rely on vendor-specific toolchains and interfaces. As a result, developers often have to adapt their programs for each target individually, which complicates deployment and limits P4 code portability.

Focus on FPGA-based SmartNICs. Despite these practical limitations, FPGA-based SmartNICs remain the most compelling platform for high-performance, programmable data planes. Their ability to implement the match-action abstraction of P4 in hardware enables the deployment of custom line-rate network functions. Several works [33, 30] reinforce this view, arguing that FPGA-based SmartNICs strike a practical balance between flexibility, performance, and deployment cost. Given these considerations, this thesis focuses primarily on FPGA-based SmartNICs as the most suitable hardware platform for programmable data planes.

2.4.2 The BMv2 Software Target

While hardware platforms offer impressive capabilities for high-speed packet processing, they often come with considerable complexity in terms of toolchains, deployment workflows, hardware cost and availability. To support the development and validation of P4 programs in more accessible environments, the P4 community maintains a software reference implementation known as BMv2 (Behavioral Model version 2 [63]).

BMv2 is a C++-based interpreter that executes the behavior defined by P4 programs compiled via the p4c [66] compiler. Although designed to simulate switch architectures, BMv2 is highly generic and flexible and can be used to prototype a wide range of network functions, including those intended for deployment on SmartNICs. Its architecture accommodates key features, such as match-action tables, parsers, deparsers, and externs. BMv2 also supports integration with standardized control-plane APIs through P4Runtime [67, 61], which communicates via gRPC, enabling dynamic control over the data plane. Moreover, its compatibility with network emulation environments such as Mininet [56, 54] has made BMv2 a popular choice in academic and experimental research. This ecosystem allows researchers to simulate complex topologies and observe P4 behavior under realistic traffic patterns without requiring specialized hardware.

Unlike hardware targets, BMv2 does not aim for line-rate performance. It prioritizes functional correctness, portability, and observability. Depending on the complexity of the deployed program and the host system, throughput can reach up to 1 gigabit per second [64]. However, its strengths lie elsewhere: BMv2 offers line-by-line debugging, extensive logging, and reproducibility—features that are essential for rapid prototyping, controlled experimentation, and iterative development.

For these reasons, BMv2 was selected as the primary target for the implementation and experimental evaluation in this thesis. It provides a stable, accessible, and sufficiently expressive platform for exploring and validating the behavior of P4-based network functions, while preserving basic compatibility with future hardware deployment paths.

2.5 Memory-Efficient Approximate Filters

Probabilistic set-membership data structures, commonly referred to as filters [47], offer a compact and efficient way to represent sets $S \subseteq U$ by supporting approximate membership queries, where U represents the universe of possible elements. Given a query element $x \in U$, a filter answers whether $x \in S$, with a bounded false positive probability ε, but no false negatives for our purposes. In contrast to exact data structures such as hash tables or dictionaries, filters can reduce memory usage by an order of magnitude or more, making them well-suited for memory-constrained environments such as programmable data planes. This efficiency is achieved by relaxing correctness guarantees: elements not in the set may occasionally be reported as present.

For the purpose of this thesis, we distinguish two categories of filters [70]:

- **Semi-dynamic filters**, which support insertions but not deletions. These are suitable when the set grows monotonically. A canonical example is the Bloom filter [10].
- **Dynamic filters**, which support both insertions and deletions. Examples include the Cuckoo filter [32] and the Quotient filter [7].

In this section, we consider only *fixed-size* variants of these filters, that is, data structures whose memory footprint remains constant over time. This restriction reflects the deployment context of programmable data planes, where static memory allocation is standard. Only filters that guarantee the absence of false negatives are considered.

Motivation and Use Cases Set-membership filters are widely employed in systems that must process large volumes of data while maintaining minimal memory overhead. Their utility arises from the ability to provide fast, approximate lookups in throughput-sensitive and memory-constrained environments.

Common application domains include databases and storage systems, where filters are used to avoid costly disk lookups. For instance, Google Bigtable integrates Bloom filters to pre-filter row key queries before accessing SSTables [16]. Additionally, in network measurement and classification, filters allow efficient identification of known flows or addresses at line rate. For example, Cuckoo filters have been used in high-speed telemetry systems for accurate flow tracking [86]. These examples illustrate the contexts where filters can yield drastic memory savings—often by multiple orders of magnitude—without compromising latency or query performance, particularly in environments where a small and controllable false positive rate is acceptable.

2.5.1 Bloom Filter

The Bloom filter, introduced by Burton Bloom in 1970 [10], is the foundational structure among probabilistic set-membership data structures. Its simplicity has led to widespread adoption across storage, networking, and database systems [80, 76]. The Bloom filter provides approximate set membership with a guaranteed absence of false negatives.

Conceptually, a Bloom filter represents a set $S \subseteq U$ using a bit array $B \in \{0,1\}^m$ and a collection of k independent hash functions $h_1, h_2, \ldots, h_k$, where each $h_i : U \to \{0, \ldots, m-1\}$ [47]. Initially, all bits in the array are set to zero. Insertions and queries are handled by accessing the positions determined by the hash functions.

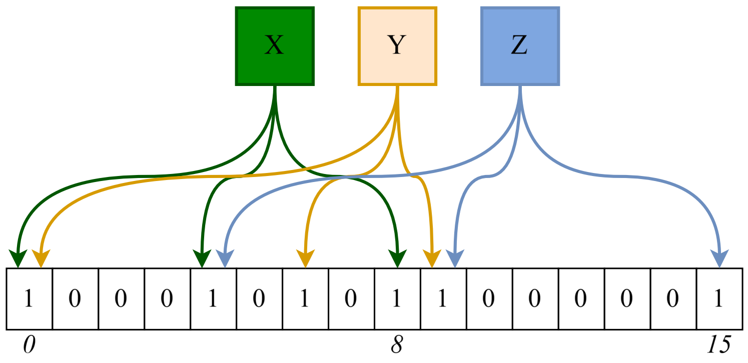

Figure 2.5: Standard Bloom filter example with $k = 3$ hash functions applied to three distinct elements: X, Y, and Z. Each element sets bits at positions determined by the corresponding hash functions, illustrating the fundamental insertion mechanism.

Insertion and Querying To insert an element $x \in U$, it is hashed using each of the k hash functions, and the resulting bits $B[h_i(x)]$ for $i = 1, \ldots, k$ are set to 1 [47]. The insertion procedure is illustrated in Figure 2.5, where three elements are inserted using three hash functions, each mapping to distinct positions in the bit array. To query whether an element $y \in U$ is in the set, the same hash functions are used. If all $B[h_i(y)]$ are set, the filter returns "possibly in the set." If any bit is 0, the filter concludes that $y \notin S$.

Due to shared bit usage among multiple elements, false positives may occur when all relevant bits are incidentally set by other elements. However, false negatives are impossible.

Design Considerations and Variants The false positive rate $\varepsilon \in (0, 1)$ depends on the number of inserted elements $n = |S|$, the bit array size m, and the number of hash functions k. For a fixed n, one can optimize m and k to minimize ε [73]. A space-optimized Bloom filter has a per-element space requirement of approximately [32]:

$$\text{bits per item} = 1.44 \cdot \log_2(1/\varepsilon). \tag{2.1}$$

This is close to the theoretical lower bound of $\log_2(1/\varepsilon)$ bits per element. Several Bloom filter variants exist to address practical constraints. In the *partitioned Bloom filter* [2], the bit array is divided into k equal sections, one per hash function. This design enhances the uniformity of false positives. In contrast, the standard *non-partitioned* Bloom filter uses a shared array, which can be more space-efficient but may suffer from clustered bit saturation.

Due to their low memory usage and simplicity, Bloom filters remain a key building block for set membership testing in resource-constrained systems.

2.5.2 Cuckoo Filter

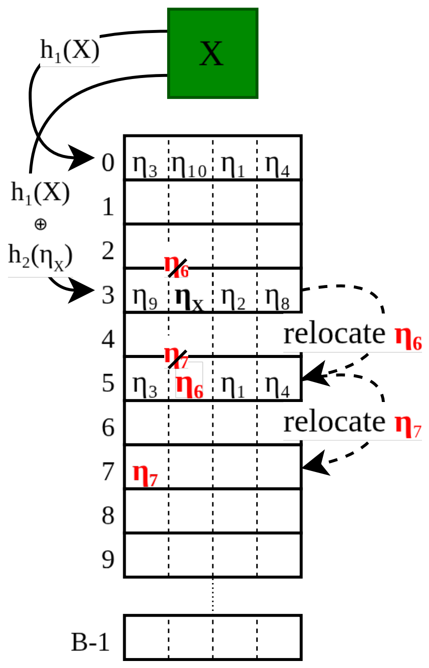

Figure 2.6: Insertion of element X into Cuckoo filter where both candidate buckets are full. Two relocation steps are required before successful insertion.

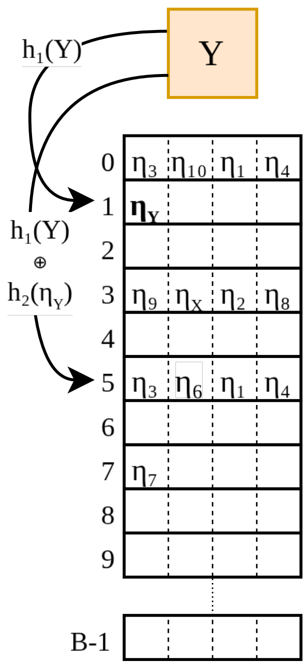

Figure 2.7: Insertion of element Y into Cuckoo filter where one candidate bucket has space. The fingerprint η_y is inserted directly without relocation.

The Cuckoo filter is a dynamic set-membership data structure derived from *Cuckoo hashing* [68]. In contrast to Bloom filters, it supports both insertions and deletions while maintaining similar guarantees of space efficiency and bounded false positives. It is particularly well-suited for use cases where elements must be frequently removed or replaced. Figures 2.7 and 2.6 illustrate the core components and mechanisms of Cuckoo filters and will be referenced throughout this section.

In a Cuckoo filter, each element $x \in U$ is represented not by its full value but by a compact *fingerprint* $\eta_x \in \{0,1\}^f$. This fingerprint is stored in one of two candidate buckets within an array of buckets of size B. These bucket positions are computed as $h_{1(x)}$ and $h_{1(x)} \oplus h_{2(\eta_x)}$, where h_1 maps the original item to a bucket index, and h_2 maps fingerprints to XOR offsets. This process is illustrated in Figures 2.7 and 2.6, where elements X and Y are each mapped to two candidate buckets. Each bucket can accommodate up to b entries.

A core enabling feature of Cuckoo filters is the use of *partial-key cuckoo hashing* [32]. Since only the fingerprint is stored in the filter, the second candidate bucket must be derived from the original bucket index and the fingerprint itself, rather than from the full element. This approach enables compact storage but limits the flexibility of the insertion process.

Insertion, Querying, and Deletion To insert an element, the algorithm first computes its fingerprint and attempts to store it in either of its two candidate buckets. If both are full, the filter randomly selects a fingerprint from one of the buckets, evicts it, and relocates it to its alternate bucket. This process may repeat until an empty slot is found or a relocation threshold is reached, at which point the insertion is deemed unsuccessful. The insertion process is illustrated in Figures 2.6 and 2.7, which show the relocation and direct insertion cases, respectively. Together, they reflect the adaptive behavior of Cuckoo filters in response to varying table occupancy.

To test whether an element $y \in U$ is in the set, the filter checks whether its fingerprint η_y is found in either of the two associated buckets. If present, the filter reports potential membership; otherwise, the element is certainly not in the set.

Deletions proceed by locating a matching fingerprint and removing it from its bucket. If multiple copies are present, only one is deleted.

Design Considerations and Properties Cuckoo filters exhibit competitive space efficiency, especially when the desired false positive rate ε is low. The fingerprint size f and the load factor α, which measures how full the table is, determine the expected false positive rate and overall capacity. The space usage per item is defined as [32]:

$$\text{bits per item} = \frac{\log_2(1/\varepsilon) + 3}{\alpha}. \tag{2.2}$$

This places Cuckoo filters close to the theoretical lower bound while providing functionality that Bloom filters lack, such as deletions.

An edge case that needs to be considered is that duplicate insertions of elements with identical fingerprints may lead to overflows if the same item is inserted more than $2b$ times, where b denotes the number of slots per bucket. In such cases, both candidate buckets may become saturated with indistinguishable entries, making further insertions impossible without evicting or resolving them externally. Ideally, Cuckoo filters are deployed in scenarios where repeated insertions are rare, and where occasional overflows can be either tolerated or mitigated through higher-level system logic. A practical advantage of Cuckoo filters, particularly relevant for programmable data planes, is that the number of relocation steps during insertion is strictly bounded (typically to a maximum number of 500 relocations [32]).

Cuckoo filters are often favored in environments where operations beyond insertion and querying are needed, while still providing strong guarantees on space and error bounds.

2.5.3 Quotient Filter

The Quotient filter [69, 47, 87] is a dynamic probabilistic data structure that provides approximate set membership with support for deletions. Similar to the Cuckoo filter, it stores fingerprints instead of full keys, but it does so using a different encoding scheme based on the principle of *quotienting* [7]. Its design allows for compact storage, but at the cost of more complex access patterns.

The core idea of quotienting is to split a hash fingerprint $f(x) \in \{0,1\}^p$ into two components: a q-bit *quotient* and an r-bit *remainder*, such that $p = q + r$. Given a hash function $h : U \to \{0, \ldots, 2^p - 1\}$, each element $x \in U$ is mapped to a fingerprint $f(x)$. The quotient $\mathrm{quot}(x) = \lfloor f(x)/2^r \rfloor$ determines the slot index, while the remainder $\mathrm{rem}(x) = f(x) \bmod 2^r$ is stored at that slot or near it [87].

Each slot in the table stores the remainder, along with three metadata bits used for decoding runs and clusters, which enables the compact resolution of collisions via linear probing.

Insertion, Querying, and Deletion To insert an element x, its fingerprint is split, and the remainder is placed in the slot indexed by the quotient. If the slot is occupied by another remainder with the same quotient, the filter performs linear probing to find the next available position. Runs (i.e., sequences of remainders sharing the same quotient) are stored contiguously, and overlapping runs are shifted accordingly to preserve ordering.

To query whether an element y is present, the filter uses the quotient to identify the starting slot, then scans the corresponding run to check whether the associated remainder is present. Deletion follows the same search procedure and removes one matching remainder if found.

Design Considerations and Properties Quotient filters maintain a competitive space efficiency relative to other filters, particularly when deletions are required. Their false positive rate depends on the fingerprint size p, while their memory usage is determined by the number of remainders and the metadata overhead. The number of bits required per element, depending on false positive rate ε and load factor α, is defined as [69]:

$$\text{bits per item} = \frac{(\log_2(1/\varepsilon) + 2.125)}{\alpha}. \tag{2.3}$$

While Quotient filters offer strong space bounds and support for deletions, they come with a notable drawback: lookup operations are inherently more complex due to the need for scanning runs and decoding metadata. In contrast to Bloom and Cuckoo filters, which typically involve only a few fixed-position probes, quotient filters require potentially long and variable-length linear scans. This performance degradation becomes particularly pronounced as occupancy increases, making the structure less suitable for latency-sensitive environments or high-load settings. Also it becomes ill suited for programmable data planes like P4 where variable length linear memory scans are not supported natively.

Chapter 3

Related Work on SYN Flood Protection in Programmable Data Planes

The challenge of mitigating SYN flood attacks within programmable data planes has attracted significant academic attention. Traditional techniques, such as SYN-Cookies, offer stateless protection with perfect precision, but their application in high-speed environments is often limited by computational and memory constraints. As a result, recent research has increasingly focused on offloading defense mechanisms into programmable network hardware to preserve service availability while maintaining high mitigation efficacy.

Effective defenses must balance several critical properties: they must successfully block malicious SYN packets to protect the server, preserve the availability of legitimate connections by minimizing false positives and false negatives, and scale efficiently with the number of concurrent flows while operating within strict processing and memory constraints. These considerations frame the evaluation of prior work and guide the development of new approaches in this thesis.

This chapter surveys existing approaches to SYN flood mitigation in programmable data planes, with a particular focus on SmartNICs and P4-capable hardware. It categorizes defenses by their architectural characteristics, beginning with ML-based systems, followed by rule-based designs, and concluding with SYN-Cookie-based strategies.

3.1 ML-based Approaches

Several works have explored using machine learning to detect and mitigate SYN flood attacks in programmable data planes. Musumeci et al. [55] propose an ML-based detection framework where a P4 switch periodically exports traffic statistics to an external classifier. Features are extracted over sliding traffic windows and fed into a supervised learning model that labels time windows as ”attack” or ”no attack.” Based on this output, the switch can either forward or drop traffic. In terms of mitigation success, the framework can detect large-scale SYN floods based on traffic anomalies. However, its reaction effectively cuts off all traffic during an attack, without distinguishing between benign and malicious connections. Availability preservation is therefore not achieved, as legitimate flows are indiscriminately dropped. While scalability is not the primary concern, as only aggregated statistics are exported, the system introduces control plane latency that delays the reaction to attacks. Furthermore, if attackers use short, high-intensity bursts within the export interval, the system may fail to block malicious packets altogether, exposing a critical weakness in its protection strategy.

Dimolianis et al. [22] present a hybrid architecture combining supervised ML-based traffic classification with data plane filtering via XDP on a SmartNIC. Malicious signatures are identified offline and installed as simplified filtering rules, while ambiguous flows fall back to SYN-Cookie verification. This design improves mitigation success by blocking large volumes of attack traffic early, while fallback mechanisms preserve availability. Scalability is achieved through aggressive signature reduction, which minimizes the SmartNIC's memory footprint. However, the approach does not address the limitations of conventional SYN-Cookie implementations in the data plane as discussed in this thesis.

Miano et al. [52] develop a similar approach, integrating ML-guided filtering into server-side SmartNIC processing. Compact signature filters, derived from minimal feature sets, are installed in the data plane to block obvious attack traffic, while SYN-Cookies validate unmatched packets. Mitigation success is achieved through efficient offloading, with availability protected by protocol-level fallbacks. The ML architecture is explicitly optimized for SmartNIC memory constraints, making it scalable under high churn. Nevertheless, like other ML-based designs, its effectiveness depends on maintaining timely, accurate filter updates—an assumption that may not hold against carefully crafted, indistinguishable attacks.

In summary, ML-based approaches to SYN flood protection offer promising scalability and adaptability but cannot eliminate classification errors and often require SYN-Cookie fallback mechanisms for robustness. Their dependence on training data and control plane coordination introduces unavoidable latency and uncertainty. Notably, even the most promising ML-based defenses ultimately rely on naive implementations of SYN-Cookies, without accounting for their inherent limitations. This underlines the importance of continued research into efficient, lightweight SYN-Cookie architectures for environments requiring maximum assurance — the direction pursued by this thesis. Thus, while ML-based methods may augment SYN-Cookie-based defenses by pre-filtering clearly malicious packets, they fall short as standalone solutions.

3.2 Rule-Based Approaches

Several approaches to SYN flood mitigation have explored rule-based designs that do not rely on the SYN-Cookie principle. Palleri et al. [71] propose a SYN flood defense on the NETFPGA-SUME SmartNIC platform, where protection is triggered once an arbitrary threshold of SYN packets to different ports is exceeded. The detection logic is simplistic and ineffective against sophisticated attackers who randomize target ports to evade threshold detection. Nevertheless, the work provides valuable insights into SmartNIC resource constraints: even this basic approach already consumes around 37% of the FPGA-based SmartNIC's memory, highlighting how even minimalistic security mechanisms can impose significant overheads.

Fei et al. [35] propose a rule-based system for in-network DDoS detection, including SYN floods, using short-lived Bloom filters to track IP addresses based on packet fingerprints. The design assumes that SYN packets from the same botnet share similar signatures, allowing attacks to be detected within brief observation windows, such as 0.25 seconds. While computationally efficient, this approach critically relies on the homogeneity of attack traffic and can be easily evaded by botnets that randomize TCP header fields. Similar to ML-based designs, it serves more as a preliminary filter rather than a complete mitigation solution. The system, implemented on BMv2, demonstrates efficiency but remains effective only against a limited class of SYN flood attacks.

Another rule-based approach is ConPoolUBF [76], which combines updatable Bloom filters with connection pooling to manage TCP handshakes entirely within the data plane. While innovative in offloading handshake management from servers, the system introduces significant overhead: epoch-based Bloom filters must be updated frequently, and complete per-flow TCP state, including sequence and acknowledgment numbers, must be maintained in the data plane. These requirements impose considerable memory and processing burdens, making the approach difficult to scale on resource-constrained programmable devices.

In summary, rule-based defenses offer appealing simplicity but generally fall short in addressing the full spectrum of SYN flood attack strategies. Their reliance on rigid assumptions, coarse thresholds, or high memory overheads undermines the accuracy, scalability, and practical deployability of mitigation. As such, these approaches appear less promising for robust and scalable protection compared to mechanisms based on precise, protocol-aware defenses—such as SYN-Cookie-based solutions, including our proposed design.

3.3 SYN-Cookie Based Approaches

Recent work has adapted SYN-Cookie mechanisms to programmable data planes to defend against SYN flood attacks while minimizing server resource usage. Scholz et al. [77] implement a SYN proxy architecture validated across multiple P4 platforms, including several software targets and FPGA-based SmartNICs. Incoming SYN packets are validated using SYN-Cookies, and only confirmed flows are forwarded to the server. However, after cookie validation, the programmable hardware device must maintain complete per-connection state. This includes TCP 4-tuples and sequence numbers necessary for sequence number translations between server and client to maintain protocol correctness. This mechanism is discussed in further detail in the architecture chapter (Section 5.2) as it's fundamental for CUCKOOGUARD. It is effective at blocking spoofed connections, but this approach incurs substantial memory overhead: its implementation consumes about 35% of the available SmartNIC memory when handling only 100 parallel connections, with memory usage expected to grow further under higher connection loads. This highlights a fundamental limitation—the design does not optimize memory usage, making it unsuitable for typical high-connection or high-churn environments.

To address these challenges, Yao et al. [92] propose SMARTCOOKIE, a split-proxy architecture designed for constrained data planes. A programmable switch performs lightweight SYN-Cookie validation and memory-efficient flow filtering. At the same time, complete TCP sequence number translation and connection state management are delegated to a server-side eBPF agent. By avoiding per-connection state on the switch, SMARTCOOKIE significantly reduces memory pressure while preserving protocol transparency. Nevertheless, the design introduces new trade-offs: coordination between the switch and server adds system complexity, and considerable rates of false positives (7.66% in their evaluation) in the flow filter trigger unnecessary cookie verifications at the server during ACK floods, seriously undermining the intended CPU offload benefits.

In summary, SYN-Cookie-based defenses offer the most promising foundation for precise, scalable, and hardware-friendly SYN flood mitigation. Split-proxy architectures such as SMARTCOOKIE demonstrate that efficient SYN-Cookie offloading is feasible even on constrained data planes. However, they leave key challenges unresolved: significant coordination overhead remains, false positives in flow filtering still cause unnecessary server load, and handling high connection churn under limited memory remains difficult. These limitations motivate the need for a more optimized design. The CUCKOOGUARD architecture introduced in this thesis aims to address these challenges and improve the scalability, accuracy, and efficiency of SYN-Cookie-based flood defenses in programmable data planes.

Chapter 4

Problem Analysis

Distributed Denial-of-Service (DDoS) attacks continue to pose a significant threat to modern network services worldwide. Recent observations highlight Taiwan as a particularly affected region [17]. Notably, during the visit of U.S. House Speaker Nancy Pelosi to Taiwan in August 2022, DDoS activity escalated sharply: the Ministry of Foreign Affairs website received over 8.5 million access requests within a single minute, effectively overwhelming the site's capacity.

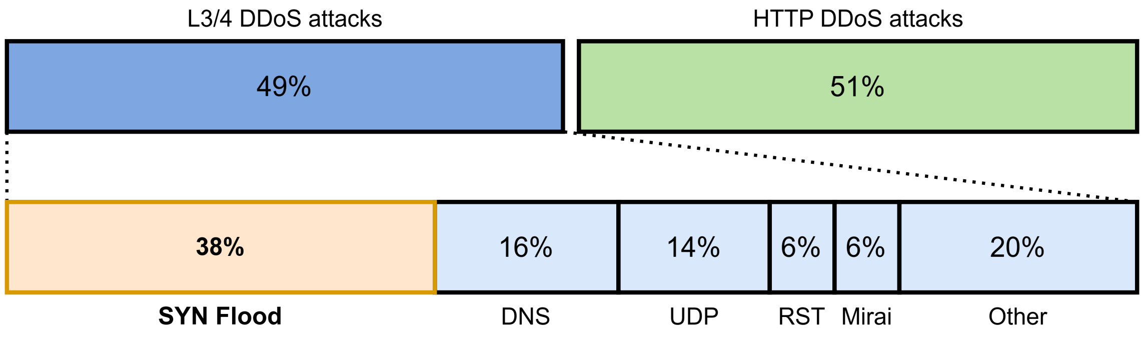

Figure 4.1: Distribution of DDoS attack types in Q4 2024 across Cloudflare's networks [20].

On a global level Taiwan ranked third in Q4 2024, among the most targeted countries for DDoS attacks [20]. According to Cloudflare, 38% of all network-layer DDoS attacks during this period were classified as SYN flood attacks (see Figure 4.1), underlining their significance as a persistent and growing attack vector. Moreover, SYN floods were identified as an emerging threat category, with a 130% increase in frequency compared to the previous quarter, underscoring their escalating impact.SYN floods pose not only a technical threat but also a severe economic risk, with over 90% of mid-size and large enterprises reporting average hourly downtime costs exceeding 300,000 USD—excluding legal or regulatory liabilities [44].

4.1 Problem Statement

SYN flood attacks aim to exhaust server resources by sending large volumes of TCP SYN packets, thereby overwhelming a server's ability to manage legitimate connection requests. The objective is to degrade or completely deny access to services such as websites, APIs, and other online platforms. Such attacks are effective because naive network stack implementations allocate memory or perform computationally expensive operations, such as cookie hashing, immediately upon receiving a SYN packet. When handled directly on server hardware, these operations exhaust memory and CPU resources under high-volume attack conditions. Therefore, SYN flood defenses must focus specifically on new connection attempts, where the authenticity of a sender cannot yet be confirmed.

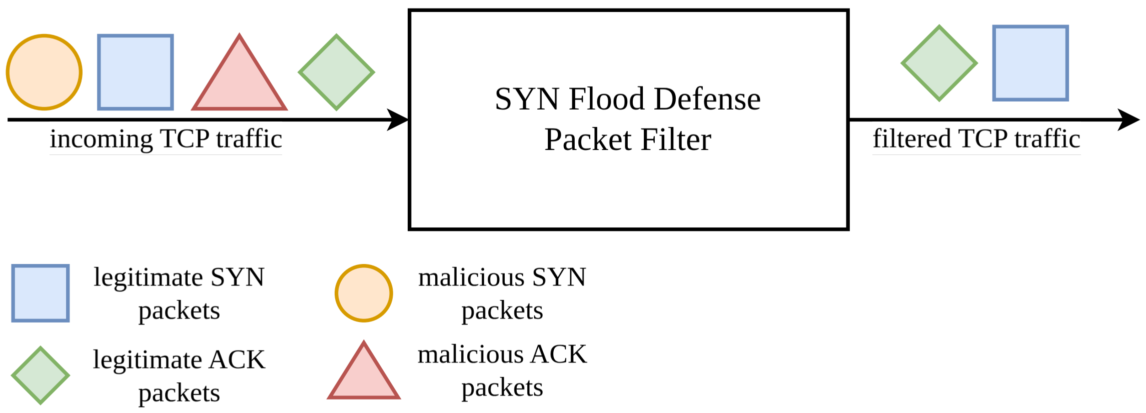

Figure 4.2: Illustration of SYN flood mitigation using flow filtering: malicious SYN and ACK packets are intercepted and discarded, while legitimate traffic is forwarded to the server.

The central difficulty is distinguishing between malicious SYN packets and legitimate ones before the three-way handshake is completed. Consequently, effective defenses must implement fine-grained per-packet validation without prematurely discarding legitimate traffic or keeping costly state for incomplete handshakes. Simply blocking all incoming SYN packets would result in self-inflicted denial-of-service. As illustrated in Figure 4.2, the defense must selectively filter malicious packets while preserving service availability for genuine users.

On top of that, efficiency and scalability are key for protecting critical server resources under attack feasibly. To achieve that, such defenses must maintain precise filtering capabilities at high traffic volumes to ensure uninterrupted service even during massive flooding events. These defenses must go beyond established built-in security mechanisms (Section 2.2), which continue to fail in protecting server resources.

4.2 Background and Context

4.2.1 Attack Types

For this thesis, we distinguish two types of SYN flood attacks, while recognizing that finer-grained classifications exist. First, **randomized spoofed SYN floods** send stateless SYN packets with fake source IP addresses, making attribution difficult. Second, **botnet-based realistic SYN floods** originate from compromised devices using valid IP addresses, often exhibiting more legitimate traffic characteristics. However, both types of attacks exploit the TCP handshake in the same way. For SYN flood protection at the protocol level (i.e., SYN-Cookies), it is not critical to distinguish between attack types, unlike IP address filter-based or ML-based defenses, where traffic classification is more important. As discussed in the related works, while ML-based and statistical filtering methods can complement SYN-Cookie defenses, they must be applied with care, as they can introduce service degradation by erroneously blocking legitimate traffic.

4.2.2 Hardware Accelerating SYN Flood Defenses

SYN flood defenses must operate under extremely high packet rates, which CPU-only solutions cannot keep up with. General-purpose compute hardware is not designed for the high-throughput, low-latency requirements of network packet processing at line rate. Although incremental improvements in CPU architectures continue, the scalability required to process millions of SYN packets per second is fundamentally constrained by the sequential CPU processing paradigm. For that reasons, critical networking tasks have historically been offloaded to specialized hardware, such as network interface cards (NICs), to achieve predictable and sufficient performance. Recent developments in network programmability, particularly with SmartNICs and other P4-programmable devices, now enable flexible, line-rate processing without burdening the host CPU [23] (see Section 2.4.1). In this thesis, we leverage these hardware accelerated programmable data planes to deploy our SYN flood defense, taking advantage of their ability to filter and validate packets at wire speed independently. Nevertheless, we recognize that the data plane is a shared environment hosting multiple network functions, and we consider how our defense interacts with other co-resident applications.

Why P4? P4 is a portable domain-specific language designed for programming network functions (see Section 2.3) on hardware accelerators such as SmartNICs and programmable switches. Given our focus on hardware-based SYN flood defenses, P4 is a natural choice: it enables precise, low-latency packet validation at line rate while maintaining some level of portability across various hardware targets. A SYN flood defense implemented in P4 can operate alongside other network functions as part of a broader in-data-plane security suite. Moreover, the use of P4 ensures that the academic contributions of this thesis remain applicable to a wide range of real-world deployment scenarios, beyond a single proprietary platform.

4.2.3 FPGA-based SmartNICs for Data Plane Acceleration

FPGA-based SmartNICs provide an attractive platform for deploying efficient and flexible SYN flood defenses. Unlike CPU-bound solutions, FPGAs provide true hardware parallelism, enabling line-rate packet processing without introducing significant latency or burdening the server. Architecturally, FPGAs combine reconfigurable logic blocks, fast on-chip memories, and highly efficient memory access patterns [39, 84, 1], making them well-suited for precise, fine-grained network functions, such as SYN-Cookie validation. From a performance-per-watt perspective, FPGA-based SmartNICs achieve higher efficiency than CPUs while remaining reprogrammable, a balance that fixed-function ASICs cannot offer [30, 33].

Recent research increasingly envisions FPGA-based SmartNICs as hosting multiple, dynamically deployed network functions rather than being dedicated to a single task. Lin et al. [50] and Su et al. [79] propose architectures where network functions are flexibly virtualized across individual or interconnected SmartNICs, abstracting hardware boundaries and enabling dynamic, multi-tenant deployments. Since FPGA reprogramming is slow and disruptive, and modern advances such as Partial Reconfiguration still have inherent limitations [14], keeping defenses always loaded and activating them as needed avoids downtime due to reconfiguration. This approach aligns with broader trends in cloud infrastructure, where functions coexist, and isolation and scalability are critical. Byways [38] further emphasizes the need for secure and efficient deployment of network functions due to the requirement for them to remain functional in multi-tenant environments. This motivates defenses that seamlessly integrate into evolving multi-function SmartNIC environments. Therefore, this thesis positions FPGA-based SmartNICs as a foundational platform for scalable, resource-efficient SYN flood protection, integrated into the broader trend of dynamic, multi-function network service deployments.

Memory as a Limiting Factor in Data Planes

Memory constraints are a critical challenge for deploying SYN flood defenses in programmable data planes. FPGA-based SmartNICs typically offer only a few megabytes of fast on-chip memory, which must accommodate all deployed network functions, including flow tracking, SYN state handling, and packet classification of SYN flood defenses. Studies such as [22] and [77] demonstrate that even relatively simple network functions can consume more than a third of available SmartNIC on-chip memory, highlighting the importance of resource-efficient designs. Similarly, Miano et al. [52] identify limited memory resources as a key obstacle, motivating the use of distributed architectures that offload functionality beyond the SmartNIC itself.

Other work, such as Balla et al. [9], further illustrates these limitations: their FPGA-based DDoS mitigation system, implemented using P4, is constrained to a maximum of 250,000 entries in its firewall rule table, setting a hard limit on the number of malicious IP addresses that can be actively blocked. Even when applying selective filtering, such as targeting high-volume sources, the inherent hardware limitations significantly constrain scalability during large-scale spoofed attack scenarios—particularly when the firewall is deployed alongside other essential network functions.

Tight memory budgets are not exclusive to FPGA-based SmartNICs; programmable switches often operate under similar constraints, with total available memory sometimes limited to only a few tens of megabytes. SMARTCOOKIE [72] explicitly acknowledges these limitations and employs a distributed architecture where memory-intensive tasks are offloaded to a server-side agent.

Furthermore, the aforementioned trend toward multifunctional deployments, where multiple network functions coexist on the same SmartNIC, imposes additional pressure to minimize both the algorithmic complexity and memory footprint of each function. Solutions that consume excessive memory or logic resources not only limit scalability but also interfere with the deployment of other critical services sharing the device. Consequently, we view computational and memory resource consumption as a primary optimization goal.

4.3 Case Study

This section examines two common types of Internet-facing servers—REST API servers and streaming servers—and derives additional requirements for SYN flood defenses based on their typical traffic patterns and operational characteristics.

4.3.1 Protecting REST API Servers

REST API servers provide lightweight, stateless services over HTTP(S), often serving mobile apps, web clients, or microservices. They handle a large number of short-lived TCP connections, with clients frequently issuing rapid, bursty connection attempts. Some systems also maintain huge numbers of idle connections via HTTP keep-alive to reduce latency when setting up. SYN flood defenses for REST API servers must tolerate high connection churn without exhausting memory or introducing connection setup delays. Line-rate operation and memory-efficient tracking are critical to avoid misclassifying legitimate connection bursts.

4.3.2 Protecting Streaming Servers

Streaming servers typically manage fewer but longer-lived TCP connections, maintaining continuous sessions for video, audio, or data delivery. While connection surges may occur during major events, the overall churn rate remains low compared to REST APIs. Here, SYN flood defenses must filter malicious SYNs without disrupting active, long-lived sessions. Robust per-connection tracking over extended periods is essential to preserve service quality under attack.

4.4 Requirements for a SYN Flood Defense in Programmable Data Planes

Based on the preceding analysis, we define a set of key requirements for SYN flood defenses deployed in hardware-accelerated programmable data planes. Each requirement addresses a distinct aspect of scalability, performance, and deployability critical to effective, real-world protection.

- **R1 – Low Memory Consumption and High Connection Capacity (MEM)**
  The defense must minimize its memory footprint to fit within the limited on-chip memory resources available on SmartNICs, while supporting large numbers of simultaneously active legitimate connections.

- **R2 – Low Latency (LAT)**
  The defense must add minimal per-packet processing delay to preserve end-to-end application performance.

- **R3 – High Throughput (THR)**
  The system must handle high volumes of SYN packets at line rate without introducing processing bottlenecks.

- **R4 – Transparency (TRP)**
  The defense must be protocol-transparent to both clients and servers, preserving normal TCP handshake and traffic behavior without introducing anomalies.

- **R5 – Server Offloading (OFF)**
  The solution must offload connection validation from the server, preventing server CPU and memory exhaustion under attack.
- **R6 – High Churn Tolerance (CHN)**
  The defense must remain effective when legitimate connection attempts occur at high rates, ensuring rapid tracking and validation of short-lived connections.
- **R7 – Connection Stability and Long-Term Robustness (STA)**
  The defense must maintain the stability of long-lived connections and preserve consistent behavior and performance over time, even under prolonged attack conditions.

While not all of these requirements have been explicitly formalized in prior work, several similar or partially overlapping design goals have been identified, reinforcing the validity of the principles outlined here. Scholz et al. [77] highlight the importance of high throughput and memory efficiency, describing **R3 (THR)** as the ability to operate at or beyond line rate, and discussing **R1 (MEM)** in the context of SYN cache constraints that limit scalability under high connection volumes. SMARTCOOKIE [72] further emphasizes service quality and protocol transparency by targeting low end-to-end latency and avoiding disruptions to standard TCP behavior, directly addressing **R2 (LAT)** and **R4 (TRP)**. Importantly, both approaches share the overarching objective of reducing server-side processing burden, thereby aligning with **R5 (OFF)** as a central design goal.

R6 (CHN) and **R7 (STA)** are explicitly introduced in this work, as they address critical yet previously underexplored challenges in realistic deployment scenarios—namely, the need for defenses to remain effective under high connection churn and to preserve connection stability during prolonged deployment periods.

While existing defenses provide valuable groundwork, this thesis generalizes and formalizes these insights into a unified and comprehensive set of requirements, summarized in Table 4.1, tailored explicitly for deployment in hardware-accelerated programmable data planes.

4.5 Comparison with Existing Solutions.

Based on the requirements defined in Section 4.4, the existing SYN flood defense solutions introduced in Chapter 3 were evaluated and compared to CUCKOOGUARD, the new architecture introduced in this thesis. The outcome of this analysis is summarized in Table 4.2. While individual approaches address subsets of the requirements, none of them fulfill all demands. CUCKOOGUARD is specifically designed to close these gaps, with the following chapter detailing how it achieves full compliance across all criteria.

Table 4.1: Architectural requirements for a SYN flood defense.

Requirement	**Description**
R1 (MEM)	Low memory consumption and high connection capacity
R2 (LAT)	Low latency
R3 (THR)	High throughput
R4 (TRP)	Protocol transparency
R5 (OFF)	Server offloading
R6 (CHN)	High churn tolerance
R7 (STA)	Connection stability and long-term robustness

Table 4.2: Comparison of SYN flood defense solutions against the requirements.

Approach	**MEM**	**LAT**	**THR**	**TRP**	**OFF**	**CHN**	**STA**
ML-based [22, 52]	×	✓	✓	○	○	×	×
Palleri et al. [71]	×	✓	✓	×	✓	○	×
Fei et al. [35]	✓	×	✓	×	✓	○	×
ConPoolUBF [76]	×	✓	○	✓	✓	×	✓
SYN Proxy [77]	×	✓	✓	✓	✓	×	✓
SmartCookie [92]	✓	✓	○	✓	○	×	✓
CuckooGuard	✓	✓	✓	✓	✓	✓	✓

✓*: fully satisfies,* ○*: partially satisfies,* ×*: does not satisfy.*

Chapter 5

CuckooGuard Architecture

The design of a scalable and precise SYN flood mitigation system in programmable data planes must be grounded in a careful analysis of trade-offs between performance, resource efficiency, and architectural feasibility. In the previous chapter, a set of concrete requirements (**R1–R7**) was defined to characterize the properties expected of any such system.

This chapter presents the CuckooGuard architecture, which is developed in direct response to these requirements. The central idea is to perform efficient connection validation at the data plane level using programmable hardware. As one of its core components, the system uses an approximate flow filter to track verified connections within the SmartNIC. Given the centrality of this component, the chapter begins by evaluating alternative filter designs and motivating the choice of a Cuckoo filter. The remainder of the chapter then introduces the full architectural pipeline, including its deployment strategy and inter-component coordination mechanisms. Finally, we discuss how the design is expected to behave under realistic attack and traffic scenarios.

5.1 Flow Filter Design Rationale

At the heart of CuckooGuard lies a dynamic, memory-efficient flow filter that tracks trusted TCP flows. This flow filter is a crucial decision point, enabling stateful SYN packet handling while preserving server transparency and, most importantly, memory efficiency. Given the strict memory and processing constraints of programmable network devices such as SmartNICs, the choice of filter data structure significantly impacts system feasibility and performance.

This section compares three widely used approximate set-membership data structures—Bloom, Quotient, and Cuckoo filters—based on requirements derived from the overall system goals. The algorithmic principles of these filters are introduced in Section 2.5, and are therefore not repeated here; instead, we focus on evaluating their practical suitability for deployment in a constrained programmable data plane.

5.1.1 Filter Requirements

To guide the filter selection process, we identify the following specific requirements, derived from the general architecture goals **R1 (MEM)**, **R3 (THR)**, and **R7 (STA)**:

F1 – Memory Efficiency: The filter must use as little memory per tracked TCP flow as possible while maintaining an acceptable false positive rate.

F2 – Precision: Faulty filtering decisions caused by false positives must be low and stable across varying traffic loads.

F3 – Performance: All operations (insertion, lookup, and deletion) must be executable with minimal latency and predictable runtime behavior in P4 data plane applications.

F4 – Deletion Support: The filter must support element removal explicitly or via a timeout mechanism to handle high churn and functioning over extended periods, preventing filter saturation.

The design context assumes the tracking of full 4-tuples (source/destination IP and port), which form a canonical identifier for established TCP connections. Traditional exact data structures such as hash tables, maps, or match-action tables—commonly employed in P4 programs—are ill-suited for data plane deployment in this context, as their per flow memory requirements are high and typically need to grow unboundedly with the number of tracked connections, conflicting with the strict memory constraints of programmable hardware. Approximate filters provide an attractive alternative, offering compact, constant-size representations at the cost of probabilistic membership testing.

In what follows, we evaluate the three filter candidates based on the criteria **F1–F4**, based on theoretical analysis.

5.1.2 F1 – Memory Efficiency

Given the stringent memory constraints of programmable data planes, the filter component in CUCKOOGUARD must represent each tracked connection as compactly as possible. This requirement, derived from design goal **R1 (MEM)**, directly impacts scalability under high connection volumes.

Figure 5.1 shows the theoretical memory cost per element for Bloom, Cuckoo, and Quotient filters across varying false positive rates. The x-axis is logarithmically scaled

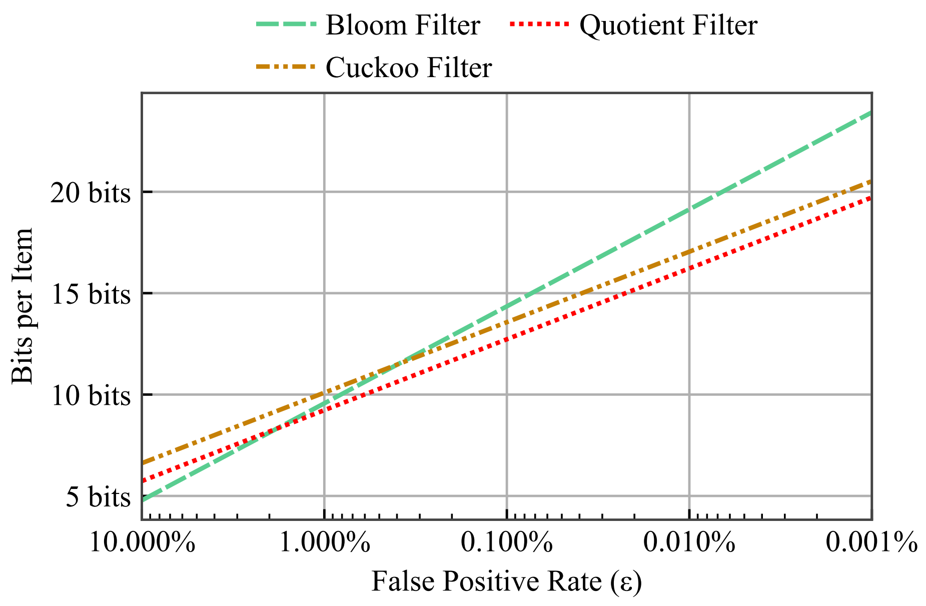

Figure 5.1: Bits per filter item required to achieve different false positive rates. Cuckoo and Quotient filters outperform Bloom filters in the low false positive region.

to highlight the low false positive range; the y-axis shows the number of bits required per tracked item. These results are derived from the formulas in Section 2.5 (Equations 2.1–2.3).

At moderate false positive rates (e.g., 1%), Bloom filters are highly memory-efficient, requiring around 9.6 bits per item. However, as the target false positive rate is further reduced ($\leq 0.1\%$), their bits-per-item requirement increases more rapidly than that of alternative filter types, limiting their suitability for applications demanding very low false positive rates. In this range, both Cuckoo and Quotient filters become more space-efficient. Quotient filters consistently use 0.875 fewer bits per item than Cuckoo filters, which is also observable in the formulas and can be traced back to the differences in encoding and querying algorithms.

Since CuckooGuard targets a high-precision operating point ($\ll 0.1\%$ false positive rate), Cuckoo and Quotient filters are significantly more memory-efficient than Bloom filters and are therefore preferred in this category.

5.1.3 F2 – Precision

In addition to memory efficiency, the CuckooGuard filter must offer highly precise membership checks to avoid forwarding unauthenticated packets to the server, violating the offload objective **R5 (OFF)** (see Section 5.2).

Figure 5.1 again highlights how filter designs behave in terms of achievable precision. At any fixed memory budget, Cuckoo and Quotient filters outperform Bloom filters in the sub-0.1% false positive range targeted by CUCKOOGUARD. Their ability to maintain low error rates under tight memory constraints makes them well-suited for security-sensitive deployments.

Cuckoo and Quotient filters jointly dominate in the high-precision regime and are thus also preferred in terms of accuracy.

5.1.4 F3 – Performance

To ensure viability for deployment in programmable data planes, all core filter operations—insertions, lookups, and deletions—must exhibit predictable, low-latency behavior. Table 5.1 summarizes the time complexities of these operations for Bloom, Cuckoo, and Quotient filters, based on the literature [32, 10, 47, 80].

Table 5.1: Runtime complexity comparison of Bloom, Cuckoo, and Quotient filters. k denotes the number of hash functions; α is the filter's load factor.

Filter Type	**Insertion**	**Query**	**Deletion**
Bloom Filter	$\mathcal{O}(k)$	$\mathcal{O}(k)$	Not supported
Cuckoo Filter	$\mathcal{O}(1)$ amortized, $\mathcal{O}(500)$ worst-case	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Quotient Filter	$\mathcal{O}(\alpha)$	$\mathcal{O}(\alpha)$	$\mathcal{O}(\alpha)$

Bloom filters require multiple hash computations per query or insertion, typically with $k = 6$ to 10, resulting in k memory accesses and hash calculations per operation. This introduces variability in performance depending on k. Additionally, Bloom filters do not support explicit deletion of entries. Their primary advantage lies in implementation simplicity, which explains their widespread adoption in P4-based applications. However, many such implementations choose suboptimal values for k, as repeated hash computations and memory accesses can degrade performance. As a result, the achievable false positive rate is often worse than theoretically possible with an optimal k.

Quotient filters, while compact, depend on scanning runs of remainders, leading to variable and potentially high latencies, especially at high load factors. This poses a challenge for P4-based implementations, which lack native support for unbounded memory access patterns or loops. In particular, the run-scanning mechanism of quotient filters is highly inefficient to implement in P4, making them impractical for data plane environments.

In contrast, Cuckoo filters support constant-time lookups and deletions—each requiring at most two independent memory accesses and hash calculations—and offer amortized constant-time insertions, making them well-suited for high-speed packet processing. While worst-case insertions may require multiple relocation steps—up to $\mathcal{O}(500)$ in extreme scenarios—these cases are rare under typical load factors ($\alpha \leq 95\%$). Empirical results in Chapter 7 validate that insertion overhead remains manageable.

Due to their constant-time operations and practical support for all required functions, Cuckoo filters are the clear winner regarding runtime performance in data plane environments.

5.1.5 F4 – Deletion Support

In high-churn network environments, especially those involving short-lived REST API connections, the ability to delete stale entries from the filter is essential to prevent filter saturation and ensure continuous availability. Table 5.2 summarizes the deletion capabilities of the evaluated filters.

Table 5.2: Deletion support across filter types.

Filter Type	**Deletion Support**
Bloom Filter	Not supported (requires expiration)
Cuckoo Filter	Supported (explicit deletion)
Quotient Filter	Supported (explicit deletion)

Bloom filters fundamentally lack support for deletion due to their bit-sharing nature. While expiration mechanisms (e.g., epoch-based resets or time-decaying variants) can approximate deletion, they compromise precision and increase implementation complexity and the memory footprint. In contrast, both Cuckoo and Quotient filters support explicit element removal. This makes them more suitable for data plane deployment, where precise, memory-stable operation is critical.

5.1.6 Justification of Cuckoo Filters as Final Design Choice

To consolidate the findings from the previous sections, Table 5.3 summarizes the performance of each filter type across the four key evaluation criteria **F1–F4**. The table serves as the basis for selecting the best filter data structure as the core component of the flow filter in the CuckooGuard architecture. As detailed in the next section (see Section 5.2), this filter plays a central role in enabling connection-level DDoS mitigation directly within the programmable data plane.

After careful consideration, Cuckoo filters are selected because they offer the best overall balance across all criteria. In particular, their superior memory–precision trade-off compared to Bloom filters makes them suitable for precise flow tracking under memory constraints. Furthermore, their support for (amortized) constant-time insertions, lookups,

Table 5.3: Comparison of candidate filter types across filter evaluation criteria **F1–F4**.

Filter Type	**F1: Memory**	**F2: Precision**	**F3: Performance**	**F4: Deletion**
Bloom Filter	○	○	✓	×
Quotient Filter	✓	✓	×	✓
Cuckoo Filter	✓	✓	✓	✓

✓*: fully satisfies,* ○*: partially satisfies,* ×*: does not satisfy.*

and deletions makes them algorithmically well-suited for programmable data planes such as P4, where bounded processing steps are essential.

Critically, explicitly removing individual entries enables efficient handling of high-churn traffic as the key feature of CUCKOOGUARD. This marks a significant improvement over Bloom-filter-based approaches, which are often utilized due to implementation simplicity.

While Cuckoo filters provide powerful capabilities, they also present implementation challenges, as will be discussed in Chapter 6. Their compelling combination of precision, memory efficiency, and dynamic operation makes them a strong foundation for CUCKOOGUARD's programmable SYN flood defense.

5.2 Architecture Components

Building on the split-proxy model introduced previously (see Section 3.3), the CUCKOOGUARD architecture proposes a refined, high-performance design tailored for SmartNIC deployment. The primary objective of CUCKOOGUARD is to realize the full functionality of a SYN-Proxy [77], while addressing its traditional performance and deployment limitations. To facilitate a clear understanding of the protocol-level interactions that underpin this functionality, Figure 5.2 illustrates the precise behavior of a SYN-Proxy during the TCP three-way handshake and regular TCP traffic forwarding. This depiction highlights how the proxy intercepts and validates incoming SYN packets using SYN-Cookies, effectively completing a handshake with the client on behalf of the server before initiating a separate, backend handshake with the protected server only after validation succeeds. After that, regular TCP traffic is forwarded with translated sequence numbers.

To address the unique memory and performance limitations of SmartNICs and server hardware, respectively, the architecture distributes this connection-handling logic across two cooperative components to leverage each component's advantages:

- The **SmartNIC Agent** performs line-rate packet filtering and TCP connection validation in the data plane.

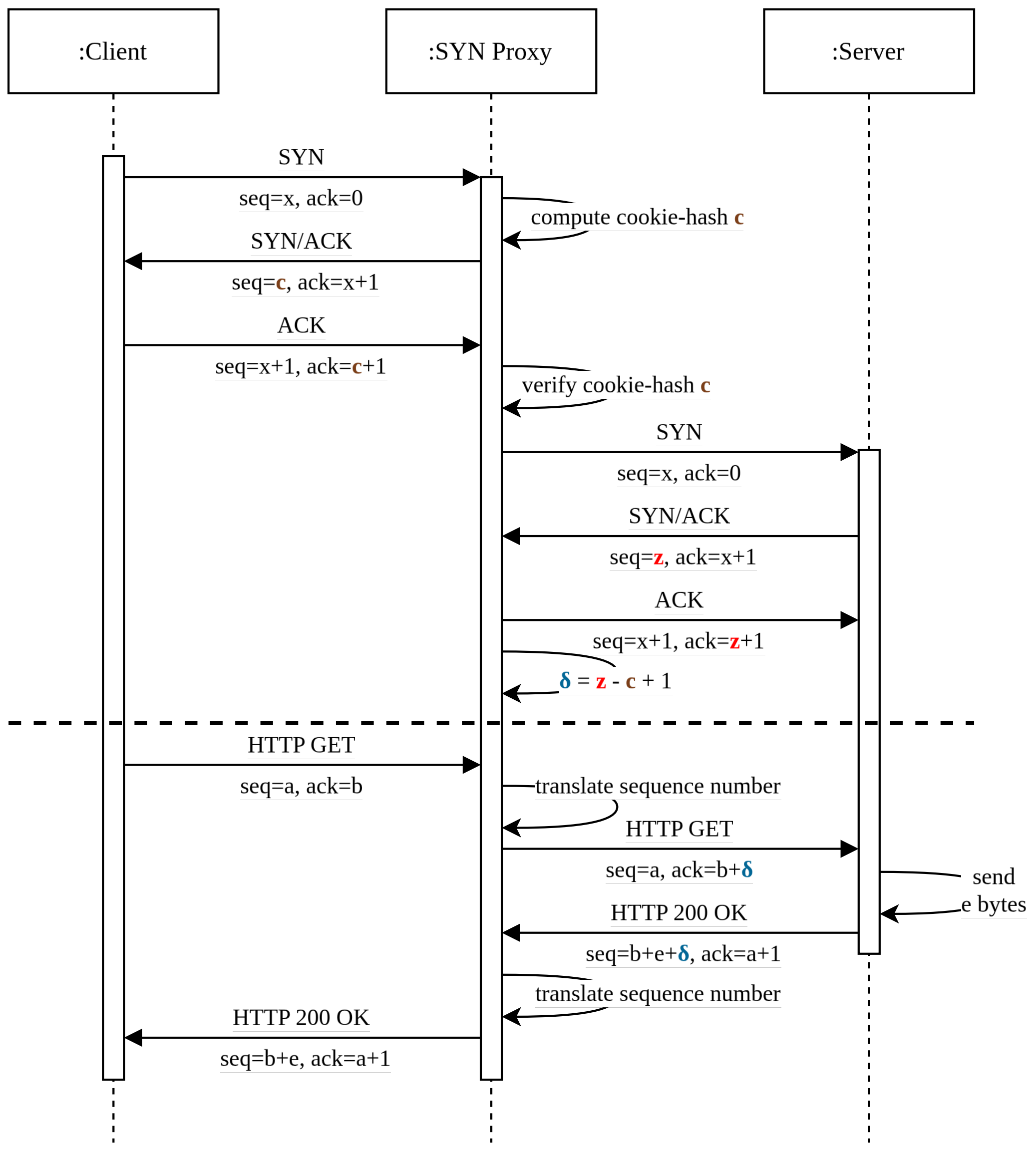

Figure 5.2: Protocol-level visualization of SYN-Proxy operations.

- The **Server Agent** executes lightweight control tasks and keeps detailed connection state within the host's kernel space.

This division eliminates the need for memory-hungry per-connection state on the Smart-NIC and enables transparent, protocol-compliant connection handling. Figure 5.3 provides a detailed overview of the architecture and the interactions between its components.

In contrast to prior designs that place the hardware-accelerated defense component centrally within the network, CUCKOOGUARD is deployed directly at the server edge. This choice aligns with the typical deployment model of SmartNICs, which are designed to provide line-rate packet processing for individual hosts rather than entire networks. Placing the defense mechanism at the server edge also brings important security advantages: it protects the server from external SYN flood attacks and malicious traffic within the same network, such as compromised tenant workloads or misbehaving internal services.

This close coupling between the SmartNIC and its associated server also simplifies coordination. Unlike distributed solutions that must manage state across multiple locations, the CuckooGuard architecture operates with a one-to-one mapping between the SmartNIC Agent and the Server Agent.

5.2.1 SmartNIC Agent

The SmartNIC Agent is deployed entirely in the SmartNIC's data plane, intercepting all incoming TCP packets. Its primary task is to perform SYN-Cookie-based authentication without maintaining any explicit per-flow state. Once a connection is verified via SYN-Cookies, the SmartNIC Agent tags the final ACK packet of the TCP handshake and forwards it to the Server Agent. Upon receiving further confirmation from the Server Agent, the SmartNIC inserts the TCP connection's flow info into its flow filter. To track these verified benign flows efficiently, the agent implements an approximate flow filter as a Cuckoo filter. Utilizing an approximate filter significantly reduces memory usage—a significant advantage given the tight resource constraints of SmartNICs. From that point onward, arriving packets belonging to this known connection bypass further validation and are simply forwarded.

Although the Cuckoo filter is probabilistic and may yield false positives, occasionally allowing unverified ACKs to reach the server, all SYN packets are reliably intercepted. This makes the architecture completely immune to SYN Floods, limited only by the line-rate specifications and hash calculations of the SmartNIC. To ensure security and correctness in the case of false positives, the Server Agent revalidates any unverified ACKs via a fallback cookie check. Finally, when a connection terminates, the Server Agent recognizes that and signals the SmartNIC Agent to remove the corresponding entry from the flow filter, preserving accuracy and freeing memory for new flows.

5.2.2 Server Agent

The Server Agent is implemented using two coordinated eBPF programs: one at the ingress (XDP [83]) and one at the egress (TC [27]) of the kernel networking stack. Operating entirely in-kernel, these programs avoid costly user-space context switches and execute with minimal overhead. Both programs share access to a BPF map [82], which stores the state of currently active connections and the sequence number offsets needed to maintain protocol correctness. The Server Agent fulfills two key responsibilities:

Once a connection request passes cookie verification at the SmartNIC, the SmartNIC Agent adds a setup tag to the final ACK of the TCP handshake. Then, it is forwarded to the Server Agent, initiating a second handshake between the Server Agent and the actual

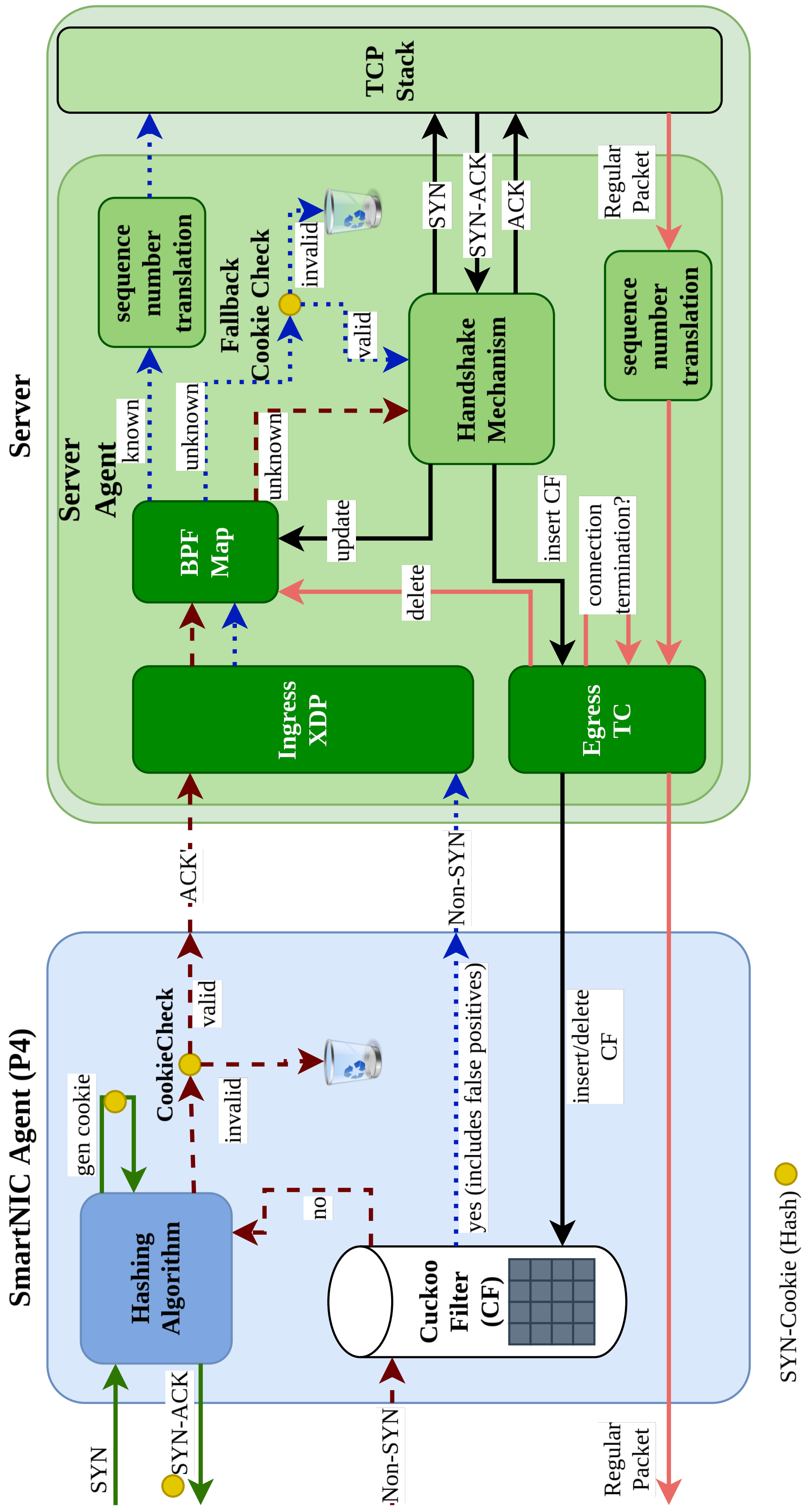

Figure 5.3: Deployment of CuckooGuard, illustrating the separation of handshake validation and stateful connection management between the SmartNIC and the server.

TCP server. Due to the server selecting a new initial sequence number independently (as required by the TCP protocol), the Server Agent computes and stores the sequence number differential in the BPF map. This allows future packets of the connection to be correctly translated and relayed. After the handshake is completed, the Server Agent instructs the SmartNIC to register the connection in the Cuckoo filter. When the connection terminates (see Chapter 6 for details on connection termination detection), the Server Agent likewise signals the SmartNIC to remove the corresponding flow entry from the Cuckoo filter.

If an ACK packet that does not correspond to an existing connection and is not tagged as part of a verified handshake reaches the Server Agent, the agent performs a fallback cookie check. If successful, the handshake process is initiated; otherwise, the packet is dropped. Due to cookie-hashing, this fallback path is computationally expensive, and thus, minimizing its invocation is a primary optimization goal (**R5 – OFF**). Reducing the false positive rate of the SmartNIC's Cuckoo filter is, therefore, critical and is the central focus of our evaluation in Chapter 7.

5.3 Operations

The operation of the proposed architecture is described with three representative traffic scenarios:

5.3.1 Scenario – Benign Client Connection

A legitimate client sends a SYN, and the SmartNIC Agent replies with a SYN-ACK containing a cookie. Upon receiving an ACK from the client including the cookie hash, the SmartNIC Agent verifies the cookie and forwards a tagged packet to the Server Agent. The Server Agent then completes the handshake with the server, calculates sequence number offsets, saves connection information in the BPF map, and finally messages the SmartNIC agent to update the Cuckoo filter to incorporate the newly established connection. Subsequent packets pass the Cuckoo filter at the SmartNIC Agent, and the Server Agent translates the sequence numbers and forwards the packets to the server. Upon connection teardown, the Server Agent signals the SmartNIC Agent to remove the connection's flow entry from the Cuckoo filter.

5.3.2 Scenario – SYN Flood

An attacker initiates a SYN flood by transmitting a high volume of SYN packets. The server, however, remains fully protected, as the SmartNIC intercepts all incoming SYNs and responds with SYN-ACKs embedding a cookie value, without maintaining any per-connection state. Until the handshake is successfully completed by the client, no connection state is created. Crucially, unsolicited SYN packets from external sources are never forwarded to the server, ensuring that the server remains isolated from incomplete or spoofed connection attempts.

5.3.3 Scenario – ACK Flood

Malicious ACK packets attempt to bypass cookie checks. Some may pass the Cuckoo filter due to false positives and reach the Server Agent, which performs a fallback cookie check that eliminates all malicious ACKs. The processing cost of the cookie hashing at the Server Agent remains bounded by the Cuckoo filter's false positive rate.

5.4 Qualitative Discussion on System Requirements

To assess the adequacy of the CuckooGuard architecture, we revisit the system-level requirements **R1–R7** defined in Chapter 4. Each requirement is addressed below in the context of the architectural design choices presented in this chapter.

R1: Low Memory Consumption and High Connection Capacity (MEM). The CuckooGuard architecture achieves memory efficiency using a split-proxy design that separates precise connection tracking and flow filtering. The SmartNIC maintains no explicit per-flow state and relies on a compact Cuckoo filter to filter out most malicious packets. This filter consumes significantly less memory than traditional connection tables, enabling high connection capacity. A detailed per-flow state—consisting only of a TCP sequence number delta, connection state, and connection identifier—is stored in a BPF map on the server and is not subject to SmartNIC memory limitations.

R2: Low Latency (LAT). All latency-critical operations are executed in the SmartNIC's data plane or within eBPF programs in the kernel, both optimized for minimal delay. The defense integrates seamlessly with the standard TCP handshake process, introducing no user-visible round-trip delays.

R3: High Throughput (THR). The design ensures line-rate throughput through light-weight processing: In the SmartNIC Agent, SYN packets are verified via a hardware-accelerated cookie mechanism, and subsequent ACK packets are processed using efficient filter lookups. For verified flows, packets pass through without additional computation; only the sequence number needs to be adjusted by the Server Agent.

R4: Transparency (TRP). CuckooGuard functions as a transparent SYN proxy that preserves TCP semantics. Client and server observe a normal three-way handshake. The Server Agent operates entirely within the kernel and does not interfere with application-layer behavior, preserving out-of-the-box compatibility with unmodified software stacks.

R5: Server Offloading (OFF). Under attack conditions, the architecture effectively shields the server from connection attempts. SYN packets are intercepted and handled at the SmartNIC; only verified connections are forwarded. While false positives during ACK floods may occasionally lead to fallback checks, the Cuckoo filter's precision minimizes this overhead. Thus, CPU and memory resources on the host remain largely unaffected even under heavy SYN and ACK floods.

R6: High Churn Tolerance (CHN). The Cuckoo filter supports explicit deletions and maintains high space efficiency, enabling it to track rapidly changing sets of active connections. Unlike time-decaying Bloom filters, which may saturate under churn, the filter dynamically adapts to connection turnover. This ensures correct behavior under high connection setup and teardown rates.

R7: Connection Stability and Long-Term Robustness (STA). The filter maintains only active flows because flow entries are explicitly inserted and removed based on connection state. This dynamic behavior ensures that long-lived connections are not evicted prematurely and that stale entries do not accumulate. As a result, CuckooGuard remains robust and consistent even during extended deployment and attack periods.

In summary, the CuckooGuard architecture satisfies all seven design requirements through careful decomposition of responsibilities, SmartNIC–kernel coordination, and efficient data structures. This validation supports its suitability as a scalable and practical SYN flood defense mechanism for SmartNIC and other programmable data plane environments.

Chapter 6

Implementation

To validate the feasibility of the CuckooGuard architecture and compare different flow filter designs experimentally, we implement two variants of SYN flood defenses in P4: CuckooGuard using a Cuckoo Filter and the baseline SmartCookie using a Bloom filter. Both use the split-proxy model introduced in Chapter 5.2. The implementations are built to run on the *Behavioral Model v2* (BMv2) [63] (see Section 2.4.2), a widely used software switch that serves as a reference implementation for the P4 language.

BMv2 lacks support for secure hash functions, so all variants use CRC-based hashing [36] for SYN-Cookie generation. While insecure for deployment, this suffices for functional evaluation of the architecture and for comparing the operational behavior of the filter structures. Cookie security is not the focus of this evaluation. We refer to prior work such as [92, 78] for implementing the necessary cryptographic hash functions in programmable data planes.

BMv2 provides a practical and faithful environment for expressing and debugging idiomatic P4 code. Its support for tables, actions, control flow, and register arrays allows for prototyping packet-processing pipelines that closely mirror real deployments. However, as a software-based switch, BMv2 cannot replicate the performance characteristics of hardware P4 targets. Therefore, while our implementations offer insights into correctness and filter effectiveness, absolute throughput or latency measurements do not represent hardware performance and are hence omitted. An overview of P4 itself is provided in Section 2.3.1.

The implementation's source code is available online on GitHub [26] and GitLab [25].

Goal of the Implementations The primary goal of the implementation effort is to realize and compare two filter-based connection tracking mechanisms within the CuckooGuard architecture. By expressing both Bloom and Cuckoo filters in P4, we are able to assess their practicality under the constraints imposed by data plane programming.

Unlike general-purpose languages, P4 restricts developers to a constrained, pipeline-oriented execution model with no loops, recursion, or sequential memory access. This demands algorithmic adaptations to suit the language's capabilities and the underlying target. For each implementation, we highlight key design decisions, limitations, and workarounds that emerged during the porting process.

In the sections that follow, we detail the structure and implementation logic of both filter designs, with particular focus on how the Cuckoo filter maps to P4 programming constructs such as register arrays, match-action tables, and metadata fields.

6.1 CuckooGuard

6.1.1 SmartNIC Agent: P4

The SmartNIC Agent tracks verified TCP connections using a memory-efficient Cuckoo filter [32], implemented entirely in P4 using BMv2. Cuckoo filters offer explicit deletion, insertion, and low false positive rates.

Cuckoo Filter Configuration

The filter is implemented as a fixed-length register array, organized into B buckets, each holding b fingerprints. A fingerprint η is a compact representation of a connection's 4-tuple. To optimize for a low false positive rate ε, the size of each fingerprint f and the total number of buckets B are derived from the available memory m in bits and expected number of active benign flows n, with α denoting the load factor:

$$f = \left\lfloor \frac{m \cdot \alpha}{n} \right\rfloor \tag{6.1}$$

$$B = \left\lfloor \frac{m}{f \cdot b} \right\rfloor \tag{6.2}$$

These parameters are chosen optimally according to standard values established in literature [32].

Fingerprint and Index Calculation

To compute the fingerprint, the 4-tuple of the connection is hashed using CRC32. As a hash function, we use CRC32, which offers good distribution characteristics despite lacking full bijectivity for our input sizes [75]. Listing 6.1 shows the fingerprint generation in P4.

```
const bit<6> fp_bit_index = 32 - FINGERPRINT_SIZE;

action cuckoo_calc_fingerprint() {
    bit<32> index_hash;
    hash(index_hash, HashAlgorithm.crc32, (bit<32>)0,
        {hdr.ipv4.src_addr, hdr.ipv4.dst_addr,
        hdr.tcp.src_port, hdr.tcp.dst_port},
        (bit<32>)4294967295);

    meta.cuckoo_fingerprint = index_hash[31:fp_bit_index];
}
```

Listing 6.1: Fingerprint calculation in P4 using CRC32

After the fingerprint is computed, two candidate bucket indices are derived. The first index is obtained by hashing the 4-tuple bound to the number of buckets B, and the second is computed via an XOR between the first index and a fingerprint-derived hash. Since BMv2 does not support modulo operations directly, we use a conditional subtraction to ensure the index stays within the allowed range $[0,(B-1)]$. Listing 6.2 shows the P4 implementation.

```
action cuckoo_calc_index_pair() {
    // Calculate index1 based on the full 4-tuple
    hash(meta.cuckoo_index1, HashAlgorithm.crc32, (bit<32>)0,
        {hdr.ipv4.src_addr, hdr.ipv4.dst_addr, hdr.tcp.src_port,
            hdr.tcp.dst_port},
        (bit<32>) N_BUCKETS_MINUS_ONE);
    // First calculation step for index2 based on the
       fingerprint
    hash(meta.cuckoo_index2, HashAlgorithm.crc32, (bit<32>)0,
         {meta.cuckoo_fingerprint},
         (bit<32>) N_BUCKETS_MINUS_ONE);

    // Second calculation step for index2
    bit<32> temp_index;
    temp_index = meta.cuckoo_index1 ^ meta.cuckoo_index2;

    if (temp_index >= N_BUCKETS) {
        temp_index = temp_index - N_BUCKETS;
    }

    meta.cuckoo_index2 = temp_index;
}
```

Listing 6.2: Index pair calculation for Cuckoo hashing in P4

Core Operations: Insertion, Lookup, and Deletion

The SmartNIC Agent performs three primary operations on the Cuckoo filter (see Section 2.5):

- **Insertion:** A fingerprint is inserted into one of its two candidate buckets indicated by the two indices. If both are full, a random bucket is chosen, and the fingerprint is randomly inserted, causing an existing entry to be evicted and relocated, potentially triggering a chain of displacements.
- **Lookup:** To check for membership, both candidate buckets are scanned for the fingerprint. A match indicates likely membership (with some false positive risk).
- **Deletion:** The fingerprint is searched in both candidate buckets and deleted if found.

The lookup and deletion mechanisms are relatively straightforward and, therefore, not discussed in further detail. Instead, the focus is placed on illustrating how the insertion logic—the most complex operation due to its potential for recursive evictions—is realized within the constraints of P4 using packet recirculation.

Insertion Mechanism in P4

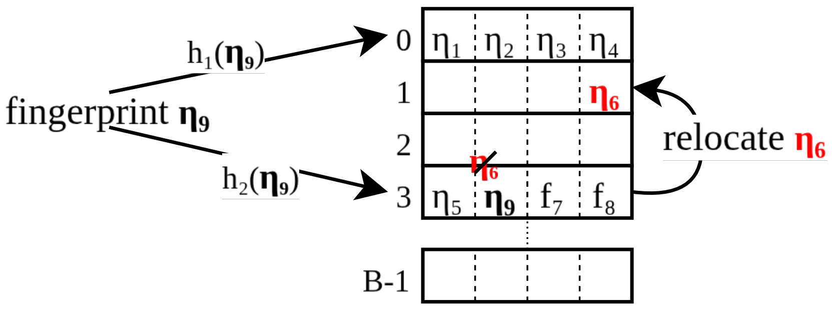

Figure 6.1: Insertion of fingerprint η_9 into a Cuckoo filter with B buckets using Cuckoo hashing (h_1, h_2).

The insertion process is illustrated in Figure 6.1. Due to P4's lack of native looping constructs and dynamic control flow, the recursive aspect of the insertion logic is emulated through *packet recirculation*. When both candidate buckets are full, the fingerprint selected for displacement (here η_6) is stored in packet metadata along with the current recirculation count. The packet is then recirculated alongside its metadata to retry insertion of the displaced fingerprint into its alternate bucket. This process may repeat up to a configured maximum number of evictions, denoted as `MaxNumKicks`. In line with the original configuration proposed by Fan et al. [32], we set `MaxNumKicks` to 500.

6.1.2 Server Agent: eBPF

The Server Agent is implemented using eBPF and distributed across the egress and ingress paths of the Linux kernel's networking stack. Specifically, the eBPF implementation utilizes Traffic Control (TC) at the egress and eXpress Data Path (XDP) at the ingress. Both paths share access to a common BPF map, which maintains the TCP connection state:

```
BPF_TABLE_PINNED("hash", map_key_t, map_val_t, nonpercpu_bpf_map
    , 2000, "/sys/fs/bpf/my_nonpercpu_map");
```

Listing 6.3: Shared BPF map definition and structures

Where the structures `map_key_t` and `map_val_t` are defined as follows:

```
typedef struct map_key_t {
    uint32_t src_ip;
    uint32_t dst_ip;
```

```c
    uint16_t src_port;
    uint16_t dst_port;
} map_key_t;

typedef struct map_val_t {
    uint32_t delta;
    uint8_t map_state;
} map_val_t;
```

Listing 6.4: Structure definitions for BPF map keys and values

All together, the Server Agent keeps 136 bits of explicit state for each verified (benign) connection.

Extended TCP Connection State Machine

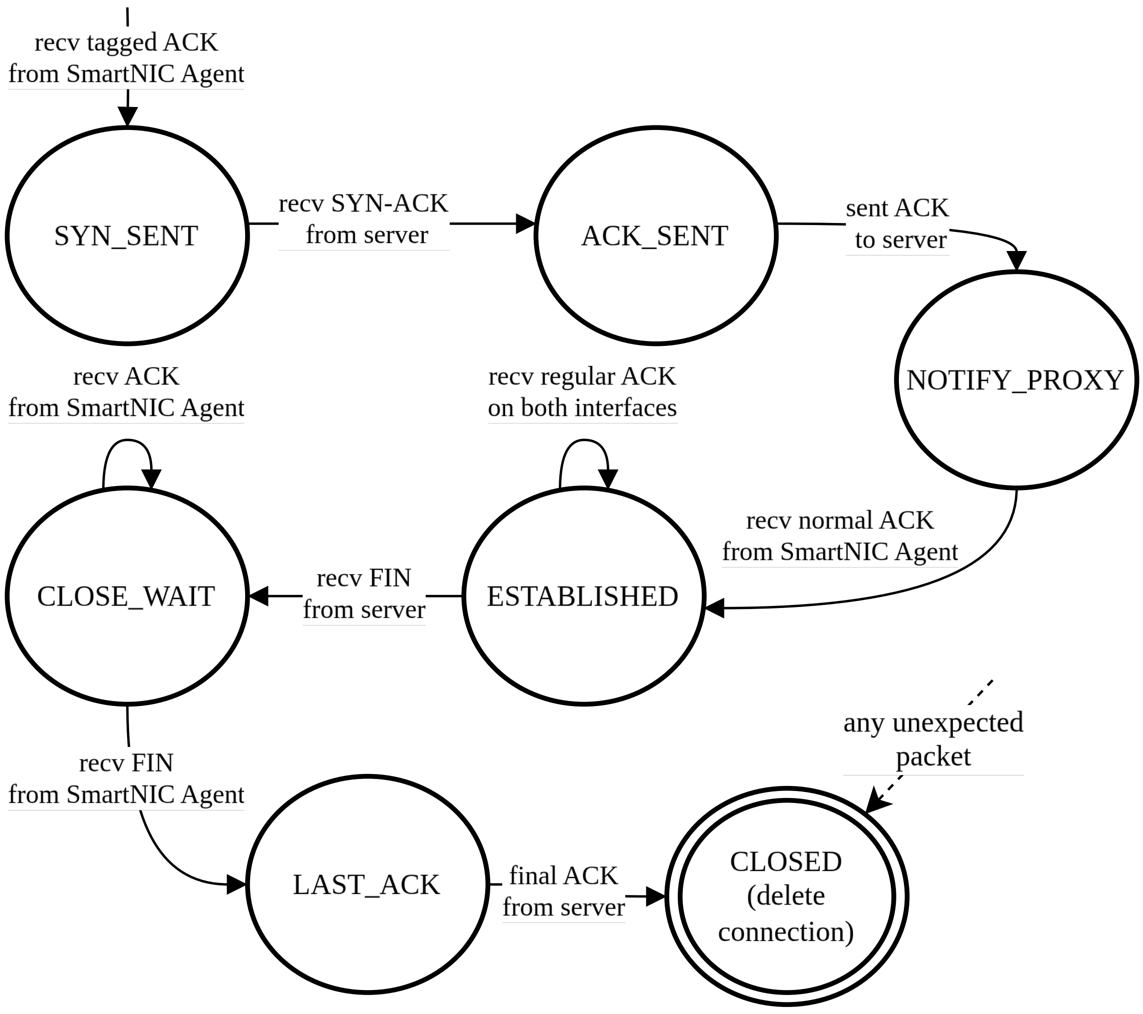
recv tagged ACK from SmartNIC Agent
SYN_SENT
recv SYN-ACK from server
ACK_SENT
sent ACK to server
NOTIFY_PROXY
recv ACK from SmartNIC Agent
recv regular ACK on both interfaces
recv normal ACK from SmartNIC Agent
CLOSE_WAIT
recv FIN from server
ESTABLISHED
any unexpected packet
recv FIN from SmartNIC Agent
LAST_ACK
final ACK from server
CLOSED (delete connection)

Figure 6.2: Connection State Machine in the Server Agent.

Figure 6.2 depicts the connection states monitored by the Server Agent for each individual connection. Upon receiving a tagged ACK packet from the SmartNIC Agent, indicating a legitimate new connection attempt, the Server Agent commences the TCP three-way handshake with the server by sending a SYN packet to the server (`SYN_SENT`). Upon receiving the SYN-ACK response from the server, the connection state advances to `ACK_SENT`. Once the final ACK packet is sent to the server, completing the handshake, the state machine immediately transitions to `NOTIFY_PROXY`. In this state, the SmartNIC Agent is notified to incorporate the newly established connection's flow identifier into its flow filter, thus ensuring the correct forwarding of all subsequent packets.

After successful establishment, the connection persists for regular TCP communication in the `ESTABLISHED` state. The server initiating connection termination, indicated by a FIN packet, transitions the state machine into the `CLOSE_WAIT` state. During `CLOSE_WAIT`, the server may still receive ACK packets from the client. The connection remains in this state until a FIN packet from the client is received, triggering a transition to the `LAST_ACK` state, awaiting the server's final acknowledgment.

Eventually, the state machine reaches the `CLOSED` state, prompting the removal of the corresponding flow entry from the Cuckoo filter and purging all related connection states from the BPF map. Robustness to unexpected and connection-terminating behaviour is explicitly integrated into the state machine design; unexpected packets of an ongoing connection, such as RST packets signaling abrupt connection termination, immediately transition the connection to the `CLOSED` state to reset the connection and prevent memory resource exhaustion.

Robust Detection of Connection Termination in the Connection State Machine

The implemented state machine comprehensively supports one of the standard TCP connection termination scenarios described in RFC 793 [59]:

- **Remote TCP-initiated termination via FIN segments**, i.e., the server initiates connection closure. This is the most common scenario for web servers and similar services.

Other termination scenarios defined in RFC 793 include:

- **Closure initiated by the user**. This scenario is not within the scope of the current implementation, but it can be readily incorporated by extending the current state machine model. We decided to omit it to prevent unnecessary complexity in the implementation and the thesis. Nevertheless, we formally verified that its inclusion is feasible.

- **Simultaneous termination initiated by both connection endpoints**. This scenario is likewise not supported in the current state machine. We deliberately excluded it to reduce implementation complexity and facilitate comprehensive testing. However, its addition presents no conceptual difficulties.

Robustness to irregular client behaviour

More importantly, our state machine model is designed to handle irregular and unexpected client behavior. This robustness is critical to prevent scenarios in which the connection state is retained indefinitely, leading to memory exhaustion within our architecture. The following outline the system's resilience to such irregular cases, assuming the server's TCP stack, which is under our control, adheres to protocol standards.

RFC 793 explicitly outlines the procedures for managing standard termination scenarios. It ensures that segments preceding and including FIN segments are reliably retransmitted until acknowledged, thereby preserving protocol integrity (RFC 793, Section 3.5). Additionally, unsolicited FIN segments received from the network must trigger immediate acknowledgments and user notifications, facilitating graceful connection closure.

Moreover, RFC 793 mandates that in the event of a client-side TCP stack crash and restart, the client must issue an RST packet upon receiving further packets from the server. This behavior immediately notifies the server to close its corresponding connection state (RFC 793, Section 3.4), ensuring timely resource cleanup.

The case of a completely unresponsive client is also addressed. As stipulated in RFC 1122 [13] (Section 4.2.3.5), persistent lack of acknowledgment after repeated retransmission attempts necessitates connection abortion. RFC 793 defines this as: *"an abort causes the TCB to be removed, and a special RESET message to be sent to the TCP on the other side of the connection"* (RFC 793, Section 3.4).

This behavior is essential, as it guarantees that connection termination is always observable by the Server Agent—either explicitly through the standard TCP closing sequence or implicitly via exceptional events, such as the reception of a TCP reset (RST) segment. Consequently, the Server Agent's TCP state machine, assuming that the server adheres strictly to TCP protocol specifications, can robustly handle irregular scenarios. This design ensures effective resource reclamation and prevents the uncontrolled accumulation of connection state, safeguarding against memory exhaustion.

6.2 SMARTCOOKIE Baseline

Our SMARTCOOKIE baseline builds upon the open-source P4 implementation [72] originally designed for the Intel Tofino platform, which we ported to the BMv2 software target.

The implementation also follows a split-proxy architecture: SYN-Cookie generation and flow filtering are handled entirely within the data plane (P4), while a server-side eBPF agent is responsible for sequence number translation and connection state management. The available memory m (in bits) and the expected number of concurrently active benign flows n are relevant parameters used in the following. For flow filtering, we evaluate two Bloom filter variants:

- The *(Ideal) Bloom filter* is a theoretical precision upper bound for standard Bloom filters. The number of hash functions, k, is set as an integer variable, and using linear search, the optimal value minimizing expected false positives can quickly be found (ideal k does not exceed 30 up to filter sizes of up to 1 TB). The expected false positive rate is calculated using the following exact formulation [2]:

  $$F_s(n,m,k) = \sum_{i=1}^{m} S(n,m,k,i) \left(\frac{i}{m}\right)^k, \tag{6.3}$$

  where:

  $$S(n,m,k,i) = B(nk,m,i); \tag{6.4}$$

  $$B(n,m,i) = \binom{m}{i} \cdot i^n \cdot m^{-n}. \tag{6.5}$$

  To implement k hash functions, we use CRC32 and CRC16 [36] as base functions ($k = 1$ and $k = 2$) and generate further hashes using the method [32]:

  $$\text{hash}_k(x) = \text{CRC32}(x) + (k-2) \cdot \text{CRC16}(x). \tag{6.6}$$

  This represents an idealized setting, as real hardware targets such as Intel Tofino often impose strict limits on the number of hash function evaluations and memory accesses per packet, constraining filter scalability. Moreover, since there is no practical mechanism in P4 to dynamically adjust the number of hash functions based on the available memory m and the number of tracked elements n, the optimal configuration must be hardcoded. As a result, multiple pre-tuned implementation versions are required, each tailored to specific deployment scenarios to ensure maximal efficiency.

- The *(Ideal) Bloom filter with time-decaying,* enables connection entry expiry by maintaining two (Ideal) Bloom filters in parallel, with alternating resets every interval t. Each filter uses $m/2$ bits of memory and flows not seen for $2t$ are implicitly removed. The optimal parameters for each are determined equivalently to those for the ideal Bloom filter.

6.3 Static Code Analysis and Comparison

This section presents a static analysis comparing the implementation complexity of the CuckooGuard and SmartCookie flow filters, focusing solely on the P4 components executed on the SmartNIC Agent. Our objective is to approximate the relative hardware cost, measured in terms of logical complexity, of each approach when synthesized onto an FPGA of a SmartNIC. Although exact hardware usage (e.g., LUTs or Flip-Flops) depends on downstream optimizations and synthesis targets, static metrics provide a first-order estimate of comparative resource requirements.

To this end, we apply the ABC complexity metric [85], which evaluates the structural complexity of the code based on the number of assignments (A), branches (B), and conditionals (C). This metric has been used in software engineering as a lightweight predictor of implementation cost and maintainability, and here serves as a rough proxy for logic resource demand. We focus exclusively on the code implementing the flow filters, omitting other components unrelated to the filtering logic to ensure a fair comparison.

Table 6.1: Static analysis of P4 flow filter implementations using the ABC complexity metric.

Filter Type	**ABC Tuple (A, B, C)**	**ABC Scalar**	**Lines of Code (LoC)**
Bloom Filter ($k = 7$)	(16, 42, 10)	46.04	60
Cuckoo Filter	(34, 40, 21)	56.54	126

The results summarized in Table 6.1 indicate that the Cuckoo filter implementation incurs higher complexity than the Bloom filter baseline, as was expected. This suggests a greater demand for logical resources (e.g., LUTs, Flip-Flops) when targeting FPGA-based SmartNICs. However, it is essential to emphasize that the code complexity of the Bloom filter depends on the number of hash functions k (see Section 6.2). For a typical $k = 7$, the Bloom filter logic is relatively compact, but increasing k to be optimal for larger filters would proportionally raise its complexity, potentially narrowing the gap. Moreover, these figures represent unoptimized source-level complexity; hardware synthesis results may differ based on backend compiler optimizations or manual hardware-aware code tuning.

6.4 Security Analysis

Given that DDoS mitigation represents a critical security component, we must ensure that an adversary, possessing knowledge of our defensive approach (CuckooGuard), cannot bypass or compromise it. Here, we analyze two critical attack scenarios:

1. Forging packets that bypass the Cuckoo filter, with the objective of exhausting the server's computational resources through repeated fallback cookie checks.

2. Crafting packets that purposefully induce collisions within the Cuckoo filter, causing targeted denial-of-service (DoS) conditions.

6.4.1 Forged Packets and Computational Exhaustion

An attacker may theoretically craft packets specifically designed to generate identical fingerprints within the Cuckoo filter. Let us formalize this threat scenario: given a packet m with a fingerprint $h(m)$, an attacker could seek to generate a malicious packet m' where $m' \neq m$, yet $h(m') = h(m)$. Initially, the attacker would establish a benign TCP connection that legitimately registers the fingerprint $h(m)$ in the Cuckoo filter. Subsequently, the attacker transmits forged ACK packets with distinct connection tuples from the benign connection (e.g., different source ports or source IP addresses), explicitly designed to map to the same fingerprint $h(m)$.

If these spoofed packets exhibit an identical 4-tuple, the kernel-level TCP/IP stack will detect anomalies and trigger connection resets, subsequently removing these entries from the Cuckoo filter, effectively mitigating this risk. However, if spoofed packets differ sufficiently (distinct 4-tuple), each forged ACK packet triggers computationally intensive fallback cookie validation in the Server Agent. This tactic could saturate server computational resources, achieving a denial-of-service through resource exhaustion.

6.4.2 Targeted Collision Attacks

An alternative attack scenario exploits hash collisions to specifically target and deny service to a particular user. This attack involves the following steps: First, the attacker identifies the victim's anticipated connection tuple, specifically the source IP address and port. The attacker then proactively establishes multiple benign connections designed to intentionally produce identical fingerprints within the Cuckoo filter, matching the anticipated fingerprint of the victim's future connection.

Considering each bucket within the Cuckoo filter stores b entries (typically $b = 4$), the attacker strategically fills both candidate buckets associated with the victim's fingerprint, totaling $2b$ occupied entries. Practically, this equates to establishing only $2b$ benign-looking connections, which is within feasible limits. Consequently, when the victim subsequently attempts to establish their legitimate connection, due to fully occupied buckets, the Cuckoo filter reaches its eviction limit (defined by `maxNumKicks`), and the victim's legitimate fingerprint insertion fails. As a result, subsequent legitimate packets and connection attempts from the victim are blocked, never reaching the server, thereby causing a targeted denial-of-service.

However, practically achieving multiple hash collisions (specifically $2b$ identical fingerprints) is challenging. The attacker's degrees of freedom in packet crafting are limited, primarily restricted to adjusting the source port and potentially the source IP, depending upon their network configuration.

Notably, victims are generally resilient to this attack scenario due to the standardized use of randomized source ports generated immediately before initiating a connection. For the described collision attack to succeed, the attacker must predict the victim's source port before the victim's final handshake packet reaches the server. In a realistic scenario, a victim randomizing source ports would require the attacker to directly access network traffic between the victim and the server. The attacker must observe the victim's initial SYN packet to determine the victim's source port and IP address. Following this observation, the attacker must rapidly establish the necessary $2b$ malicious connections before the victim completes their handshake.

This scenario imposes significant timing constraints, making it even more challenging for an attacker. Although an attacker could theoretically prepare in advance to quickly generate the required attack packets, the feasibility of successfully executing such an attack remains highly limited without direct network interception capabilities.

6.4.3 Second Preimage Resistance and Practical Considerations

The feasibility of the described attacks depends on the Cuckoo filter fingerprint hash function's resistance properties, specifically, second preimage resistance. Second preimage resistance ensures that, given a legitimate packet m with hash output $h(m)$, it is computationally infeasible to find a distinct packet m' where $m' \neq m$, yet $h(m') = h(m)$.

Our prototype currently employs CRC32 for fingerprint hashing, which lacks second preimage resistance, rendering it vulnerable to this collision-based exploitation. Thus, an attacker with full access to source code and deployment specifics could feasibly craft malicious packets meeting the collision criteria outlined above. Consequently, in practical and production deployments, we must replace CRC32 with a cryptographically secure hash function possessing the necessary second preimage resistance.

A robust candidate is SipHash, designed explicitly for performance-critical network environments, providing cryptographic security against collision attacks [6]. Implementations of cryptographic hash functions such as SipHash have been demonstrated in data plane environments using P4 [72].

To ensure robust security in production environments, we strongly recommend selecting cryptographically secure hash functions compatible with available hardware acceleration capabilities in the deployed SmartNIC. By employing a cryptographically secure hash function with sufficient second preimage resistance, our CuckooGuard architecture effectively mitigates the described collision-based attack vectors, ensuring reliable and robust DDoS protection.

Chapter 7

Evaluation

The following evaluation section presents a comprehensive analysis of the CuckooGuard architecture based on the implementations introduced in the preceding implementation chapter. The primary objective is to assess the practical viability, efficiency, and precision of CuckooGuard by comparing alternative filtering strategies and state-of-the-art approaches.

The evaluation is structured around the following core goals:

- Cuckoo filter insertion overhead
- Bloom vs. Cuckoo filter connection tracking behavior
- Comparative analysis of flow filter accuracy
- CuckooGuard vs. SmartCookie performance comparison in a realistic scenario

7.1 Experimental Setup

All experiments follow a unified setup and parameterization to enable fair comparison of connection-tracking approaches, focusing on memory usage and precision of flow filters rather than absolute throughput or latency. Bloom filter variants are evaluated within the SmartCookie baseline implementation (Section 6.2), while Cuckoo filters are tested using CuckooGuard's implementation (Section 6.1). Experiments are conducted on a single commercial off-the-shelf (COTS) machine running `Ubuntu 20.04.6 LTS` with kernel version `5.15.0-134-generic`, using Linux virtual interfaces and network namespaces to emulate the split-proxy setup. In addition to the practical, time-decaying (Ideal) Bloom filter, we include an (Ideal) Bloom filter as a non-deployable reference to illustrate the theoretical precision limit. For consistency, decay is disabled in the time-decaying variant (via timeout t) unless stated otherwise.

We fix the number of actively tracked connections at $n = 5000$ and vary the available memory m (in bits), which determines the per-connection memory budget m/n and directly impacts filter precision. Due to BMv2's performance constraints, relatively small values for n and m are used to allow fine-grained variation of m and high-resolution benchmarking results. While not representative of large-scale deployments, these parameters can be proportionally scaled, preserving the filters' relative behavior [32]. To ensure statistical robustness, each measurement was repeated ten times and averaged; false positive rate measurements were computed from 100 000 randomly generated spoofed packets per run. Unless otherwise stated, Cuckoo filter configuration parameters are set as follows: bucket capacity $b = 4$, and target load factor $\alpha = 0.95$.

To validate scalability, we adopt one large-scale experimental scenario introduced by Yoo et al. [92] for their SMARTCOOKIE architecture. This baseline tracks 579 600 concurrent TCP connections under an ACK flooding attack, using time-decaying, partitioned Bloom filters [80] with $k = 3$ partitions of 2^{20} bits each, totaling approximately 786 KB of memory. We use this setup to quantify the concrete advantages of CUCKOOGUARD under realistic load.

The experimental evaluation focuses on two primary Key Performance Indicators (KPIs):

- **False Positive Rate**, which directly impacts server-side computational load by triggering unnecessary cookie validations during ACK floods.

- **Recirculation Overhead**, which quantifies the number of packet reprocessing cycles required at the SmartNIC (Cuckoo filter Insertion).

7.2 Experimental Results

7.2.1 Cuckoo Filters offer better Precision under low False Positive Rate Requirements

Figure 7.1 shows the relationship between the false positive rate and the available memory per connection. Both Bloom filter variants exhibit gradually improving (decreasing) false positive rates with increasing memory; the Cuckoo filter, on the other hand, displays step-wise improvements. These steps are a direct consequence of fingerprint size adjustments (Section 6.1.1): as the available memory increases, the fingerprint size increases discretely, thereby exponentially reducing the probability of false positives due to hash collisions. The Cuckoo filter maintains a significant advantage over all types of Bloom filters up until a false positive rate of $\sim 0.1\%$. This is particularly beneficial, as achieving a low

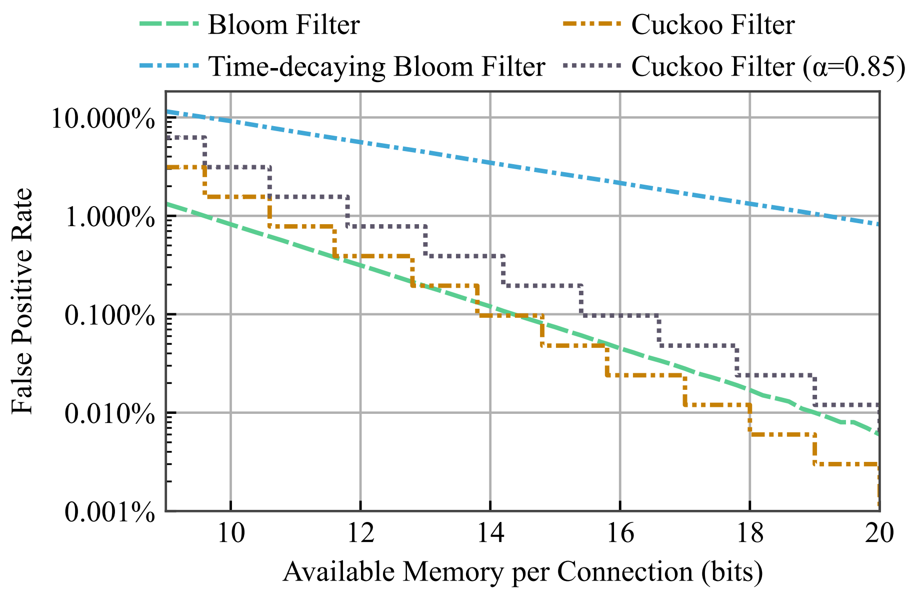

Figure 7.1: Memory-precision tradeoff: Comparison of false positive rates for Cuckoo filter and Bloom filter variants across varying memory per connection.

false positive rate $< 0.1\%$ is essential for effective server offloading (**R5 – OFF**) in CuckooGuard.

It can be observed that SmartCookie's Bloom filter with time-decaying consequently experiences significantly worse precision compared to CuckooGuard's Cuckoo filter. This confirms that CuckooGuard's Cuckoo filter achieves superior precision and therefore also memory efficiency.

7.2.2 Optimized Cuckoo Filter Insertion Overhead has Minimal Impact on False Positive Rates

Although prior experiments operated CuckooGuard's Cuckoo filter near its space-optimal capacity ($\alpha = 0.95$) to minimize the false positive rate, this configuration may not be ideal for performance. Therefore, we measure the average recirculation overhead under a fixed memory budget of $m =$84 227 bits while varying the load factor α, which denotes the fraction of occupied entries in the Cuckoo filter. Each configuration is evaluated over 1000 insertions, with the standard deviation capturing variability across these trials. Recirculation overhead is expressed as the average number of additional pipeline passes per insertion, represented as a percentage, where 100% corresponds to one extra pass per inserted flow entry.

Figure 7.2 illustrates the recirculation overhead of CuckooGuard's Cuckoo filter across various load factors. The average overhead, presented on a logarithmic scale, remains moderate—approximately 154%—at a load factor of $\alpha = 0.85$, but increases

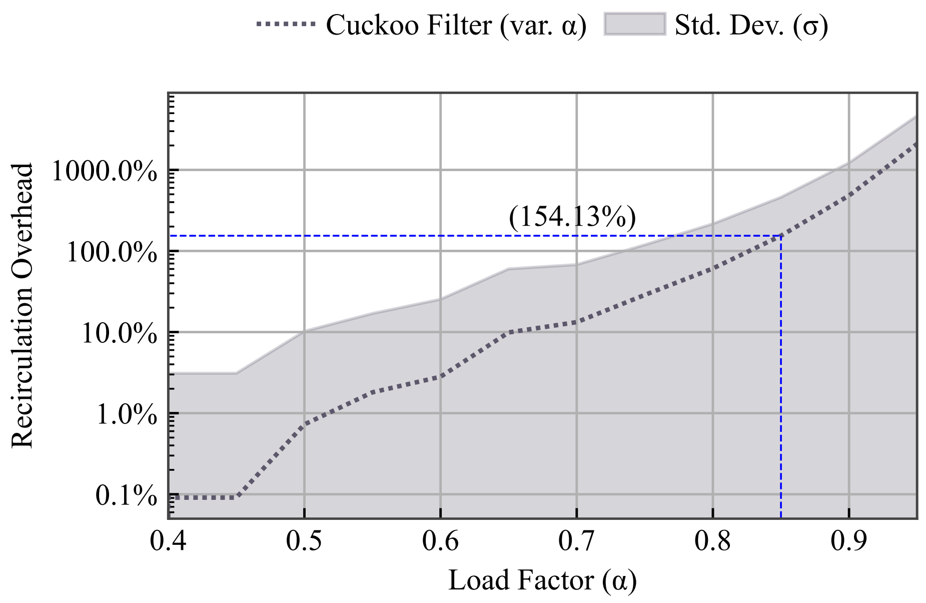

Figure 7.2: Insertion overhead of Cuckoo filters increases sharply with higher load factors.

sharply beyond this point, exceeding 1000% at $\alpha = 0.95$. Notably, the standard deviation also grows with the average overhead, reaching a maximum of $> 2000\%$. The variability is more controlled at lower load factors, e.g., $\alpha = 0.85$, but it escalates as the filter approaches full capacity. This insertion overhead must be interpreted in context. Since flow insertion occurs only once per TCP connection, whereas all other packets of the connection are forwarded without recirculation, the cost is amortized in long-lived connections. However, the overhead can accumulate and potentially degrade throughput in high-churn scenarios, such as REST APIs or microservice-based architectures, where connections are frequently established and torn down.

To assess the viability of operating the Cuckoo filter at a recirculation-optimized load factor $\alpha \leq 0.95$, we fix the available memory per connection at ~ 16.85 bits, corresponding to a total memory budget of 84 227 bits for 5000 concurrent connections. We then vary the load factor α of the Cuckoo filter to evaluate its impact on the false positive rate. Figure 7.3 presents the resulting false positive rates of the Cuckoo filter across different load factors and compares them to those of alternative flow filtering approaches.

We correlate recirculation cost with an achievable false positive rate to determine a practical operating point for α. The Cuckoo filter's false positive rate improves monotonically with a higher load, reaching a minimum at $\alpha = 0.95$; even at a conservative 0.85 load factor, it remains just above the idealized Bloom filter and outperforms the time-decaying Bloom filter by a wide margin.

This insight suggests that CUCKOOGUARD's Cuckoo filter can be tuned to mitigate its primary disadvantage—recirculation—without significantly compromising precision. A load factor of $\alpha = 0.85$ offers a favorable trade-off, achieving a low false positive

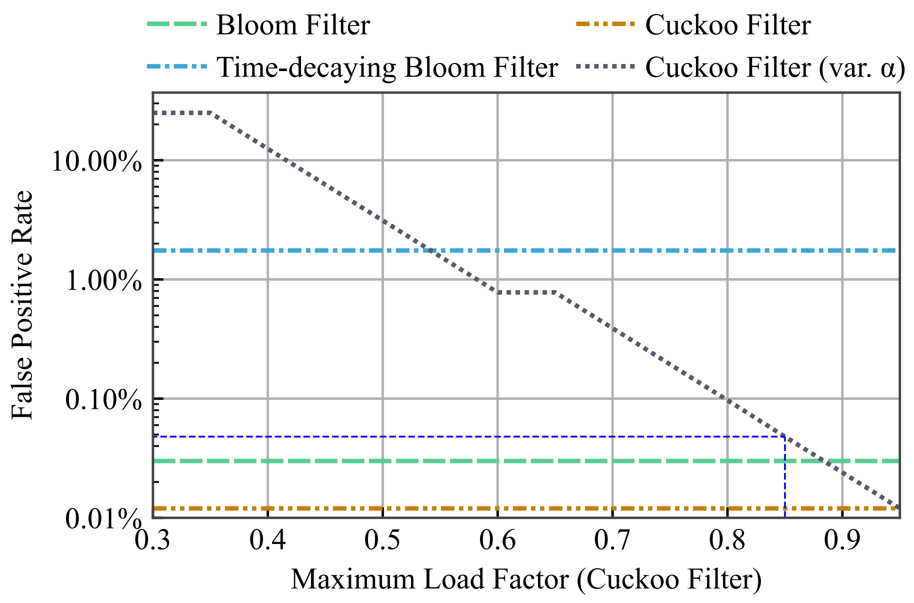

Figure 7.3: False positive rate of Cuckoo filters under varying load factors. Higher occupancy improves precision but increases insertion overhead.

rate throughout (see Figure 7.1) while maintaining reasonable processing overhead. This corresponds to an average of just over one additional recirculation per TCP connection, making the overhead negligible in most deployment scenarios.

7.2.3 Cuckoo Filter Ensures Minimal Overhead and Precise Connection Tracking

Figure 7.4 shows both designs' flow filter occupancy over time under a workload of 200 new connections per second (*cps*), each lasting exactly 5 seconds. This setup reveals how each filter handles dynamic connection churn across three distinct phases: a *growth phase* (0-5 s), a *steady-state phase* (5-35 s), and a *drain phase* (35-40 s). The Cuckoo filter exhibits linear growth during the initial phase, stabilizing at 1000 flow entries—the number of active connections sustained at 200 *cps* with 5-second lifetimes. During the steady-state phase, flow entry insertions and deletions balance out, keeping occupancy flat. Once new arrivals stop, the filter empties smoothly. This behavior reflects precise connection tracking via explicit deletions triggered by CUCKOOGUARD's Server Agent, the distinctive feature of CUCKOOGUARD.

In contrast, the time-decaying Bloom filter ($t = 5\,s$) exhibits oscillating occupancy during the steady-state phase, fluctuating between 2000 and 5000 entries with sharp drops every 5 seconds due to filter instance resets. Each connection's flow entry is inserted into two overlapping filter instances: one is reset every 5 seconds, and both remain active for 10 seconds. The overlap between active and new connections extends the effective tracking

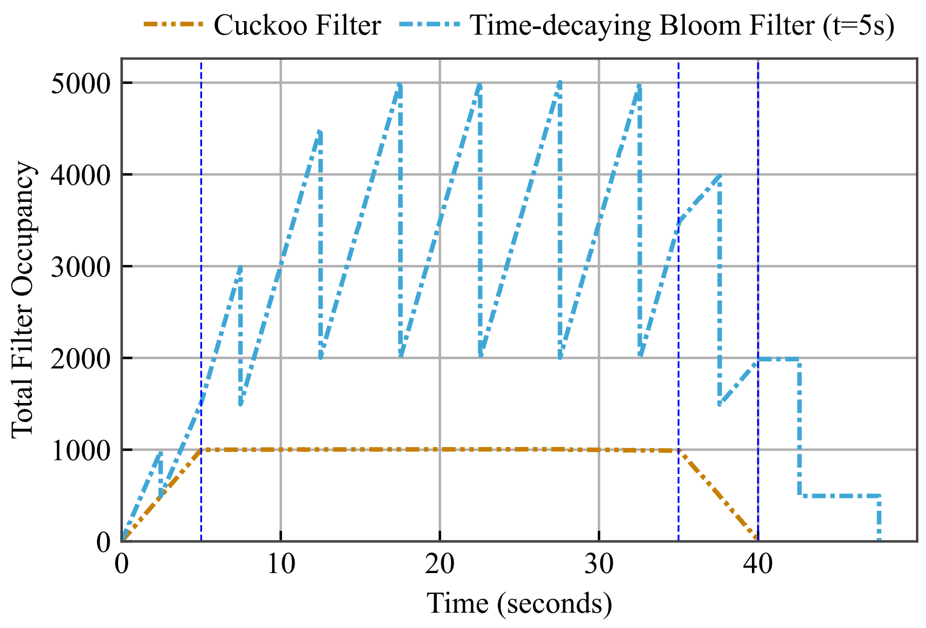

Figure 7.4: Temporal behavior of filter occupancy under dynamic connection churn. Cuckoo filters track flows precisely with minimal memory overhead, while time-decaying Bloom filters require overprovisioning.

window per instance to 15 seconds. At a rate of 200 *cps*, each instance must support up to 3000 flow entries. During the drain phase, occupancy decreases as existing connections expire. Full clearing occurs once traffic ceases and both filter instances complete their rotation.

Lacking support for flow entry deletion, this Bloom filter must overprovision. Each filter instance holds 3× the active flow entries, and a second instance doubles that requirement, resulting in a 6× flow entry overhead. In contrast, CUCKOOGUARD's Cuckoo filter maintains a tight one-to-one mapping with active connections, offering precise expiration and optimal flow entry overhead.

7.2.4 Computational Overhead Decreased by 79% Compared to the State of the Art: SMARTCOOKIE

To ensure a fair and direct comparison, CUCKOOGUARD is configured to match the memory and connection tracking parameters of the SMARTCOOKIE baseline, following the methodology described in Section 6.1.1.

During evaluation, both systems are subjected to an ACK flood attack consisting of 100 000 spoofed and randomized ACK packets. This attack is repeated across 10 independent runs to ensure statistical reliability. In this setting, CUCKOOGUARD achieves an average false positive rate of 1.56%, significantly outperforming the 7.66% rate observed in the SMARTCOOKIE baseline. This improvement substantially reduces unnecessary server-side cookie verifications, leading to an estimated decrease in CPU overhead of up to **79%**. Moreover, our results suggest that the performance gap widens even further at lower false positive thresholds (e.g., below 0.1%), indicating strong scalability potential of CUCKOOGUARD.

Due to the lack of cryptographic hash function support in BMv2, direct measurement of server-side CPU usage was not conducted. In realistic deployments, cryptographic hashing is expected to constitute the primary computational bottleneck for the Server Agent, and performance evaluations based on CRC hashing would not yield representative results. Instead, we refer to the evaluation of SMARTCOOKIE [72], which reported a CPU usage of approximately 372 million instructions per second (*MIPS*) under a 30 Mpps ACK flood attack [72]. Since our eBPF implementation is a modified derivative of the SMARTCOOKIE Server Agent and achieves a more than fourfold reduction in false positive rate, we estimate the server-side CPU overhead of CUCKOOGUARD to be proportionally lower. Based on this relationship, CUCKOOGUARD is expected to reduce CPU usage to approximately 78 MIPS under comparable conditions. We emphasize that this estimate is only a coarse approximation; precise performance measurements are left for future work once a hardware-based implementation with cryptographic hash support is available. Nonetheless, these projections suggest that CUCKOOGUARD offers substantial practical benefits in terms of server offloading.

Chapter 8

Conclusion

8.1 Learned Lessons

This thesis introduced CuckooGuard, a novel SYN flood defense architecture for programmable data planes that leverages a dynamic connection-tracking mechanism based on Cuckoo filters. The architecture achieves high precision in flow filtering while maintaining minimal overhead, making it particularly well-suited for deployment in memory-constrained environments such as FPGA-based SmartNICs.

Among several candidate data structures considered for flow filtering, the Cuckoo filter emerged as the most suitable choice due to its compact memory footprint, support for explicit entry deletions, and tunable trade-offs between precision and performance. Our implementation demonstrates that Cuckoo filters significantly outperform Bloom filter variants in terms of false positive rates, especially under strict memory constraints. While flow entry insertions into the Cuckoo filter necessitate packet recirculation—an inherent limitation in data plane environments—the associated overhead is bounded and can be controlled by adjusting the filter's load factor.

Experimental evaluation confirms the viability of CuckooGuard, showing a substantial **79% reduction in server CPU load** compared to the state-of-the-art SmartCookie architecture under equivalent memory conditions. These findings underscore the efficiency of offloading SYN-Cookie validation and flow tracking to the programmable data plane, enabling precise and scalable defense even in high-churn environments.

Key Achievements:

- Development of an architecture that satisfies all defined design requirements (**R1–R7**), including memory efficiency, transparency, and high connection churn tolerance.
- Robust protection against both SYN and ACK flood attacks for any TCP-based

server, without requiring changes to the application layer.

- Full exploitation of programmable hardware to enable low-latency, line-rate packet processing in the data plane.
- Demonstrated significant improvement over prior work, achieving up to 79% reduction in server-side computational overhead under identical conditions, and exhibiting markedly better performance in more realistic, high-precision deployment scenarios.
- Introduction of Cuckoo filters as a foundational building block for connection-tracking in programmable data planes.
- Implementation of a deployable proof-of-concept architecture that can be readily adapted for real-world network defense scenarios.

8.2 Future Work

While CuckooGuard has demonstrated strong performance in a simulated environment, several avenues for future exploration remain. Each of the following directions addresses a distinct challenge or limitation and outlines a detailed plan for further development and evaluation.

8.2.1 Deployment and Evaluation on SmartNIC Hardware and Servers

To validate CuckooGuard under realistic network conditions, a comprehensive empirical evaluation on production-grade SmartNIC hardware is essential. This involves establishing a hardware testbed and collecting detailed performance metrics, including throughput, latency, and false positive rates.

The evaluation should begin by setting a performance target (e.g., 200 Gbps line rate, > 30 Mpps) and acquiring compatible hardware. Ideally, the setup should include a programmable (FPGA-based) SmartNIC that supports P4, along with a high-performance NIC capable of hardware timestamping for accurate latency measurements. A server with such a SmartNIC will host CuckooGuard, while a dedicated client system will generate test traffic over a high-speed link.

The testbed must be able to support a variety of traffic patterns, combining legitimate and attack traffic. Simulated workloads should, for example, include high-churn REST API traffic and long-lived streaming sessions to reflect realistic use cases. For attack traffic, randomized SYN and ACK flood attacks should either be synthesized or obtained from open-source research datasets to benchmark resilience.

Beyond basic deployment, this line of work also opens the door for analyzing architectural variants of the system. A particular focus lies on the possibility of relocating the Server Agent from the host to a general-purpose CPU integrated on the SmartNIC itself, if available. Such an offloading strategy could reduce inter-device communication latency, but might introduce additional resource contention on the SmartNIC. Trade-offs in throughput, latency, and memory efficiency must be systematically evaluated for both the hybrid and fully offloaded models of CuckooGuard.

Furthermore, this deployment effort should also integrate a cryptographically secure and hardware-friendly hash function for SYN-Cookie generation. SipHash [6], for instance, is a strong candidate and should be implemented both on the SmartNIC and server. The computational hashing performance will become especially relevant under high ACK flood scenarios, where cookies must be verified at scale at both the server and the SmartNIC.

To complement the empirical evaluation, an additional objective should be to characterize how large the flow filter must be—i.e., the number of memory bits per connection—to prevent CPU overload during worst-case conditions. Specifically, under a constrained CPU budget (e.g., measured in MInstr/s) and a fixed deployment scenario, the analysis should determine how to dimension the Cuckoo filter data structure to maintain line-rate processing (e.g., 200 Gbps) while ensuring that no server-side resources are overstrained.

The expected outcome of this research direction is a fully instrumented and tunable deployment of CuckooGuard on real hardware, with quantified performance under representative workloads and attack scenarios. This includes latency measurements with high precision, throughput benchmarks in packets per second, and resilience testing against various flooding patterns. The results will form a critical step toward production-readiness and inform future optimization and hardening strategies.

8.2.2 Integration in Multi-Tenant SmartNIC Environments

In many datacenter environments, SmartNICs host multiple network functions concurrently. Integrating CuckooGuard in such a setting requires careful attention to resource isolation, scheduling, and cooperative execution.

The first step involves surveying prior work on multi-functional SmartNICs, especially Meili [79] and SuperNIC [50]. Building upon these insights, a multi-tenant setup should be prototyped on an FPGA-based SmartNIC or a simulated environment using BMv2 [63].

This direction should investigate how CuckooGuard coexists with other network functions such as TCP proxying, NAT, or packet inspection. Key research questions include:

- How can resource usage be minimized while CuckooGuard is in stand-by mode?

- What are the optimal conditions and mechanisms for activating CUCKOOGUARD?
- How can the SmartNIC's memory hierarchy be leveraged to reallocate resources to CUCKOOGUARD dynamically?
- Where in the data plane processing pipeline should CUCKOOGUARD be placed to avoid interference while preserving correctness?

This should result in a working integration prototype and performance, isolation, and interference analysis under different coexistence scenarios. This investigation focuses primarily on the SmartNIC Agent and assumes the Server Agent remains unchanged.

8.2.3 Semi-Sorted Cuckoo Filter Optimization

The original CUCKOOGUARD implementation uses standard Cuckoo filters for flow tracking. However, variations such as semi-sorted Cuckoo filters [32] offer a promising path toward improved memory efficiency at the expense of increased complexity.

This direction begins with an in-depth review of semi-sorted Cuckoo filters and their properties. The existing BMv2 [63] P4 implementation should be modified to incorporate this optimization and evaluated for its impact on memory footprint, lookup latency, insertion overhead, and performance.

Once the modified filter is operational, its integration into the CUCKOOGUARD architecture should be benchmarked against the original design. The evaluation should include synthetic traffic and realistic scenarios introduced in this thesis (e.g., API vs. streaming workloads).

Optionally, this may extend into hardware deployment as described above, to measure how these filter optimizations affect performance on real-world platforms.

8.2.4 Robust Handling of TCP Options

SYN-Cookie mechanisms often simplify or omit TCP options, which can reduce compatibility with modern applications. To ensure protocol transparency, CUCKOOGUARD must robustly handle TCP options such as Window Scaling [11] and Selective Acknowledgments (SACK) [34].

This direction should begin with a comprehensive survey of TCP options in common use, identifying their roles and implications for SYN flood defenses. The next step involves extending CUCKOOGUARD's Server Agent and potentially the SmartNIC Agent to implement a complete TCP state machine capable of interpreting and responding to these options.

The implementation phase should include reproducing various TCP behaviors in a testbed using applications that rely on these options. Compatibility issues must be identified and analyzed, followed by proposed architectural improvements to the Server Agent and/or SmartNIC Agent where necessary.

The outcome should include a fully option-aware Server Agent implementation, a test suite of option-heavy TCP clients, and a compatibility report outlining current limitations and potential fixes. This effort strengthens CUCKOOGUARD's protocol fidelity and broadens its applicability to real-world network deployments.

8.2.5 Integrating CUCKOOGUARD with Existing Network Telemetry Frameworks

Cuckoo filters are widely employed in network telemetry systems for their ability to provide fast, memory-efficient way to capture flow statistics. This opens a promising direction for integrating CUCKOOGUARD with existing telemetry frameworks that already leverage similar data structures for flow measurement and monitoring. The core idea is to unify connection tracking for the purpose of SYN flood protection and telemetry into a single, shared Cuckoo filter instance, reducing redundancy and conserving scarce SmartNIC memory.

To pursue this integration, a detailed survey of network measurement/ telemetry architectures using Cuckoo filters, such as [51, 37, 74], should first be conducted. These systems often use Cuckoo filters to maintain summaries of observed flows or to detect heavy hitters. In parallel, CUCKOOGUARD employs a Cuckoo filter to track verified TCP connections. A comparative analysis would identify the structural and semantic overlap between these use cases and assess compatibility at the data structure level.

Following this, a unified filter design should be developed, capable of maintaining both TCP connection validation state and telemetry-related metadata. The feasibility of such a combination must be investigated, particularly under high-load scenarios involving frequent insertions and deletions. Critical design challenges include maintaining acceptable false positive rates, and long-term filter coherence which is necessary for CUCKOOGUARD to work correctly.

The implementation phase would prototype this integration within a simulated environment (e.g., using BMv2) or on a SmartNIC platform if available. Evaluation metrics should include memory savings compared to a naive dual-filter design, impact on filter precision, and latency overhead for query operations.

Ultimately, this direction explores whether dual-purpose Cuckoo filters can support flow validation and telemetry without compromising performance or accuracy. If successful, the approach could significantly improve memory efficiency and functional density of SmartNIC deployments, especially relevant in multi-tenant environments where resource sharing is critical.

Bibliography

[1] A. M. Abdelhadi and G. G. Lemieux. Modular sram-based binary content-addressable memories. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 207–214. IEEE, 2015.

[2] P. S. Almeida. A case for partitioned bloom filters. *IEEE Transactions on Computers*, 72(6):1681–1691, 2022.

[3] AMD. Vitisnetp4: P4 ip for adaptive socs and fpgas. `https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/ef-di-vitisnetp4.html`, 2025. Accessed: 2025-05-29.

[4] APS Networks. APS2156D: 2.0Tb/s P4-Programmable TOR Switch. `https://www.aps-networks.com/products/aps2156d/`. Accessed: 2025-04-22.

[5] Arista Networks. Arista 7170 Series: High Performance Multi-function Programmable Platforms. `https://www.arista.com/en/products/7170-series`. Accessed: 2025-04-22.

[6] J.-P. Aumasson and D. J. Bernstein. Siphash: A fast short-input prf. In S. Galbraith and M. Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012*, pages 489–508, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, 2011.

[8] D. J. Bernstein. Syn cookies. `https://cr.yp.to/syncookies.html`, 1996. Accessed: 4 Feb. 2025.

[9] R. Bifulco, S. Miano, and C. Schneble. Accelerated DDoS attacks mitigation using programmable data plane. In *Proc. of the 15th ACM/IEEE Symposium on*

Architectures for Networking and Communications Systems (ANCS'19), pages 1–2, Cambridge, UK, September 2019.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[11] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, Sept. 2014.

[12] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[13] R. T. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, Oct. 1989.

[14] A. Caulfield, P. Costa, and M. Ghobadi. Beyond smartnics: Towards a fully programmable cloud. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2018.

[15] Center for Strategic and International Studies. Significant cyber incidents. `https://www.csis.org/programs/strategic-technologies-program/significant-cyber-incidents`, 2025. Accessed: 2025-04-19.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[17] L. Y.-C. Chang. Taiwan: A battlefield for cyberwar and disinformation. *Melbourne Asia Review*, 17, 2024. Accessed: 2025-04-29.

[18] X. Chen, C. Wu, X. Liu, Q. Huang, D. Zhang, H. Zhou, Q. Yang, and M. K. Khan. Empowering network security with programmable switches: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 25(3):1653–1704, 2023.

[19] Cisco Systems. Cisco Nexus 34180YC and 3464C Programmable Switches Data Sheet. `https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html`. Accessed: 2025-04-22.

[20] Cloudflare, Inc. DDoS Threat Report for 2024 Q4. https://radar.cloudflare.com/reports/ddos-2024-q4, February 2025. Accessed: 2025-04-19.

[21] T. L. K. Community. syncookies.c — syn cookie implementation in linux kernel. https://github.com/torvalds/linux/blob/master/net/ipv4/syncookies.c, 2025. Accessed: 2025-04-16.

[22] M. Dimolianis, A. Pavlidis, and V. Maglaris. SYN flood attack detection and mitigation using machine learning traffic classification and programmable data plane filtering. In *Proc. of the 24th Conference on Innovation in Clouds, Internet and Networks (ICIN'21)*, pages 1–8, Paris, France, March 2021.

[23] T. Döring, H. Stubbe, and K. Holzinger. Smartnics: Current trends in research and industry. *Network*, 19, 2021.

[24] DPDK Project. Data plane development kit (dpdk). https://www.dpdk.org/, 2025. Accessed: 2025-05-29.

[25] T. Döring. CuckooGuard: High-Precision SYN Flood Defense for SmartNICs: SYN Flood attack simulation and mitigation in P4. https://gitlab.com/Akira37/CuckooGuard-SYN-Flood-Defense-P4, 2024. Accessed: 2025-04-19.

[26] T. Döring. CuckooGuard: High-Precision SYN Flood Defense for SmartNICs: SYN Flood attack simulation and mitigation in P4. https://github.com/Akira1906/CuckooGuard-SYN-Flood-Defense-P4, 2025. Accessed: 2025-05-10.

[27] eBPF Documentation Project. Traffic control (tc) – classifier and action program type. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SCHED_CLS/, 2025. Accessed: 2025-03-31.

[28] eBPF Project. ebpf: Extended berkeley packet filter. https://ebpf.io/, 2025. Accessed: 2025-05-29.

[29] Edgecore Networks. DCS801 (Wedge100BF-32QS): 6.4T Programmable Data Center Switch. https://www.edge-core.com/product/dcs801/. Accessed: 2025-04-22.

[30] F. Faghih, T. Ziegler, Z. István, and C. Binnig. Smartnics in the cloud: The why, what and how of in-network processing for data-intensive applications. In *Companion of the 2024 International Conference on Management of Data*, pages 556–560, 2024.

[31] O. I. Falowo, M. Ozer, C. Li, and J. B. Abdo. Evolving malware and ddos attacks: Decadal longitudinal study. *IEEE Access*, 12:39221–39237, 2024.

[32] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[33] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[34] S. Floyd, J. Mahdavi, M. Mathis, and D. A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, Oct. 1996.

[35] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno. Towards a unified in-network DDoS detection and mitigation strategy. In *Proc. of the 6th IEEE Conference on Network Softwarization (NetSoft'20)*, pages 218–226, Ghent, Belgium, June 2020.

[36] T. Fujiwara, T. Kasami, and S. Lin. Error detecting capabilities of the shortened hamming codes adopted for error detection in ieee standard 802.3. *IEEE Transactions on communications*, 37(9):986–989, 1989.

[37] J. Grashöfer, F. Jacob, and H. Hartenstein. Towards application of cuckoo filters in network security monitoring. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 373–377. IEEE, 2018.

[38] X. Han, Y. Gao, T. Wood, and G. Parmer. Byways: High-performance, isolated network functions for multi-tenant cloud servers. In *Proc. of the ACM Symposium on Cloud Computing (SoCC'24)*, pages 792–810, Redmond, WA, USA, December 2024.

[39] 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. *IEEE Communications Surveys & Tutorials*, 23(4):2551–2595, 2021.

[40] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, Jan. 2019.

[41] Intel Corporation. Intel® P4 Studio: Developer Tools to Transform Networking. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html. Accessed: 2025-04-22.

[42] Intel Corporation. Intel® tofino™ 2.0 tbps, 2 pipelines - product specifications. https://www.intel.com/content/www/us/en/products/sku/218656/intel-tofino-2-0-tbps-2-pipelines/specifications.html, 2016. Accessed: 2025-02-06.

[43] Intel Corporation. Intel® Tofino Products, PCN 827577-00, Product Discontinuance, Tofino End of Life. https://www.intel.com/content/www/us/en/content-details/827577, August 2024. Product Change Notification 827577-00.

[44] ITIC. ITIC 2024 Hourly Cost of Downtime Report. https://itic-corp.com/itic-2024-hourly-cost-of-downtime-report/, 2024. [Online; accessed 12-May-2025].

[45] Kerrisk, Michael and The Linux man-pages project. Bpf-helpers(7) - linux programmer's manual. https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html, 2024. Accessed: 2025-04-19.

[46] D. Kopp, C. Dietzel, and O. Hohlfeld. DDoS Never Dies? An IXP Perspective on DDoS Amplification Attacks. In O. Hohlfeld, A. Lutu, and D. Levin, editors, *Passive and Active Measurement*, pages 284–301, Cham, 2021. Springer International Publishing.

[47] D. J. Lee. *A Practical Adaptive Quotient Filter*. PhD thesis, Williams College, 2021.

[48] J. Lemon. Resisting syn flood dos attacks with a syn cache. In *BSDCon 2002 (BSDCon 2002)*, 2002. Accessed: 2025-04-19.

[49] Y. Li, A. Kashyap, Y. Guo, and X. Lu. Compression analysis for bluefield-2/-3 data processing units: Lossy and lossless perspectives. *IEEE Micro*, 44(2):8–19, 2023.

[50] W. Lin, Y. Shan, R. Kosta, A. Krishnamurthy, and Y. Zhang. SuperNIC: An FPGA-based, cloud-oriented SmartNIC. In *Proc. of the 32nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'24)*, pages 130–141, Monterey, CA, USA, March 2024.

[51] J. Liu, P. Xun, and B. Wang. Relevant backtracking: An efficient telemetry data collection method for data center networks. In *2023 15th International Conference on Communication Software and Networks (ICCSN)*, pages 100–107. IEEE, 2023.

[52] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese. Introducing SmartNICs in server-based data plane processing: The DDoS mitigation use case. *IEEE Access*, 7:107161–107170, 2019.

[53] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Comput. Surv.*, 54(4), May 2021.

[54] Mininet Project Contributors. Mininet: An Instant Virtual Network on Your Laptop (or Other PC). `https://mininet.org/`. Accessed: 2025-04-22.

[55] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini, and M. Tornatore. Machine-learning-assisted ddos attack detection with p4 language. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2020.

[56] Networked Systems Group, ETH Zurich. p4-utils: A set of utilities to work with p4 software switches and mininet. `https://github.com/nsg-ethz/p4-utils`, 2024. Accessed: 2025-04-19.

[57] M. Nickel and D. Göhringer. A survey on architectures, hardware acceleration and challenges for in-network computing. *ACM Transactions on Reconfigurable Technology and Systems*, 2024.

[58] NVIDIA Corporation. NVIDIA BlueField-3 Data Processing Unit (DPU) Datasheet. `https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield`. Accessed: 2025-04-22.

[59] I. S. I. U. of Southern California. Transmission Control Protocol. RFC 793, Sept. 1981.

[60] P4 Language Consortium. P4_16 Portable Switch Architecture (PSA) Specification. `https://p4.org/p4-spec/docs/PSA.html`. Accessed: 2025-04-22.

[61] P4 Language Consortium. p4runtime: Specification Documents for the P4Runtime Control-Plane API. `https://github.com/p4lang/p4runtime`. Accessed: 2025-04-22.

[62] P4 Language Consortium. P4˜16 language specification, version 1.0.0. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2020. Accessed: 2025-03-28.

[63] P4 Language Consortium. behavioral-model: P4 software switch (bmv2). https://github.com/p4lang/behavioral-model, 2024. Accessed: 2025-03-28.

[64] P4 Language Consortium. Bmv2 performance guide. https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md, 2025. Accessed: 2025-04-16.

[65] p4lang. Intel® P4 Studio Software Development Environment. https://github.com/p4lang/open-p4studio. Accessed: 2025-04-22.

[66] p4lang. p4c: P4_16 Reference Compiler. https://github.com/p4lang/p4c. Accessed: 2025-04-22.

[67] P4.org API Working Group. P4Runtime Specification Version 1.3.0. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.pdf, 2021. Accessed: 2025-04-22.

[68] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[69] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1386–1399, 2021.

[70] P. Pandey, M. Farach-Colton, N. Dayan, and H. Zhang. Beyond bloom: A tutorial on future feature-rich filters. In *Companion of the 2024 International Conference on Management of Data*, pages 636–644, 2024.

[71] F. Paolucci, F. Civerchia, A. Sgambelluri, A. Giorgetti, F. Cugini, and P. Castoldi. P4 edge node enabling stateful traffic engineering and cyber security. *Journal of Optical Communications and Networking*, 11(1):A84–A95, 2019.

[72] Princeton Cabernet Group. Smartcookie. https://github.com/Princeton-Cabernet/p4-projects/tree/master/SmartCookie, 2023. Accessed: 2025-03-30.

[73] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[74] J. Qi, W. Li, T. Yang, D. Li, and H. Li. Cuckoo counter: A novel framework for accurate per-flow frequency estimation in network measurement. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–7. IEEE, 2019.

[75] P. Reviriego and S. Pontarelli. Perfect cuckoo filters. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, pages 205–211, 2021.

[76] M. E. Şahin and M. Demirci. Conpoolubf: Connection pooling and updatable bloom filter based syn flood defense in programmable data planes. *Computer Networks*, 231:109802, 2023.

[77] D. Scholz, S. Gallenmüller, H. Stubbe, B. Jaber, M. Rouhi, and G. Carle. Me love (SYN-)Cookies: SYN flood mitigation in programmable data planes. *arXiv preprint arXiv:2003.03221*, 2020.

[78] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle. Cryptographic hashing in P4 data planes. In *Proc. of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'19)*, pages 1–6, Cambridge, UK, September 2019.

[79] Q. Su, S. Wu, Z. Niu, R. Shu, P. Cheng, Y. Xiong, C. J. Xue, Z. Liu, and H. Xu. Meili: Enabling SmartNIC as a service in the cloud. *arXiv preprint arXiv:2312.11871*, 2024.

[80] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[81] The FreeBSD Project. Freebsd handbook: Firewalls. `https://docs.freebsd.org/en/books/handbook/firewalls/`, 2024. Accessed: 2025-04-19.

[82] The Linux Kernel Organization. Bpf maps — linux kernel documentation. `https://www.kernel.org/doc/html/latest/bpf/maps.html`, 2025. Accessed: 2025-03-30.

[83] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[84] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, pages 122–135, 2017.

[85] Wikipedia contributors. ABC Software Metric — Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/ABC_Software_Metric`, 2025. [Online; accessed 12-May-2025].

[86] Q. Xiao, H. Wang, and G. Pan. Accurately identify time-decaying heavy hitters by decay-aware cuckoo filter along kicking path. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2022.

[87] M. Xie, Q. Chen, T. Wang, F. Wang, Y. Tao, and L. Cheng. Towards capacity-adjustable and scalable quotient filter design for packet classification in software-defined networks. *IEEE Open Journal of the Computer Society*, 3:246–259, 2022.

[88] Xilinx. Alveo U25 SmartNIC Product Image. `https://web.archive.org/web/20200511172852im_/https://www.xilinx.com/content/dam/xilinx/imgs/kits/alveo-u25-hero-of-card.jpg`, 2020. Image accessed via Internet Archive on 2025-04-22.

[89] Xilinx. open-nic: An OpenNIC platform based on Xilinx FPGA. `https://github.com/Xilinx/open-nic`, 2021. Accessed: 2025-04-22.

[90] Xilinx, Inc. Alveo U25 Product Brief. `https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u25-product-brief.pdf`, 2020. Accessed: 2025-02-05.

[91] Q. Yan, F. R. Yu, Q. Gong, and J. Li. Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges. *IEEE Communications Surveys & Tutorials*, 18(1):602–622, 2016.

[92] S. Yoo, X. Chen, and J. Rexford. SmartCookie: Blocking Large-Scale SYN floods with a Split-Proxy defense on programmable data planes. In *Proc. of the 33rd USENIX Security Symposium (USENIX Security ’24)*, pages 217–234, Philadelphia, PA, August 2024.