BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FLEXIBLE AND EVENT-TRIGGERED DATA PLANE

FLEXIBILNÍ A UDÁLOSTMI ŘÍZENÁ DATOVÁ CESTA

PHD THESIS
DISERTAČNÍ PRÁCE

AUTOR — **Ing. JAN KUČERA**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. JAN KOŘENEK, Ph.D.**
ŠKOLITEL

BRNO 2024

Abstract

The dissertation thesis deals with the topic of high-speed network monitoring and security. It focuses on optimizing network security monitoring applications using programmable data planes. It exploits the specific network traffic characteristics, the flexibility of network data plane programmability and proposes multiple hardware architectures and algorithms corresponding to the required network traffic processing. The thesis introduces a new concept of flexible heavy flow-based acceleration for network monitoring and intrusion detection, an event-trigged push-based approach for detecting high-volume traffic clusters, and specific window-based network traffic aggregation and feature extraction, particularly for DDoS defense. The architectures use probabilistic data structures and techniques to ensure high detection performance and throughput with low resource requirements. The investigated concepts demonstrate the network data plane optimizations for three different use cases: (1) the flow-based network monitoring, (2) in-band network events detection, and (3) real-time DDoS mitigation. The research findings brought significant improvements in network security monitoring. Informed packet pre-filtering considerably reduces the amount of network traffic to be analyzed and achieves higher throughput. The event-triggered approach addresses application scalability issues and reduces data-control plane communication overhead. Compared to the other state-of-the-art solutions, the event-triggered approach can save a significant amount of control plane traffic of more than two orders of magnitude. Furthermore, the presented implementation results are utilized in DDoS Protector, an accelerated DDoS protection solution that has been commercialized and protects the Czech academic network.

Keywords

High-speed networks, monitoring, security, IDS, DDoS attack mitigation, hardware acceleration, optimization, programmable data plane, FPGA, P4.

Abstrakt

Disertační práce se zabývá tématem monitorování a bezpečnosti vysokorychlostních počítačových sítí. Zaměřuje se na optimalizaci aplikací pro bezpečnostní monitorování sítě pomocí programovatelné datové cesty. Využívá specifických vlastností síťového provozu, flexibility programovatelnosti datové cesty a implementuje řadu hardwarových architektur a algoritmů odpovídajících požadovanému zpracování síťového provozu. V rámci práce byl navržen nový koncept flexibilní akcelerace monitorování sítě a detekce škodlivého provozu založený na těžkých tocích, přístup událostmi řízené detekce významných shluků síťového provozu a specifická agregace síťového provozu založená na principu klouzavého okna a extrakce příznaků pro ochranu proti DDoS útokům. Vytvořené architektury využívají pravděpodobnostní datové struktury a techniky pro dosažení vysoké míry úspěšnosti detekce, vysoké propustnosti a nízkých nároků na hardwarové zdroje. Navržené přístupy jsou demonstrovány pro tři různé aplikace: (1) monitorování sítě na bázi síťových toků, (2) vestavěná detekce síťových událostí a (3) potlačení DDoS útoků v reálném čase. Výsledky výzkumu přináší významné zlepšení v oblasti bezpečnostního monitorování sítě. Řízená předfiltrace paketů značně snižuje množství síťového provozu, který musí být nutně analyzován, a dosahuje tak vyšší propustnosti. Přístup založený na událostmi řízené detekci událostí řeší problémy se škálovatelností aplikací a snižuje režii komunikace mezi datovou a řídicí cestou. Ve srovnání s ostatními state-of-the-art přístupy redukuje množství řídicí komunikace o více než dva řády. Představené implementační výsledky práce jsou navíc integrovány jako součást systému DDoS Protector, což je akcelerované řešení ochrany proti DDoS, které bylo jako výsledek výzkumu komercializováno a v současné době chrání českou akademickou síť.

Klíčová slova

Vysokorychlostní sítě, monitorování, bezpečnost, IDS, potlačení DDoS útoků, hardwarová akcelerace, optimalizace, programovatelná datová cesta, FPGA, P4.

Reference

KUČERA, Jan. *Flexible and Event-Triggered Data Plane.* Brno, 2024. PhD thesis. Brno University of Technology, Faculty of Information Technology.

Rozšířený abstrakt

Cílem práce je akcelerace síťových aplikací a návrh vhodných datových struktur, architektur a algoritmů, které využijí programovatelnosti datové cesty k dosažení vyšší výkonnosti. Koncepty navržené v této práci využívají specifických vlastností síťového provozu, agregaci provozu na principu oken nebo extrakci příznaků ze síťového provozu a selektivně redukují množství informací a paketů, které je třeba při analýze zpracovat. Tím snižují režii komunikace mezi datovou a řídicí cestou a umožňují optimalizovat výkon aplikace.

Namísto přímé akcelerace časově kritických operací, jako to dělají jiná řešení, využívá představený koncept akcelerace alternativní přístup. Zaměřuje se na redukci (předfiltraci) síťového provozu, který musí být aplikací nutně zpracován. Přístup využívá právě specifických vlastností síťového provozu k dosažení akcelerace. Na základě těchto vlastností je totiž možné rozhodnout, které pakety jsou pro detekci bezpečnostních událostí významné, tj. které musí být analyzovány a které mohou být vhodně redukovány, agregovány nebo zahozeny. Tímto způsobem je možné soustředit dostupné výpočetní zdroje aplikace pouze na relevantní část síťového provozu a dosáhnout její akcelerace. Navržený přístup vychází ze dvou předpokladů: (1) Největší část síťového provozu (z hlediska objemu) je přenášena velmi malým počtem relativně velkých síťových toků. (2) Z hlediska bezpečnosti je nejdůležitější analýza zejména počáteční paketů síťových toků. Ty nesou nejvíce užitečných informací (hlavičky paketů) potřebných k detekci nežádoucího síťového provozu nebo jiných bezpečnostních událostí. Současně předpokládáme, že při velkém zatížení síťové aplikace, např. systému IDS (Intrusion Detection System), jsou příchozí pakety nekontrolovaně zahazovány na jeho vstupu v důsledku zaplnění vstupních paketových bufferů na síťové kartě. Takový mechanismus zahazování paketů přitom výrazně a negativně ovlivňuje kvalitu detekce, protože všechny pakety jsou zahazovány se stejnou pravděpodobností bez ohledu na jejich důležitost z hlediska samotné detekce událostí.

Práce ukazuje, že výrazně lepších výsledků lze dosáhnout řízenou filtrací paketů jednotlivých síťových toků. Navržený přístup proto umožňuje plnohodnotné softwarové zpracování pouze prvních N paketů každého síťového toku. Větší síťové toky jsou považovány za těžké (heavy) a následující pakety takových toků lze efektivně agregovat nebo zahodit. Při vysokém zatížení aplikace lze pak selektivně zpracovat pouze nejdůležitější data nezbytná pro bezpečnostní analýzu a méně významnou část síťových toků z důvodu omezeného výkonu řízeně zahodit bez výrazného vlivu na kvalitu detekce.

Úvodní část disertační práce popisuje navržený koncept a představuje odpovídající hardwarovou architekturu pro platformu FPGA. Jelikož se původní návrh zaměřoval výhradně na použití pro monitorování sítě, je koncept pojmenován jako *Softwarově definované monitorování (SDM)*. Systém je demonstrován na příkladu analýzy aplikačních protokolů a dále ověřen na reálných síťových datech v rámci experimentálního nasazení na páteřní lince národní akademické sítě. Další část práce se potom specificky věnuje použití tohoto přístupu také k akceleraci systémů IDS (Intrusion Detection System).

Práce se v rámci navazujícího výzkumu blíže zabývá možnostmi rozdělení úlohy monitorování sítě mezi hardwarové a softwarové prostředky a analyzuje možnou režii v komunikaci mezi datovou a řídicí cestou. Původní motivací bylo zejména snížení latence při detekci těžkých síťových toků přesunem této úlohy do hardwarové architektury. Ukázalo se však, že se jedná o mnohem obecnější výzkumnou otázku. Současná řešení využívající datovou cestu k akceleraci monitorování sítě totiž implementují různé pravděpodobností datové struktury k agregaci čítačů síťových toků a poskytují je softwarové aplikaci běžící na kontroleru sítě, která tyto čítače periodicky vyčítá. Během analýzy jsme zjistili, že načtení

velkého množství čítačů a složité datové struktury z hardwarové implementace je časově velmi náročné a v daném kontextu vytváří zpoždění řádově jednotek sekund. Rychlost reakce na bezpečnostní události je tak silně závislá na schopnosti aplikace periodicky získávat tyto datové struktury z datové cesty. To však přináší zpoždění, které je pro specifické případy použití, např. detekci DDoS (Distributed Denial of Service) útoků, nežádoucí.

V této souvislosti byl v práci navržen zcela odlišný přístup pro optimalizaci komunikační režie mezi datovou a řídicí cestou (hardwarovou datovou strukturou a softwarovou aplikací). Cílem bylo za pomoci programovatelnosti datové cesty transformovat dosavadní pasivní přístup (poll-based) na aktivní systém (push-based), který je schopný iterativně a zcela autonomně detekovat několik typů síťových událostí a až následně informovat softwarový kontroler. Pro demonstraci tohoto přístupu byly vytvořeny dvě hardwarové architektury a datové struktury Unroller a Elastic Trie. Unroller umožňuje identifikaci směrovacích smyček v reálném čase. Elastic Trie se pak zaměřuje na detekci tzv. heavy-hitters (zdrojů těžkých síťových toků) nebo superspreaders (zdrojů velkého množství síťových toků) a dokáže odhalit významné změny ve vzoru síťové komunikace (change detection). Výrazně přitom redukuje režii komunikace mezi datovou a řídicí cestou. Detekce takových událostí má z bezpečnostního hlediska velký význam pro odhalení nežádoucích aktivit jako je skenování sítě, šíření malware nebo útoků DDoS. Událostmi řízený přístup implementovaný v datové cestě proto může přispět k rychlejší detekci a reakci na takové aktivity.

Zejména v posledních letech výrazně vzrostlo množství DDoS útoků. Ty jsou dnes jedním z nejrozšířenějších typů kybernetických útoků. Zbytek práce se proto podrobněji zaměřuje nejen na rychlou detekci, ale i potlačení DDoS útoků. Jsou představeny a implementovány tři strategie potlačení SYN Flood útoků v reálném čase. Metody umožňují ověřit TCP klienty pomocí kryptograficky zabezpečené informace vložené do paketů během ustavování TCP spojení. Tím lze útočníkům zabránit v podvržení zdrojové IP adresy. Navazující část potom představuje koncept agregace provozu na bázi oken a příznaků ze síťového provozu pro účely detekce DDoS v reálném čase s využitím strojového učení.

Publikované časopisecké a konferenční příspěvky zahrnuté v této práci prezentují několik hardwarových architektur a softwarových algoritmů pro optimalizaci síťových aplikací s využitím programovatelnosti datové cesty a demonstrují je na třech případech užití: (1) *monitorování sítě na bázi síťových toků*, (2) *vestavěná detekce síťových událostí* a (3) *potlačení útoků DDoS v reálném čase*. Představené výstupy a implementační výsledky práce jsou navíc integrovány jako součást systému DDoS Protector, což je akcelerované řešení ochrany proti DDoS útokům.

Flexible and Event-Triggered Data Plane

Declaration

Hereby, I declare that this dissertation thesis was prepared as an original author's work written under the supervision of doc. Ing. Jan Kořenek, Ph.D. All the relevant information and sources used during the preparation of this thesis are correctly cited and included in the list of references.

. . . .
Jan Kučera
August 27, 2024

Acknowledgements

I would like to thank several people who have supported me throughout my Ph.D. studies. First of all, I would like to thank my supervisor, doc. Ing. Jan Kořenek, PhD., for his invaluable advice, suggestions, and constant support. My thanks go to Dr. Gianni Antichi and Dr. Andrew W. Moore for hosting me at the University of Cambridge and to the people I had the pleasure of meeting at the Computer Laboratory during my stay. I am incredibly thankful to Dr. Gianni Antichi for his continued mentorship during our collaboration in the following years after my return to Brno. I am grateful to my co-authors, my past and present colleagues from the TMC department, and the DDoS Protector team at CESNET for their excellent cooperation on various research projects, with special thanks to Ing. Martin Žádník, Ph.D., for guiding me through several research efforts. I thank my parents and family for their loyal support during my studies and my friends for their frequent, annoying, and strictly forbidden questions about the progress of this thesis.

The work presented in this thesis was supported by the Technology Agency of the Czech Republic (under the project TH01010229), by the Ministry of the Interior of the Czech Republic (under the security research projects VI20192022137, VB01000015 and VB02000066), by the Ministry of Education, Youth and Sports of the Czech Republic (under the CESNET E-Infrastructure – Modernisation project EF16_013/0001797 and e-INFRA CZ project LM2023054), and by the internal Brno University of Technology projects FIT-S-17-3994, FIT-S-20-6309, and FIT-S-23-8141.

Contents

1 Introduction **2**
- 1.1 Research Area . . . 3
- 1.2 Research Objectives . . . 4
- 1.3 Thesis Outline . . . 4

2 State of the Art **5**
- 2.1 Programmable Data Plane . . . 5
- 2.2 Network Traffic Processing . . . 7
- 2.3 Network Monitoring and Telemetry . . . 10
- 2.4 Real-time Intrusion Detection . . . 12

3 Research Summary **16**
- 3.1 Research Process . . . 16
- 3.2 Papers . . . 21
  - 3.2.1 Paper I . . . 21
  - 3.2.2 Paper II . . . 22
  - 3.2.3 Paper III . . . 23
  - 3.2.4 Paper IV . . . 24
  - 3.2.5 Paper V . . . 25
  - 3.2.6 Paper VI . . . 26
- 3.3 List of Publications . . . 27

4 Research Results **29**
- 4.1 Flexible Network Monitoring . . . 29
- 4.2 Event-triggered Data Plane . . . 31
- 4.3 Real-time DDoS Mitigation . . . 35

5 Discussion and Conclusions **37**
- 5.1 Future Work . . . 38

Bibliography **40**

A Included Papers **47**
- A.1 Paper I . . . 48
- A.2 Paper II . . . 61
- A.3 Paper III . . . 70
- A.4 Paper IV . . . 79
- A.5 Paper V . . . 93
- A.6 Paper VI . . . 100

Chapter 1

Introduction

In February 1992 (32 years ago), the Czech Republic, former Czechoslovakia, was officially connected to the Internet [1], the worldwide computer network. The first connection was 19.2 kbps and led from the Czech Technical University in Prague to the University of Linz in Austria. Since then, we have witnessed an extensive global and worldwide growth of computer and telecommunication networks. The speed of these networks has grown over 10 million times. Over the past decades, the Ethernet technology has exponentially evolved, with a new higher standard introduced almost every seven years [32]. The latest 800 GbE standard was approved in February 2024, expecting the 1.6 TbE standard later in 2026.

Such development is driven by the increasing demand for higher bandwidth, especially at internet service providers, in data centers and exchange points. It follows the growth of the Internet of Things (IoT) and services, such as streaming platforms, online games, and cloud applications, as well as the shift of many activities to the online environment, including remote work, online education, and virtual meetings, leading to increases in network traffic. It has significant performance implications for the current network devices and their architectures. They need to be faster and more powerful to avoid becoming a bottleneck.

The nature of networks has also changed. Along with the development of faster speeds, various cyber security threats have evolved. These threats include DoS (Denial of Service) attacks, leaks of sensitive personal data, the spread of malware, and the dissemination of fraudulent emails, messages, or phone calls. As a result, considerable effort in packet processing is required to enable the overall reliability and protection of computer networks, making monitoring and network security a crucial application.

Network applications can be implemented on multiple hardware or software platforms. However, the computational complexity of security systems makes it challenging to meet the performance requirements of modern high-speed networks. At the same time, the architectures must be future-proof and flexible enough to allow rolling out new features quickly to accommodate monitoring and detection methods for newly emerging security threats. Although network devices can achieve high-speed performance using specialized hardware, i.e., ASICs (Application-specific Integrated Circuits), proprietary solutions suffer clearly from insufficient flexibility. Introducing a new functionality, such as a new network protocol, may necessitate a device hardware upgrade, which can be expensive. In addition, designing and manufacturing such complex custom silicon can be time-consuming and incur high economic costs. Therefore, flexibility in software implementation is highly desirable. On the other hand, recent trends like NFV (Network Function Virtualization) and SDN (Software-Defined Networking) can offer the required flexibility but may not scale to high

throughputs. Pure software processing has obvious performance issues due to using general-purpose CPUs instead of specialized network hardware.

This situation motivated the introduction of the data plane programmability [57]. The programmable data plane refers to brand-new network devices that allow network traffic processing to be dynamically and programmatically changed. Due to the emergence of new and more general software and hardware-based platforms, man can reconfigure today's data plane devices, mainly SmartNICs (Smart Network Interface Cards) and programmable switches, entirely to support future functionalities without additional hardware upgrades while maintaining the same performance as its fixed-function counterpart. SmartNICs usually implement programmability using FPGAs (Field Programmable Gate Arrays) or NPUs (Network Processing Units). Meanwhile, high-performance programmable switches are built on flexible switching chip architectures implemented as ASICs. Such evolution enables a new spectrum of network applications to be implemented in the data plane.

State-of-the-art solutions [36, 51, 98, 100, 79, 49] utilize data plane programmability to accelerate network security and monitoring tasks. They implement various probabilistic data structures to aggregate network traffic and provide them to a software controller. The controller then periodically polls the structures to get insights into the network behavior. Although such solutions improve network monitoring capabilities and overall performance, compared to earlier approaches, they still depend on polling complex data structures at short intervals [23]. Such an architecture has serious drawbacks, especially in cases where detection latency is essential. It relies on a communication channel between the data and control plane, which can lead to considerable delays. Therefore, there is a space for a new solution that improves the state-of-the-art in this area and resolves the scalability issues. It could be improved by transforming the data plane from a passive monitoring infrastructure to an active system capable of detecting network events autonomously, tracking the responsible network traffic, and informing the control plane only when specific conditions are met.

1.1 Research Area

The dissertation topic is network monitoring and security. It focuses on optimizing network security applications using a programmable data plane. The assumptions are to exploit the specific network traffic characteristics, the event-trigged push-based approach, and specific window-based network traffic aggregation and feature extraction to improve network monitoring and intrusion detection systems with respect to the needs of high-speed computer networks. The aim is to exploit the flexibility of data plane programmability to design hardware architectures and algorithms corresponding to the required network traffic processing. The focus is also on searching for new suitable data structures to optimize and implement the task efficiently in the data plane and divide it beneficially between the data and control plane. The thesis introduces multiple hardware architectures and algorithms for network data plane-based application acceleration and optimization. It proposes a concept of flexible heavy flow-based acceleration for network monitoring and intrusion detection and a concept of in-band network events detection and packet feature extraction, particularly for real-time DDoS mitigation.

1.2 Research Objectives

Based on the analysis of the research and the current state-of-the-art in the area of network traffic monitoring and security event detection in high-speed networks, we formulate the following working hypothesis for this thesis:

> *By leveraging specific network traffic characteristics, probabilistic data structures, and an event-triggered approach, we can resolve applications' scalability issues and significantly reduce (up to several orders of magnitude) their implementation overheads. It allows us to optimize network monitoring and security tasks effectively without hindering their flexibility, achieving higher performance, lower resource requirements, and/or better detection quality than existing state-of-the-art solutions.*

The main research objectives for this thesis are:

- To propose appropriate concepts that leverage the flexibility of data plane programmability to accelerate and optimize various network security and monitoring applications in high-speed networks.
- To exploit specific network traffic characteristics and to utilize probabilistic data structures and techniques to ensure the proposed solution maintains high throughput and performance while maintaining low resource requirements.
- To design hardware architectures and software algorithms corresponding to the required network traffic processing that implement the proposed concepts efficiently in the data plane and divide the task beneficially between the data and control plane.
- To synthesize the proposed hardware architectures for selected data plane platforms and to evaluate platform resource occupancy and achieved performance metrics.
- To analyze the achievable performance benefits and event detection quality of selected applications when utilizing the proposed architectures and algorithms and to compare the results with the other state-of-the-art solutions.

1.3 Thesis Outline

The thesis is composed as a collection of papers, and its research contribution is therefore presented by six peer-reviewed papers included in the thesis. These publications are attached in their original format at the end of the thesis as Appendix A. The thesis is organized as follows. This chapter (Chapter 1) motivated the thesis, introduced the research area, and summarized the research objectives. Chapter 2 surveys the state-of-the-art and provides the relevant background information for the research process. It includes the aspects of high-speed network traffic processing, and it discusses data plane programmability in the context of currently used approaches and related works to optimize network applications. Chapter 3 presents conducted research. Special attention is dedicated to informed network traffic reduction and data-control plane communication overhead. It provides an overview of the selected papers included in the thesis and summarizes their contribution. Finally, Chapter 5 concludes the thesis and its outcomes and proposes possibilities for future research directions.

Chapter 2

State of the Art

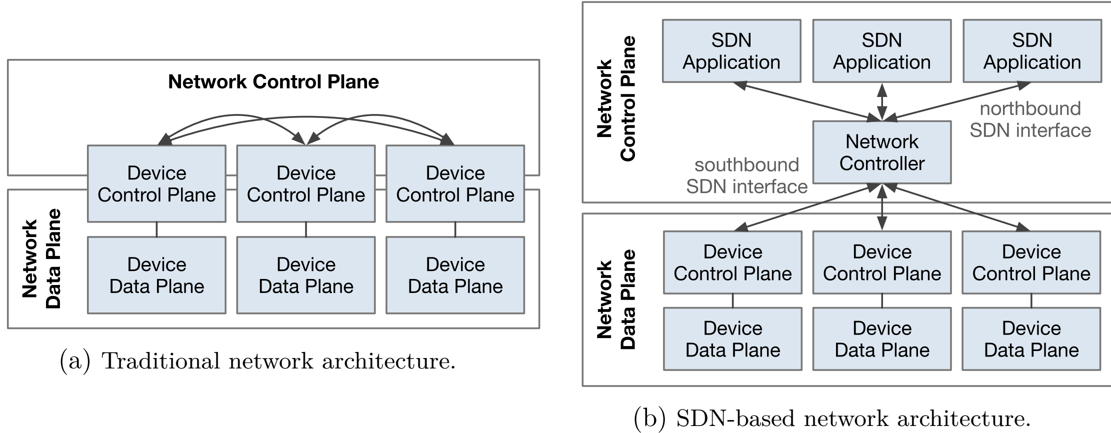
This chapter surveys the current state-of-the-art and provides relevant background information required for the research process. Section 2.1 describes the evolution and demands in this area towards data plane programmability, flexibility, and high performance. Section 2.2 discusses high-speed network traffic processing. Section 2.3 covers principles of computer network monitoring. Finally, section 2.4 presents the needs and the area of intrusion detection systems.

2.1 Programmable Data Plane

As the landscape of high-speed networks constantly evolves, current state-of-the-art network monitoring and security systems have difficulties catching up with rapid development. Although proprietary devices can achieve high performance using specialized hardware components, they incur high economic costs. For data centers, complex custom silicon available at high-end routers is too expensive. More importantly, traditional networking equipment suffers from insufficient flexibility in terms of functionality. Deploying a new feature, e.g., a newly emerged network protocol, can require a device hardware upgrade, which is very slow and costly, and it forces vendors to support the feature only when it becomes widely requested. On the other hand, pure software-based solutions following the recent trends in networking, e.g., Network Function Virtualization (NFV) and Software-Defined Networking (SDN), are highly flexible, free network operators from waiting to vendors when rolling out new features, but lead to scalability issues. The server-based packet processing incurs low performance and high overheads due to using general-purpose CPUs rather than specialized network hardware. However, the emergence of data plane programmability, mainly SmartNICs (Smart Network Interface Cards) and programmable switches, introduces new possibilities to fill this gap. The topic of programmable data planes is extensive. The following sections provide only summary information mainly based on [57, 88, 56, 49, 103].

Traditional and Software-defined Networking

As illustrated in Figure 2.1a, the design of traditional networking devices was always logically divided into two parts: (1) the fast path (device data plane) and (2) the slow path (device control plane). The fast path performs time-critical operations like packet parsing, classification, modification, and forwarding and applies policies executed on every incoming packet, which require high performance. The combination of classification and subsequent processing based on matched packet headers is usually called match-action processing. On



(a) Traditional network architecture.

(b) SDN-based network architecture.

Figure 2.1: Evolution of network architecture towards SDN (adoped from [57]).

the other hand, the slow path is responsible for managing routing information and setting those packet processing policies, i.e., where to forward a packet and how to rewrite its headers, configuring it for the fast path. Moreover, the slow paths of individual network devices interact with each other through routing protocols. Thus, together in a distributed manner, they create the global network-level forwarding policy (network control plane).

In contrast, the Software-Defined Networking (SDN) [82] approach separates the control plane functionality from the data plane and individual devices. It concentrates it in a single network-wide control plane entity, a centralized controller. The architecture is depicted in Figure 2.1b. The SDN eliminates the need for individual switches to implement the logic to maintain packet forwarding policies locally. Thus, they no longer run routing protocols to build routing tables. Instead, they receive these policies from the controller, which directly controls the state of the device data planes. The device control plane is, however, not fully extracted. A part of the device control plane is still present, and it is responsible for device management and establishing a control channel towards the controller. The communication between the controller and devices uses standardized southbound interface protocols such as OpenFlow [54], or P4Runtime [66]. In contrast, the northbound interface ensures communication between the controller and user applications. It enables the network-wide implementation of services, such as unicast routing, multicast routing, access control, quality of service, and other management and security monitoring applications.

Data Plane Programmability

SDN has revolutionized network design and management. It has made device functionality more flexible and dynamic. The centralization of the network state has given network managers more freedom to implement automated programs to control and monitor their network from a single logical point independently, without waiting for device vendors to integrate and roll out new control plane features. However, the traditional fixed data plane functionality has remained unchanged and tightly bound to the device hardware or software-based packet processing implementation. This means that such devices only understand a fixed set of protocol header fields during their lifetime, which affects most data plane operations. For example, the format of the match-action table entries is device-specific, and the device vendor also sets the types of processing actions that can be applied and their order. As a result, it is impossible to apply IP routing lookup on packets encapsulated

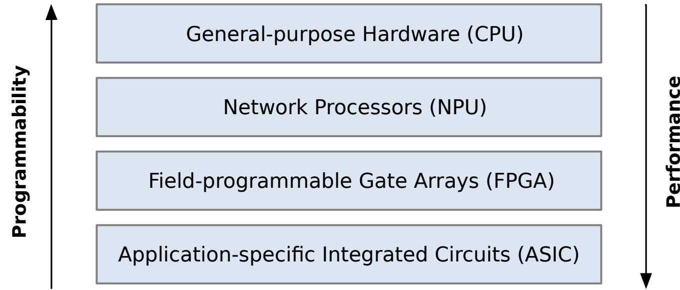


Figure 2.2: Comparing programmability and performance of data plane platforms.

in VXLAN, MPLS, or other tunnels. Even the available monitoring information exported from the data plane cannot be changed.

This inflexibility motivated the introduction of the data plane programmability. Thanks to the emergence of more general hardware and software platforms, today's data plane devices can be entirely reconfigured. In this context, *programmable data plane* refers to a new network device category that allows the packet parsing and match-action processing functionality to be dynamically and programmatically changed [57]. At the same time, domain-specific languages, e.g., Programming Protocol-independent Packet Processors (P4) [11, 65] or Network Programming Language (NPL) [63], provide a way to describe how data plane devices should process packets. They allow the definition of five essential aspects of packet processing: (1) header formats, (2) packet parsing, (3) table specification, (4) action specification, and (5) control flow. Thus, data plane programmability includes parsing new protocols, matching dynamically defined header fields, and exposing new APIs to the control plane. Thanks to that, brand-new applications can be realized in the data plane, including user-defined in-band network monitoring and telemetry.

2.2 Network Traffic Processing

Traffic processing of the data plane programmable devices can be realized on top of multiple hardware or software platforms. For example, a software data plane device runs the data plane processing on a commodity CPU, a processing pipeline of a SmartNIC can be implemented in a network processor (NPU) or an FPGA (Field Programmable Gate Array), and a programmable switch is built on top of an ASIC (Application-specific Integrated Circuit). The relationship between the programmability and the performance of these data plane platforms is depicted in Figure 2.2.

Software-based Processing

Software-based traffic processing uses general-purpose CPUs like x86 or ARM architectures in commodity servers. It mainly includes copying the packet's data from a NIC buffer to the CPU and processing it in terms of parsing, classification, filtering, and modification before moving the data back to another NIC buffer. The evolution in server-based packet processing is surveyed in [57, 56]. Current software solutions are divided into two categories depending on whether packets are processed at the kernel level or in userland.

Linux-based systems provide kernel-space technologies such as eBPF and XDP [13, 35]. XDP enables the injection of an eBFP program from userspace. After all the security properties are satisfied, the program executes in the kernel context at the earliest level

of the Linux networking stack directly upon receiving a packet after the NIC driver RX queues. Thus, it improves performance compared to the standard kernel data path. In contrast, fast packet I/O frameworks, e.g., DPDK [29] or Netmap [71], rely on bypassing the operating system and mapping hardware buffers directly to userspace memory instead of using standard kernel data path. Compared to standard sockets, it effectively eliminates the overhead of packet copies and the cost of context switching. The method provides a significant performance benefit but requires the userspace to take over the exclusive ownership of the NIC. The userspace application processes all the received packets. It cannot utilize kernel networking and must implement all the packet processing functionality, e.g., TCP/IP stack, routing tables, etc., on its own. If some packets belong to the operating system, the application must reinsert them back into the kernel. Packets are usually distributed across multiple CPU cores and processed in batches to improve data locality and reduce costs of accessing ring buffers or other resources.

Network Interface Cards

Network interface cards [55, 38, 62, 57] are used further in the previous approaches to boost the performance by offloading a part of the traffic processing workload from the host CPU. They are usually connected to the host server via the PCI Express system bus. A common feature of their packet capture interface is Direct Memory Access (DMA), which distributes the data flow to multiple software channels for efficient processing by multiple CPU cores. They provide pre-defined fixed hardware functions, mainly basic Ethernet frame processing, e.g., MAC address filtering, checksum offloads, timestamp assignment, receive, transmit segmentation, and further transferring the data stream to and from the host memory, where the complete software processing by a specific application takes place. Some NICs also offer stateless matching on selected packet fields to apply simple actions such as packet modification, allow, or drop.

SmartNICs [96, 62, 57] are network accelerators that further bring data plane programmability and flexibility to this segment. The programmable functionality is commonly realized on top of an FPGA (Field Programmable Gate Array) or an NPU (Network Processing Unit). Well-known network cards based on FPGA technology include ReflexCES boards [69], Napatech SmartNICs [80], Silicom FPGA cards [70] or NetFPGA SUME [105]. They provide fully custom packet processing, e.g., packet sampling, trimming, accurate timestamps, packet classification, stateful flow filtering, pattern matching and TCP or TLS offloads. On the other hand, they have limited on-chip memory or general computing capabilities to offload all tasks completely. Offloading various tasks may also require the host to compile and inject a new code to reconfigure the hardware co-processor. They are programmed using low-level HDL languages like VHDL, a subset of general-purpose HLS languages like C, or data plane-oriented languages such as eBPF [13] or P4 [65]. However, there is no current standard for interaction with SmartNICs. While some SmartNICs support the RTE Flow [29] interface, often only proprietary methods exist [56]. Thus, it is challenging to support multiple vendors.

Programmable Switches

While SmartNICs enable flexibility at the network's edge, programmable switches introduce the data plane programmability in the network core. High-performance programmable switches are built on top of two major flexible switching chip architectures, Reconfigurable Match-action Tables (RMT) [10] and Protocol-independent Switch Architecture (PISA)

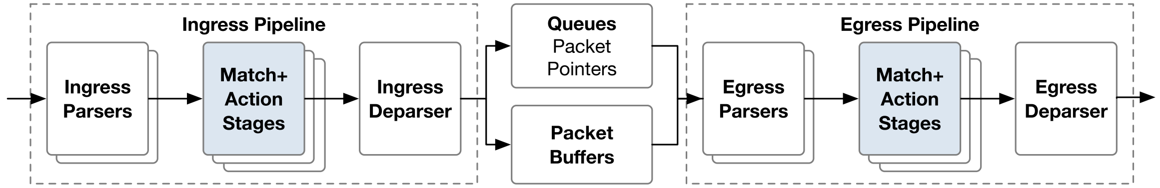


Figure 2.3: Switch architecture based on match-action tables (adoped from [57]).

[16], are implemented as ASICs, e.g., Intel Tofino [39] or Broadcom Trident [91] chips, always running at wire speed. Figure 2.3 illustrates the typical switch architecture. It usually consists of multiple (ingress and egress) pipelines of programmable match-action stages. At the beginning of a pipeline, a packet enters a programmable parser that dissects the packet buffer into individual protocol headers. The match-action stages then classify the packet based on a subset of extracted packet header fields. Classification rules are stored alongside the pipeline in banks of on-chip memory. The stage can have different matching capabilities depending on the type of lookup table. For example, an exact match hashing table is implemented in SRAM, while a wildcard matching table uses TCAM. The corresponding packet action then performs simple calculations using Arithmetic Logical Units (ALUs), updates the internal state, or instructs the switch to encapsulate/decapsulate, drop, or forward the packet, rewrite its content, or continue its processing in a subsequent match-action stage. Before leaving the pipeline, the packet enters a deparser, serializing the individual packet headers again before sending the packet to an output interface or another pipeline.

Data Plane Description

The programmer describes the data plane program as a sequence of lookup tables organized into a hierarchical structure, which is later compiled into the underlying hardware or software target configuration. The content of lookup tables populated via a control interface later defines the packet processing behavior at runtime. Thanks to the programmability, the data plane can be reconfigured to support future functionalities without additional hardware upgrades while maintaining the same power consumption and cost as fixed-function switches. Programmable network devices offer a significant advantage to network operators. They provide the ability to programmatically modify the low-level data plane functionality, enabling support for novel protocols, implementing custom forwarding logic, or even introducing entirely new applications in hardware. This can all be achieved at speeds of Tbps, enhancing the efficiency and flexibility of network operations.

Although languages like P4 are high-level from the hardware perspective, they are still very low-level for directly implementing complex network security applications. Programmable switches based on RMT architectures enable stateful processing but use a very small amount of SRAM that persists across consecutive packets. Moreover, they restrict the computational model to fulfill the high throughput requirements. For instance, the number and the complexity of pipeline stages or the number of memory accesses are constrained. Unfortunately, this fact makes it challenging to implement sophisticated network monitoring and security applications [49, 79, 51, 98, 103, 50] in the data plane.

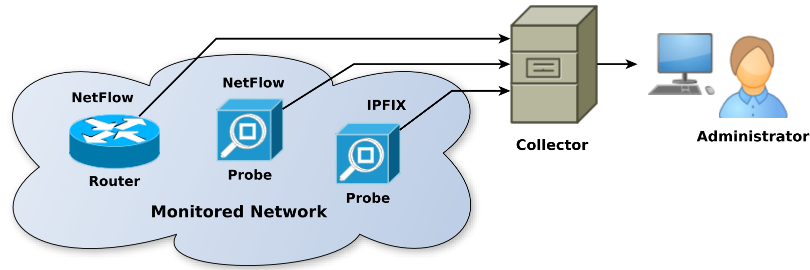


Figure 2.4: Architecture of network flow monitoring system.

2.3 Network Monitoring and Telemetry

The network measurement is the foundation of network management, security and control. According to [88], we can identify three phases in the history of computer network development: (1) traditional network measurement developed since 1995; (2) software-defined measurement developed with SDN networks since 2008; and (3) network telemetry recognized with the advent of programmable data planes since 2015.

Traditional Network Measurement

The traditional network measurement is divided into active and passive schemes. The active measurement is performed by injecting packet probes into a network and analyzing their path or latency, e.g., using ping and traceroute tools. The passive measurement (also known as network monitoring) includes polling network interface counters, packet sampling, or collecting aggregated records of network flows [34] using well-known protocols such as SNMP [17], sFlow [78], NetFlow [20] or IPFIX [21].

Due to the ease of deployment, traditional network flow monitoring methods are still widely used for network security and management applications. An example of a passive network flow monitoring system is shown in Figure 2.4. Network monitoring is based on two types of network elements. The probe (or the exporter) collects statistics (records of network flows) and sends them using NetFlow [20] or IPFIX [21] protocols. The collector receives these statistics (even from several probes simultaneously) and stores or forwards them for further analysis. The probe can be standalone or implemented as a part of other devices, such as routers. Flow records at the collector are stored for a certain period to enable tracing back the incidents later reported to the network administrator.

The network flow is defined as a sequence of IP packets that pass through a certain point in a network during a specific period and have some common property [21]. The property is based on the packet header entries or a characteristic of the packet itself, e.g., its length or arrival time. Five items are often used – source and destination IP address, source and destination port, and transport protocol identification. The communication direction is not included. Thus, a bidirectional connection is always made up of two flows. Additional parameters are subsequently measured for each flow as part of the monitoring process. For example, the number of packets that the flow carries (flow size), the number of bytes transferred (flow volume), the time interval between the first and last packet (flow duration), or the set of TCP flags that were set during the connection.

Network flow monitoring represents a resonable tradeoff regarding the volume of data collected and the amount of helpful information. However, some network security appli-

cations require network monitoring at the upper layers. As the network functionality is shifting towards the application layer (L7), the following trend in network flow monitoring is to provide enriched flow records extended by some extra information from application protocol headers. Thus, to maintain the detection quality of new security threats, the IPFIX protocol also enables the inclusion of fields from HTTP, DNS, or other L7 protocols, e.g., URLs or domain names, in the flow records.

Software-defined Measurement

The later advent of SDN networks improved the flexibility of network applications. Several new software-defined approaches focusing on performance or function measurements, such as bandwidth, packet loss, latency analysis, and path tracing detection, were proposed [88]. Also, the importance of finding high-volume traffic clusters has been recognized. Network security tasks can be performed efficiently if high-volume traffic clusters, i.e., heavy hitter, change detection, and superspreader events, are promptly detected [23, 31, 40, 51, 79].

A heavy hitter [22] is a single source or destination that sends or receives a significant amount of traffic (bytes or packets) over a short period of time. It is beneficial for short-term monitoring, e.g., detecting flash crowds and DoS (Denial of Service) attacks, and longer-term traffic engineering [31, 52, 6]. Change detection is the practice of finding which flows contribute the most to the traffic pattern changes over two consecutive time intervals [15]. The method detects traffic anomalies by deriving a model of normal behavior based on the past traffic history and looking for significant changes in short-term behavior that are inconsistent with the model [44]. Finding which flow contributes the most to the traffic pattern changes in a short period of time is important in the context of anomaly detection [45, 46]. Finally, a superspreader is a host that contacts at least a given number of distinct destinations over a short time period. Detecting a source that reaches multiple destinations is needed for a worm or scan detection [93, 97]. In addition, if the same spread detection is applied to the destination, this analysis allows DDoS (Distributed DoS) victim detection [100]. Each of these events exhibits high-volume traffic clusters differently. While in the context of heavy hitters, the cluster relates to packets or bytes arrival rate, for superspreaders, the interest shifts to the distinct flows arrival rate.

Although SDN networks and OpenFlow promise some improvements in network measurement, the flow collection methods used in traditional network monitoring are preserved [88]. The primary monitoring mechanism exposes the per-port and per-flow counters. The SDN switches employ packet sampling to lower the overhead and data collection bandwidth at the cost of estimation accuracy [19, 31, 53]. An application running on top of the controller can periodically poll these counters using the standard OpenFlow APIs and then perform a software-based algorithm to get insights into the network behavior. In fact, this approach significantly limits the original flexibility intended by SDN. While increasing the gap between two consecutive counter requests reduces the controller's ability to react quickly, continuously requesting counters from switches leads to non-scalable solutions by challenging the switch-controller interactions capabilities [23].

Network Telemetry

Lately, the advent of the flexible data plane and P4 programmability has enabled the possibility of extending network monitoring functionality with more advanced traffic analysis features. Nonetheless, current programmable devices have constrained resources to meet performance requirements (as described in section 2.1), requiring smart solutions to deal

with both computation model and memory limitations. Many recent proposals suggest a form of reusable primitives to extend data plane functionality with more advanced features to tackle these challenges and support better monitoring and security application design. In this context, network telemetry refers to a brand new approach to network data collection leveraging the data plane programmability that provides sufficient network visibility, better scalability, accuracy, coverage, and performance than the former traditional and software-defined approaches to network measurement [99, 88].

The most prominent solution in this area is the In-band network telemetry (INT) [37, 88], which proposes a method to append useful meta-information on the packet path to the packet itself. It is a P4-based framework that collects and reports the network state using the data plane without any intervention from the control plane. INT packets may contain header fields representing telemetry instructions that network devices interpret on the path. Packet sources (applications or end-hosts) can embed these instructions in data packets to request network measurement. The instructions tell the network device what state to collect. The state may then be directly exported or written into the same packet as it traverses the network. Devices can report, e.g., precise hop-based latencies and queue or buffer occupancies, which helps to improve network congestion control. The INT framework, thus, focuses primarily on measuring end-to-end latency but not on network security or traffic clusters monitoring.

The state-of-the-art solutions focused on traffic clusters instead utilize various probabilistic data structures to keep the flow state at network devices, i.e., bloom-based filters [9] and sketches [36, 51, 98], which can easily fit into the match-action data plane architecture. Several frameworks providing such primitives have been developed. Specifically, many leverage data plane programmability to directly detect and provide the high-volume traffic clusters [49, 36, 51, 98, 100, 79] to assist the control plane by exporting smart representations of aggregated traffic statistics to support real-time monitoring. FlowRadar [49] keeps track of the flows and their counters in the network and exports this information periodically to a remote controller, which decodes it and uses it in further traffic analysis. UnivMon [51], ElasticSketch [98], SketchLearn [36] use a general sketch in the data plane to keep track of the flows and export it at fixed time intervals to the control plane. HashPipe [79] directly determines the top-k heavy hitters in the data plane and exports them at a specified time.

Although such solutions, compared to traditional and software-defined measurement, improve network visibility, they still depend on exporting complex data structures to a controller for analysis, relying on communication capabilities between data and control planes, as well as polling collected statistics at short intervals [23]. Such an architecture has drawbacks similar to those of the OpenFlow protocol and SDN network. The interaction between the control and data planes can lead to cost and delay overheads, especially in use cases like DDoS detection, where the detection speed affects the ability to respond to the event promptly. Therefore, there is a valid need for designing a novel concept that improves the state-of-the-art in this area and resolves such scalability issues.

2.4 Real-time Intrusion Detection

Nowadays, considerable effort is required to ensure the security of computer networks. A potential attack must be detected immediately, and adequate network protection must be provided promptly. For example, DoS (Denial of Service) attacks are one of the ever-increasing cybersecurity threats requiring an immediate response. DDoS (Distributed DoS)

is an intended and coordinated attempt to cause a target system or service to become unavailable. To protect against such malicious attacker activities, various Intrusion Detection Systems (IDS) have gained popularity in this area, and they are, in general, a critical element of protection in network security monitoring, event detection, and eliminating malicious network traffic. The IDS is a security system that monitors the activity of a protected network, intending to identify its potential unauthorized use or abuse [61]. Identified malicious activities are collected and reported. Afterward, either automated or manual reaction can take place. IDS significantly contribute to network security by providing a deeper insight into transferred packets, e.g., their payloads. These systems often use some form of deep packet inspection (DPI), such as pattern matching or other methods, to detect characteristic signatures of malicious activity in the network data.

Instrusion Detection Aspects

Generally, two main strategies for intrusion detection exist [89]: *signature-based* and *anomaly-based*. Signature-based systems, also called misuse-based systems, utilize statistical techniques like n-grams [94] or search for known patterns in communication based on a pre-built database of previous attacks [72, 86, 90, 68, 5, 26]. Due to their design, they respond reliably only to known attacks and cannot catch unknown (zero-day) attacks. The database is required to be updated regularly to incorporate new signatures. A significant disadvantage of this approach is the dependence on the quality of the database. Pattern searches focus on both packet headers and packet payloads.

Anomaly-based systems [8, 30, 101, 58, 73, 60] rely on finding statistical deviations from a normal state of network communication. They establish a normal behavior profile and compare new behavior against this model to detect deviations. In contrast to signatures, anomaly-based systems can detect unknown attacks but suffer from higher false positive rates. For example, it may be challenging to distinguish a DDoS attack from a massive interest of legitimate users in a service.

Based on detection time, both methods can be either *online* or *offline*. The online approaches aim to detect an intrusion within specific time constraints. They are used for real-time alerting and mitigation, whereas offline methods are typically used for post-attack and forensics analyses. The response type can be *passive* or *active*. Passive IDS only monitors the system and reports potential intrusions. Active IDS takes a proactive approach and aims to adopt countermeasures for attack mitigation. Since attack mitigation needs to be performed in real-time, active IDS methods require online detection.

A typical property of DPI methods included in IDS is their overwhelming computational complexity, leading to challenges in meeting the performance requirements of modern high-speed networks. To achieve the highest detection quality, IDS should process as much relevant data as possible without becoming a network connection bottleneck. In this light, fast packet processing and early detection of attacks are crucial for a successful response to minimize damage or mitigate attacks. At the same time, IDS implementation should be flexible enough to accommodate detection methods of ever-emerging new security threats. The flexibility of software implementation is, therefore, highly desirable. However, the pure software processing has obvious performance issues. On the other hand, using powerful but considerably less flexible and fixed hardware processing offload that accelerates only a specific part of the IDS, e.g., pattern matching, is also not feasible. In this regard, the emergence of data plane programmability offers many more opportunities to achieve better performance and flexibility tradeoffs.

Indeed, several IDS using programmable switches focusing, for example, specifically on detecting and mitigating volumetric DDoS attacks have been proposed [27, 103, 50]. INDDoS [27] introduces a data structure combining a bitmap and a sketch to estimate per-destination flow cardinality to build a DDoS victim identification strategy. Poseidon [103] extends the detection strategy by a set of attack mitigation primitives running partially on a programmable switch and in server software. Jaqen [50] runs volumetric DDoS defense entirely on a programmable switch without relying on additional out-of-band detection.

Pattern-Matching Acceleration

There are several familiar software implementations of signature-based IDS. Snort [81] is an open-source intrusion detection and prevention system. It relies heavily on regular expression matching. L7-filter [5] also operates with regular expressions. It is a Linux-based packet classification software that processes the application layer (L7). Suricata [86, 24] is functionally quite similar to Snort but has better multithreaded processing support and is, overall, built to achieve higher performance. Zeek [90, 68, 85] is a very flexible framework that mainly allows the specification of custom detection rules using its scripting language. This feature makes it very powerful but also rather complex. Thus, meeting the performance requirements to enable IDS deployment in high-speed networks is challenging.

In addition to software implementations, many proposals [83, 59, 43, 104] advocate offloading the IDS functionality to a hardware accelerator. This approach, which typically involves converting regular expressions into an FPGA firmware structure, can significantly improve performance by offloading the time-consuming pattern matching from the CPU. However, there is a trade-off in reduced flexibility. The proposed methods often require recompilation of the FPGA firmware when the regular expression set changes, and advanced techniques such as TCP stream reassembly or encrypted network packet processing are often missing from hardware-based IDS accelerators, potentially leaving vulnerabilities for covert attacks. It is important to note that while the known IDS (Snort, Suricata or Zeek) are primarily based on pattern matching, this is just one part of a complex system. All the mentioned IDS allow user-defined detection described by their scripting language. To enhance the quality of detection, they also use other heuristic methods that preserve state information between individual packets of a network flow to provide a context. Thus, approaches based solely on pattern-matching acceleration may limit flexibility and prove impractical.

Therefore, there is a space for alternative approaches. In particular, we assume the use of high-speed network traffic characteristics and the possibility of pre-processing (pre-filtering) part of the network traffic, which is not essential from the security analysis point of view. Such pre-filtering implemented in the data plane can thus enable significant software load reduction. An example of such a more flexible acceleration approach is Shunting [95]. It is a hardware accelerator that redirects only potentially suspicious (interesting) traffic to the software for further analysis.

Real-time Attack Mitigation

Despite the reliability of signatures, research has also shifted towards anomaly and machine learning (ML) approaches [8, 30] to address unknown attacks or encrypted traffic processing. Several ML-based anomaly approaches evaluate each packet within the attack discriminator separately. Therefore, the detection methods moved especially towards Recurrent Neural Networks (RNN) due to their ability to maintain context. DeepDefense

[101] uses a bi-directional LSTM-RNN with the last 100 packets. LSTM-BA [48] improved its performance by combining an RNN with a Naive Bayes classifier. Per-packet methods are also sometimes combined with windowing for additional context. Kitsune [58] uses per-packet feature extraction and KitNET, an ensemble of autoencoders – unsupervised artificial neural networks, for anomaly detection. Chronos [73] employs the same feature extraction as Kitsune, but their handling and model architecture differ. Doshi [28] combines stateful window and stateless packet header features.

Although the above methods achieve promising results, they still require evaluating every packet using a complex and computationally intensive ML model. Despite sufficient performance for smaller-scale local networks, the per-packet approach might be infeasible for more extensive networks with only a few nanoseconds to process a packet available. Therefore, researchers started investigating alternative approaches, such as flow-based intrusion detection [84, 18]. For example, mainstream ML-based DDoS detection methods predominantly utilize network flows [60]. Although the traditional network flow collection is widely deployed and well-matured, flow-based intrusion detection might still be unsuitable if a near-instant reaction to attack is required. In addition to losing packet payload information, flow-based detection has two more flaws: 1) A network flow captures only a single data exchange between a client and a server. It provides no context about the client's previous communication. Therefore, port scans or attacks using port randomization create a separate flow for each attack packet. The capabilities of flow-based methods without flow correlation are thus significantly hampered. 2) Flow collection introduces significant delays due to creation and export intervals. Typically, a flow terminates after observing a TCP FIN flag or when a timeout occurs. However, attacks like SYN Flood do not carry FIN flags, so flows must be terminated upon timeout. Observed flows are also exported in bulks, so dozens of seconds might pass until the flow data arrives at a flow collector. Afterward, IDS still needs to access the collector to retrieve the entries, adding additional delay.

Real-time attack mitigation is the primary goal in case of practical anti-DDoS intrusion detection solutions [7]. Delays of dozens of seconds or even minutes might be unacceptable. Nevertheless, the most current research on ML-based IDS implicitly assumes only off-line scenarios [60, 12]. As outlined earlier, flow-based methods might add additional delays, and a practical and reliable method for attack detection and consequent mitigation is still missing. To enable real-time ML-based intrusion detection and mitigation in high-speed networks, we argue that one must classify at higher-level abstraction, i.e., aggregated windows or network traffic clusters. Therefore, windowing methods utilizing emerging programmable data plane capabilities and improving the drawbacks of both pure packet-based and flow-based methods are promising research directions for real-time attack mitigation.

Chapter 3

Research Summary

3.1 Research Process

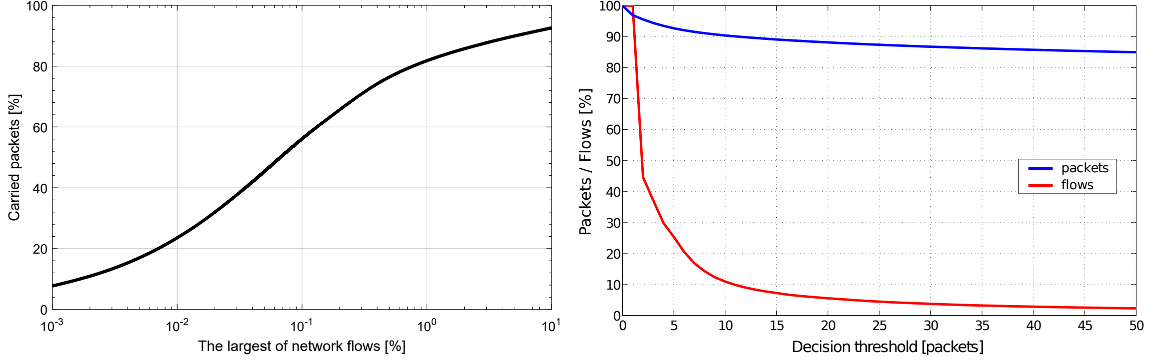
The thesis aims to accelerate network applications and design suitable architectures and algorithms using the programmable data plane to increase performance. The concepts proposed in this dissertation successively exploit either the properties of high-speed network traffic, window-based network traffic aggregations, feature extraction, or an event-triggered push-based approach. They selectively drop the level of information in packets that must be processed and reduce the data-control plane communication overhead to optimize the overall network application performance.

Instead of directly accelerating the most common and time-critical operations, as other state-of-the-art solutions do, the first acceleration concept proposed in the thesis and presented in the included papers [I, II] takes an alternative approach. It focuses on reducing (pre-filtering) network traffic, which must be analyzed by the application in the first place. The approach utilizes specific network traffic characteristics to achieve application acceleration. By exploiting these properties, it is possible to decide which packets are significant in network security event detection, i.e., which must be analyzed and which can be suitably reduced, aggregated, or discarded. In this way, it is possible to concentrate the available computational resources of the application only on the relevant part of the network traffic and achieve higher throughput.

This acceleration concept applied for network monitoring and intrusion detection systems follows two assumptions:

- The most high-speed network traffic (in terms of volume) is carried by a very small number of relatively large network flows.
- From a security point of view, the initial packets of network flows are the most important because they carry the most useful information (i.e., packet headers) needed to detect unwanted traffic or other security events.

We expect that when the network system is heavily loaded, incoming packets are dropped uncontrollably at its input due to an overflow of the input packet buffers on the network interface card, from which packets are not retrieved fast enough. This packet-dropping mechanism significantly and negatively impacts the detection quality because all packets are dropped with the same probability, regardless of their importance in terms of event detection. Thus, we hypothesize that better results can be achieved through controlled packet filtering of network flows. Under high load, only data relevant to the analysis



(a) Percentage of packets carried by the largest flows.

(b) Simple method of heavy flow detection [41].

Figure 3.1: Network traffic characteristics, simple heavy flow detection capabilities.

can be selectively processed. The less significant part of the network traffic can then be discarded in a controlled manner due to limited performance.

Figure 3.1a shows the results of network data analysis from an ISP backbone network. It presents the distribution of network flow lengths as the fraction of packets carried by the largest flows. It demonstrates, for example, that a single percentage of the largest network flows carry more than 80 % of the packets of all network traffic. This property of high-speed network traffic is called heavy-tailed distribution of network flow lengths [31, 47, 41]. The benefit of this property for acceleration depends on the ability to detect heavy flows based only on an analysis of the first few packets of each flow. The simplest method consists of marking a flow as heavy upon the arrival of a certain number N of initial packets. The graph 3.1b shows the relationship between the chosen value of N (decision threshold in packets), the fraction of marked heavy flows (in red), and the corresponding fraction of the total number of packets (in blue). For example, for N=20, only 5 % of flows are marked as heavy, while almost 90 % of all packets are covered. This property enables to reduce the packets of heavy flows to accelerate the applications. Therefore, the proposed concept assumes that only the first N packets of each network flow will always be fully processed by the original application (IDS). Larger flows can be considered heavy, and subsequent packets of such flows can, therefore, be effectively offloaded.

The approach allows for both software and hardware implementation or the division of the task between hardware and software resources. Figure 3.2 shows a diagram of the proposed architecture of such a solution. In the software part (shown in blue), the analysis of the packet headers and extraction of the items identifying the network flow is performed first. Next, an entry corresponding to the network flow is looked up in the cache (in the case of absence, an entry is created). The packet counter of the flow is incremented. The system discards the packet if the counter exceeds the specified threshold N. Otherwise, it forwards the packet to the original application (IDS) for further processing. The figure shows the optional hardware part in grey. The accelerator performs packet header analysis and searches for the corresponding network flow in its local cache. It then discards all packets of flows that persist in the cache. Thus, only packets for which no flow record is found in the hardware cache are forwarded to the software for processing. In parallel, the synchronization of items from the software cache to the hardware cache occurs. Only the heavy flows (those that have exceeded the specified threshold) are synchronized. Removal of obsolete, inactive records from the cache is also performed periodically according to set

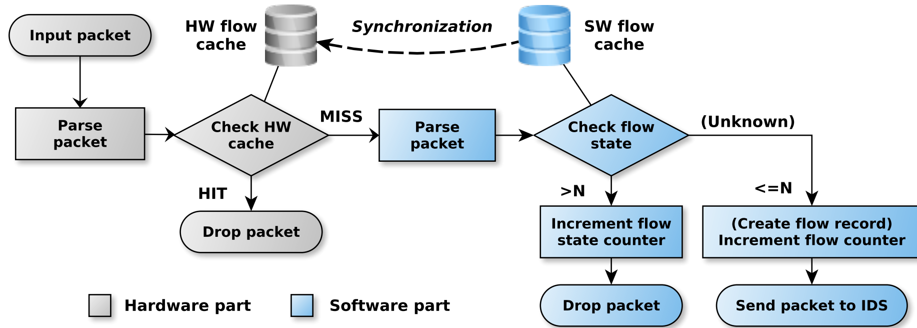


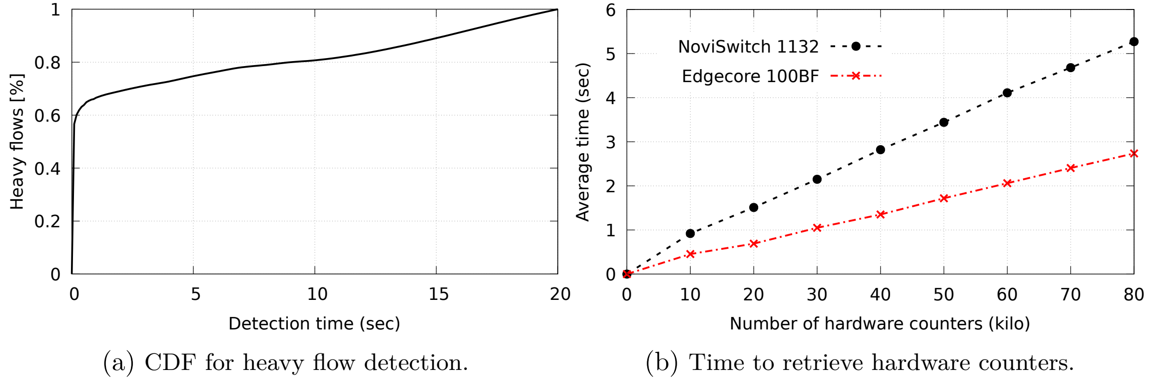
Figure 3.2: Packet processing in the proposed acceleration with hardware offload.

timeouts (not shown in the figure). Also, the threshold value N can be controlled flexibly depending on the current network load and the hardware cache capacity.

The first research paper [I] describes the concept and introduces the hardware architecture for the FPGA platform. The original work focused only on network monitoring use cases. Thus, the concept was named *Software Defined Monitoring (SDM)*. The paper primarily analyses the heavy-tailed distribution of network flow lengths and presents the case of flexible acceleration for application protocol analysis. The proposed informed discarding concept is implemented and experimentally evaluated to demonstrate its effectiveness under actual network deployment conditions. The second paper [II] further evaluates other assumptions and demonstrates the controlled reduction of incoming packet traffic as a general flow-based acceleration suitable also for intrusion detection systems. The preliminary steps in the design process are also described in other related publications [R1, R4].

Later, as the following research direction, we took a closer look at the consequences of splitting the network monitoring tasks between hardware and software and analyzed the possible overheads in the communication between data and control planes. The original motivation was to reduce the latency in detecting heavy network flows by offloading the task entirely in the data plane. However, it turned out to be a much more general research question. State-of-the-art solutions [36, 51, 98, 100, 79, 49] utilize data planes to accelerate network monitoring tasks by implementing various probabilistic data structures to aggregate flow counters and provide them to a controller. An application running on top of the controller then periodically polls the structures from the data plane and performs a software-based algorithm to get insights into the network behavior. We ran an experiment to estimate the importance of setting the correct reporting time in the context of heavy flow detection and measure the amount of time it takes to retrieve an increasing number of hardware counters from a switch.

We assumed that the switch exports the flow counters periodically, and the controller, in charge of the detection, considers heavy flows that exceed a specific percentage of the total traffic, e.g., 1 %. We considered a reporting time of 20 seconds, as suggested by state-of-the-art solutions [51, 79]. First, we computed heavy flows using real packet traces from an ISP backbone network. Later, we decreased the reporting time and calculated which heavy flows could have been detected earlier. Figure 3.3a reports the cumulative distribution function (CDF) of reported heavy flows varying the reporting time. Interestingly, on average, more than 60 % of the heavy flows could have been detected earlier, within a fraction of a second. This test suggested that, in theory, the reporting time should be set as low as possible, to enable the controller the event detection and a reaction in a more timely fashion. However,



(a) CDF for heavy flow detection.

(b) Time to retrieve hardware counters.

Figure 3.3: Experiment motivating event-triggered monitoring.

in practise, it is important to take into account also the switch-controller capabilities of collecting counters at short time scales.

To measure the time it takes to retrieve hardware counters from the data plane, we connected two switches to a controller and built an application to request an increasing number of flow counters. The switches were idle when counters were pulled. Figure 3.3b shows the results. Although the use of probabilistic data structures, i.e., sketches, can help in reducing the number of counters to be exported, past research has shown that from around 60 K to 150 K counters are still required to provide helpful information to a controller [51, 98]. In this context, OpenFlow-enabled NoviSwitch needs at least 5 seconds, and the P4-enabled Edgecore switch needs 2.5 seconds. We observed that retrieving a large amount of data from hardware is time-consuming and requires time scales of seconds. This finding has two significant implications: (1) given that the statistics retrieval process is performed periodically, the operation needs to be dimensioned with respect to the switch capabilities: the reporting time cannot be lower than the time needed to collect the statistics; (2) in the worst case scenario, a controller can apply appropriate network updates only seconds after a specific event has happened. Therefore, performing traffic analysis in the controller might introduce delays that are undesirable for specific use cases, e.g., DDoS detection.

In this context, we take a different approach to optimizing the switch-controller communication overhead. We leverage data plane programmability to transform the data plane from a passive monitoring infrastructure to an active system capable of detecting several types of network events autonomously and iteratively by tracking the responsible network traffic and only subsequently informing the control plane. We thus proposed an event-triggered and push-based approach to network monitoring, where the data plane informs the control plane only when specific conditions are met.

The paper [III] presents Unroller, an architecture that enables real-time identification of routing loops in the data plane with minimal overheads. It follows the event-triggered push-based approach and informs the controller only when a loop is identified. At the same time, the overhead of on-packet or on-switch state is reduced using probabilistic data structures. The following publication [R3] and the paper [IV] extend the concept and propose Elastic Trie, a data structure and architecture that allows the detection of traffic aggregates. It can spot changes in traffic patterns and detect either heavy hitters (sources of heavy flows) or superspreaders (sources of many distinct flows) entirely within the data plane while significantly reducing the data-control plane communication. In addition, such

events relate to network scans, the spread of malware, or DDoS attacks. Thus, the event-triggered approach can speed up the corresponding response to them.

In particular, the usage of DDoS attacks has grown massively in recent years. They are one of the most widespread cyber-attacks on the Internet today, targeting the essential goal of cybersecurity – availability, pushing more and more network infrastructure and service providers to deploy some security solution that allows them to respond to such a threat in a timely manner. Therefore, in the rest of the thesis, we focus more extensively on DDoS detection and mitigation as one of the critical use cases of network security monitoring. In this case, the detection speed significantly affects the ability of network operators to respond to security events. Meanwhile, focusing on data plane algorithms to minimize latency can, in addition to detection, enable near real-time response in the event of an attack to filter the malicious traffic promptly.

The paper [V] describes reactive defense mechanisms, which activates only when an ongoing attack is detected. Three real-time, network-based SYN Flood mitigation strategies are implemented within the data plane to reduce the processing and attack response latency. The algorithmic methods authenticate TCP clients using cryptographically secured information injected into packets during the TCP handshake to prevent attackers from IP address spoofing. The following paper [VI] later introduces Windower, a data plane feature extractor and an intrusion detection system. It accelerates existing ML-based mitigation techniques by a specific packet aggregation to reduce the amount of data needed to be processed by the ML model. In a way, it reuses the previous concept of informed reduction of incoming network traffic. It applies the concept to the area of DDoS attack mitigation to achieve an acceleration without negatively impacting the detection quality.

In summary, the published journal and conference papers included in the thesis provide multiple hardware architectures and software algorithms for network data plane optimization of three different use-cases: (1) *flow-based network monitoring*, (2) *in-band network events detection*, and (3) *real-time DDoS mitigation*. The acceleration considers informed packet reduction, network traffic aggregation, and specific data structures, which provide an event-triggered approach to enable real-time response to security events. The architectures and implementations have been practically used in network monitoring infrastructure and in the DDoS protection system for CESNET2, the Czech academic network.

3.2 Papers

The section presents the overview of all the papers included in the thesis. For each paper we describe its motivation, emphasize the most relevant outcomes and contributions in terms of the research topic presented in the thesis, and provide its original abstract. Full versions of these papers are an integral part of the thesis and can be found (in their original formating) in Appendix A.

3.2.1 Paper I

Software Defined Monitoring of Application Protocols

Lukáš KEKELY, Jan KUČERA, Viktor PUŠ, Jan KOŘENEK and Athanasios V. VASILAKOS. "Software Defined Monitoring of Application Protocols". In: *IEEE Transactions on Computers* 65.2 (2016), pp. 615–626. ISSN: 0018-9340.

Author participation: 30 %
Journal impact factor (IF): 3.7
Journal rank: Q1

Contribution

Network functionality is increasingly shifting up in the protocol stack towards upper layers. Thus, to maintain the detection quality of emerging network threats, it is crucial to be able to analyze the application layer (L7). While packet and protocol parsing is one of the most time-critical operations, the ongoing trend in network flow monitoring is to provide enriched flow records, i.e., to append some extra information from L7 protocol headers, e.g., include a URL field from HTTP protocol or a domain name from DNS protocol. Moreover, L7 protocols are evolving quickly, and advanced techniques like TCP stream reassembling are often missing in hardware accelerators. Thus, it is hard to accelerate L7 protocols in hardware easily. Therefore, the paper proposes a Software Defined Monitoring (SDM) concept. The core idea is that even advanced L7 monitoring usually needs to observe only a small fraction of traffic. For example, DNS traffic typically represents no more than 1 % of all network packets, or an HTTP header appears in the first few packets of the network flow. While the software controls the level of offload, the hardware accelerator can aggregate into flow records or drop the traffic that is no longer interesting for L7 processing. SDM allows the software to selectively drop the level of information in the data plane and control the tradeoff between the software and hardware processing according to the load and application requirements.

The result is the hardware architecture implementing the SDM concept for flexible software-controlled stateful network flow monitoring. It includes detailed synthesis results of resource utilization and achieved throughput analysis for the Virtex-7 FPGA-based network interface card. It evaluates the prototype and compares the flow offload analytical and actual measured effectivity on five selected network monitoring use cases.

Abstract

With the ongoing shift of network services to the application layer also the monitoring systems focus more on the data from the application layer. The increasing speed of the network links, together with the increased complexity of application protocol processing, require a

new way of hardware acceleration. We propose a new concept of hardware acceleration for flexible flow-based application level traffic monitoring which we call Software Defined Monitoring. Application layer processing is performed by monitoring tasks implemented in the software in conjunction with a configurable hardware accelerator. The accelerator is a high-speed application-specific processor tailored to stateful flow processing. The software monitoring tasks control the level of detail retained by the hardware for each flow in such a way that the usable information is always retained, while the remaining data is processed by simpler methods. Flexibility of the concept is provided by a plugin-based design of both hardware and software, which ensures adaptability in the evolving world of network monitoring. Our high-speed implementation using FPGA acceleration board in a commodity server is able to perform a 100 Gb/s flow traffic measurement augmented by a selected application-level protocol analysis.

3.2.2 Paper II

General IDS Acceleration for High-Speed Networks

Jan KUČERA, Lukáš KEKELY, Adam PIECEK and Jan KOŘENEK. "General IDS Acceleration for High-Speed Networks". In: *Proceedings of the 36th IEEE International Conference on Computer Design.* ICCD 2018. Orlando, FL, USA: IEEE, 2018, pp. 366–373. ISBN: 978-1-5386-8477-1.

Author participation: 30 %
Conference rank: A2 (Qualis)

Contribution

Intrusion detection systems contribute to network security through deep packet inspection, such as pattern matching, thus providing a deeper insight into network traffic. However, they suffer from overwhelming computational complexity and challenges in meeting the performance requirements of high-speed networks. The paper presents a new concept of IDS acceleration based on controlled (informed) reduction of incoming packet traffic while maintaining sufficiently good overall threat detection capabilities. The architecture presented in the previous paper [I] is adopted for general flow-based acceleration of intrusion detection systems. It proposes to drop all packets that follow the first N packets of each network flow, where the value of N can be optimized on the fly according to the current IDS load. From a different perspective, if an IDS is overloaded, it suggests skipping packets at the ends of heavy network flows instead of blind input buffer overflow behavior to retain detection quality at higher input speeds.

The result is a new concept of IDS acceleration, prototyped for Suricata IDS. However, it is general enough to be deployed with other software-based IDS. Moreover, the proposed concept can be implemented as a pure software system prefiltering the input network traffic or as a dedicated hardware accelerator.

Abstract

Network Intrusion Detection Systems have gained popularity as one of the key technologies to secure communication infrastructures. However, their high computational complexity poses performance challenges for practical deployment in modern high-speed networks. To achieve the highest quality of detection, IDS should process as much relevant data

as it can without becoming the bottleneck of a network connection. At the same time, IDS implementation should be flexible enough to accommodate detection methods of ever emerging new security threats. This paper aims at an acceleration of IDS by means of informed packet discarding, effectively focusing the available resources of overloaded IDS to the most relevant parts of analyzed traffic. Unlike previous works, the proposed scheme does not move the IDS nor any specific portion of it into the hardware accelerator. Rather it uses smart software based or hardware accelerated offload (bypass) of the traffic parts that are not likely to represent a security threat. The flexible nature of software-based IDS is therefore fully maintained, while the quality of threat detection remains sufficiently high even when processing high-speed traffic. We show that controlled (informed) discarding of well-defined portions of input traffic yields better detection rates, compared to the default uncontrolled (blind) buffer overflow discarding in high throughput scenarios. Our results show that it is entirely possible to run an IDS on a high-speed network link using single CPU with an FPGA accelerated packet pre-filtering.

3.2.3 Paper III

Detecting Routing Loops in the Data Plane

Jan KUČERA, Ran BEN BASAT, Mário KUKA, Gianni ANTICHI, Minlan YU and Michael MITZENMACHER. "Detecting Routing Loops in the Data Plane". In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT 2020. Barcelona, Spain: ACM, 2020, pp. 466–473. ISBN: 978-1-4503-7948-9.

Author participation: 28 %
Conference rank: B1 (Qualis), A (Core)

Contribution

Detection of traffic loops is essential for the performance of current computer networks. Unidentified loops may significantly increase the overall traffic, reduce throughput, or lead to packet loss, as well as other adverse effects regarding delay and jitter. The paper aims to provide a data plane-based solution that detects routing loops in real-time while keeping low switch and network overhead. The proposed approach probabilistically encodes only a small subset of switch identifications along the path on packets while guaranteeing the detection in a bounded number of hops. It is also a prototype of a more general event-triggered concept of network monitoring, where the data plane informs the control plane only when a specific network event is detected.

The result of the paper is the hardware architecture that implements the concept in P4. The paper demonstrates the performance and low overhead on several real WAN and data center topologies and provides resource utilization when compiled into three different FPGA-based targets.

Abstract

Routing loops can harm network operation. Existing loop detection mechanisms, including mirroring packets, storing state on switches, or encoding the path onto packets, impose significant overheads on either the switches or the network. We present Unroller, a solution that enables real-time identification of routing loops in the data plane with minimal

overheads. Our algorithms encode a varying fixed-size subset of the traversed path on each packet. That way, our overhead is independent of the path length, while we can detect the loop once the packet returns to some encoded switch. We implemented Unroller in P4 and compiled into three different FPGA targets. We then compared it against state-of-the-art solutions on real WAN and data center topologies and show that it requires from 6 x to 100 x fewer bits added to packets than existing methods.

3.2.4 Paper IV

Enabling Event-Triggered Data Plane Monitoring

Jan KUČERA, Diana Andreea POPESCU, Gianni ANTICHI, Han WANG, Andrew W. MOORE and Jan KOŘENEK. "Enabling Event-Triggered Data Plane Monitoring". In: *Proceedings of the 2020 SIGCOMM Symposium on SDN Research.* SOSR 2020. San Jose, CA, USA: ACM, 2020, pp. 14–26. ISBN: 978-1-4503-7101-8.

Author participation: 30 %
SIGCOMM conference oriented on
SDN and programmable data planes

Contribution

Many network applications like traffic engineering, scan, worm, DDoS, or anomaly detection benefit from real-time detection of high-volume traffic clusters. While the advent of programmable switches enabled the extension of data plane functionality to support such applications, the current solutions still rely on exporting specific complex data structures to a controller for analysis. Thus, they highly depend on the data-control planes' communication capabilities, and polling collected statistics from the data plane at short time scales. The interaction between the control and data plane is expensive and, in specific use cases, e.g., DDoS detection, implies unacceptable delay overheads. The paper presents an event-triggered push-based solution implemented within the data plane that does not need a controller to retrieve the whole data structure to detect a particular traffic cluster event. Instead, the data plane informs the controller only when specific conditions are met.

The result is a new probabilistic data structure, Elastic Trie, that enables detecting the mentioned network events entirely in the data plane, by iteratively tracking the responsible IP prefixes, and only subsequently informing the controller. The corresponding hardware architecture is designed as hash tables based prefix tree that grows or collapses to focus only on prefixes that represent significant share of network traffic.

Abstract

We propose a push-based approach to network monitoring that allows the detection, within the dataplane, of traffic aggregates. Notifications from the switch to the controller are sent only if required, avoiding the transmission or processing of unnecessary data. Furthermore, the dataplane iteratively refines the responsible IP prefixes, allowing the controller to receive information with a flexible granularity. We implemented our solution, Elastic Trie, in P4 and for two different FPGA devices. We evaluated it with packet traces from an ISP backbone. Our approach can spot changes in the traffic patterns and detect (with 95 % of accuracy) either hierarchical heavy hitters with less than 8 KB or superspreaders with less than 300 KB of memory, respectively. Additionally, it reduces controller-dataplane

communication overheads by up to two orders of magnitude with respect to state-of-the-art solutions.

3.2.5 Paper V

Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques

Patrik GOLDSCHMIDT and Jan KUČERA. “Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques”. In: *Proceedings of 2021 IFIP/IEEE International Symposium on Integrated Network Management.* IM 2021. Bordeaux, France: IFIP, 2021, pp. 772–777. ISBN: 978-3-903176-32-4.

Author participation: 50 %
Conference rank: B1 (Qualis), B (Core)

Contribution

The number and scale of DDoS attacks are steadily increasing. According to the data from several past years, TCP SYN Flood is one of the most frequent DDoS attacks. The paper proposes three real-time TCP SYN Flood mitigation heuristics implemented in the data plane of a middleware DDoS protection box. They employ a reactive defense mechanism, activating only when an ongoing attack is detected. In that case, they utilize a modified TCP three-way handshake to validate each TCP client before forwarding its SYN data. Similarly to the SYN cookies technique, the protection middlebox spoofs a (cryptographically) crafted response that only a TCP client implementing an actual TCP/IP stack can correctly answer. The validated clients are then whitelisted, and the middlebox forwards their connections without further tampering.

The result is the implementation of three data plane mitigation algorithms on top of the DPDK framework. The paper further evaluates the performance in terms of achieved throughput when implemented on single or multiple CPU cores and analyses the delay of the first client’s connection attempt and the latency of all the subsequent connections.

Abstract

TCP SYN Flood is one of the most widespread DoS attack types performed on computer networks nowadays. As a possible countermeasure, we implemented and deployed modified versions of three network-based mitigation techniques for TCP SYN authentication. All of them utilize the TCP three-way handshake mechanism to establish a security association with a client before forwarding its SYN data. These algorithms are especially effective against regular attacks with spoofed IP addresses. However, our modifications allow deflecting even more sophisticated SYN floods able to bypass most of the conventional approaches. This comes at the cost of the delayed first connection attempt, but all subsequent SYN segments experience no significant additional latency (< 0.2ms). This paper provides a detailed description and analysis of the approaches, as well as implementation details with enhanced security tweaks. The discussed implementations are built on top of the hardware-accelerated FPGA-based DDoS protection solution developed by CESNET and are about to be deployed in its backbone network and Internet exchange point at NIX.CZ.

3.2.6 Paper VI

Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning

Patrik GOLDSCHMIDT and Jan KUČERA. "Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning". In: *Proceedings of 2024 IEEE/IFIP Network Operations and Management Symposium.* NOMS 2024. Seoul, South Korea: IEEE, 2024.

Author participation: 50 %
Conference rank: A2 (Qualis), B (Core)

Contribution

The current mainstream ways of ML-based DDoS detection substantially rely on network flows. Such approaches, however, suffer from two significant drawbacks: 1) Network flows do not provide any context about the client's previous communication, i.e., a separate flow record can be created for each attacking packet. 2) Observed flow records are exported in bulks with relatively long export times (dozens of seconds). Therefore, the paper presents Windower, a feature-extraction method and a practical and reliable approach for real-time DDoS attack detection and mitigation. It defines a specific feature set of statistical information collected from packets directly in the data plane to determine the client's communication patterns and their variability in time. It employs an autoencoder ensemble as an anomaly-based classifier to identify and later block malicious clients.

The paper's result is the feature extraction algorithm. The packet-based sliding window method is implemented in the data plane to compute traffic statistics from packet headers within short time frames using data mining techniques suitable for input packet stream processing. It significantly reduces the number of ML-model evaluations needed.

Abstract

Distributed Denial of Service (DDoS) attacks are an ever-increasing type of security incident on modern computer networks. To tackle the issue, we propose Windower, a feature-extraction method for real-time network-based intrusion (particularly DDoS) detection. Our stream data mining module employs a sliding window principle to compute statistical information directly from network packets. Furthermore, we summarize several such windows and compute inter-window statistics to increase detection reliability. Summarized statistics are then fed into an ML-based attack discriminator. If an attack is recognized, we drop the consequent attacking source's traffic using simple ACL rules. The experimental results evaluated on several datasets indicate the ability to reliably detect an ongoing attack within the first six seconds of its start and mitigate 99 % of flood and 92 % of slow attacks while maintaining false positives below 1 %. In contrast to state-of-the-art, our approach provides greater flexibility by achieving high detection performance and low resources as flow-based systems while offering prompt attack detection known from packet-based solutions. Windower thus brings an appealing trade-off between attack detection performance, detection delay, and computing resources suitable for real-world deployments.

3.3 List of Publications

Papers Included in Thesis

[I] Lukáš KEKELY, Jan KUČERA, Viktor PUŠ, Jan KOŘENEK and Athanasios V. VASILAKOS. “Software Defined Monitoring of Application Protocols”. In: *IEEE Transactions on Computers* 65.2 (2016), pp. 615–626. ISSN: 0018-9340.

[II] Jan KUČERA, Lukáš KEKELY, Adam PIECEK and Jan KOŘENEK. “General IDS Acceleration for High-Speed Networks”. In: *Proceedings of the 36th IEEE International Conference on Computer Design.* ICCD 2018. Orlando, FL, USA: IEEE, 2018, pp. 366–373. ISBN: 978-1-5386-8477-1.

[III] Jan KUČERA, Ran BEN BASAT, Mário KUKA, Gianni ANTICHI, Minlan YU and Michael MITZENMACHER. “Detecting Routing Loops in the Data Plane”. In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies.* CoNEXT 2020. Barcelona, Spain: ACM, 2020, pp. 466–473. ISBN: 978-1-4503-7948-9.

[IV] Jan KUČERA, Diana Andreea POPESCU, Gianni ANTICHI, Han WANG, Andrew W. MOORE and Jan KOŘENEK. “Enabling Event-Triggered Data Plane Monitoring”. In: *Proceedings of the 2020 SIGCOMM Symposium on SDN Research.* SOSR 2020. San Jose, CA, USA: ACM, 2020, pp. 14–26. ISBN: 978-1-4503-7101-8.

[V] Patrik GOLDSCHMIDT and Jan KUČERA. “Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques”. In: *Proceedings of 2021 IFIP-/IEEE International Symposium on Integrated Network Management.* IM 2021. Bordeaux, France: IFIP, 2021, pp. 772–777. ISBN: 978-3-903176-32-4.

[VI] Patrik GOLDSCHMIDT and Jan KUČERA. “Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning”. In: *Proceedings of 2024 IEEE-/IFIP Network Operations and Management Symposium.* NOMS 2024. Seoul, South Korea: IEEE, 2024.

Other Relevant Publications

[R1] Jan KUČERA. “Application-specific Processor for Stateful Network Traffic Processing”. In: *Proceedings of the 20th Student Conference on Electrical Engineering, Information and Communication Technologies.* Student EEICT 2014. Brno, Czech Republic, 2014, pp. 198–200. ISBN: 978-80-214-4922-0.

[R2] Viktor PUŠ, Lukáš KEKELY, Jan KUČERA and Denis MATOUŠEK. “Live demonstration of application layer traffic monitoring at 100 Gbps”. In: *Proceedings of the 40th Annual IEEE Conference on Local Computer Networks.* LCN 2015. Leonia, NJ, USA: IEEE, 2015, A50–A51. ISBN: 978-1-4673-6769-1.

[R3] Jan KUČERA, Diana Andreea POPESCU, Gianni ANTICHI, Andrew W. MOORE and Jan KOŘENEK. “Enabling Event Triggered Monitoring of Traffic Clusters”. In: *5th P4 Workshop 2018.* Stanford, CA, USA, 2018.

[R4] Jan KUČERA, Lukáš KEKELY, Viktor PUŠ, Adam PIECEK and Jan KOŘENEK. “Hardware Acceleration of Intrusion Detection Systems for High-Speed Networks”. In: *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems.* ANCS 2018. Ithaca, NY, USA: ACM, 2018, pp. 177–178. ISBN: 978-1-4503-5902-3.

[R5] Mário KUKA, Kamil VOJANEC, Jan KUČERA and Pavel BENÁČEK. "Accelerated DDoS Attacks Mitigation using Programmable Data Plane". In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS 2019. Cambridge, UK: IEEE, 2019, pp. 1–3. ISBN: 978-1-7281-4387-3.

Chapter 4

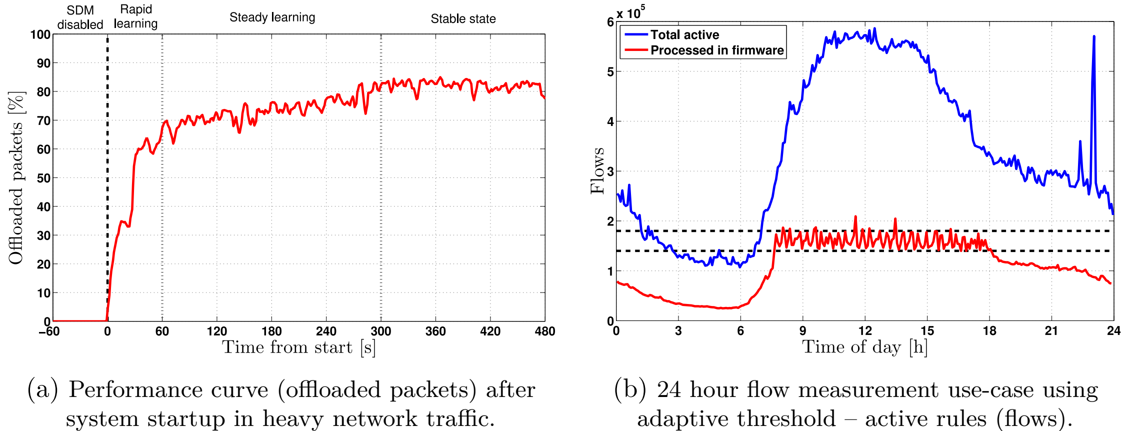
Research Results

Multiple hardware architectures and software algorithms for network data plane-based acceleration and optimization were proposed. The investigated concepts demonstrate the optimizations in three areas of network monitoring and network security applications: (1) *flow-based network monitoring*, (2) *in-band network events detection*, (3) *real-time DDoS mitigation*. In order to fulfill the objectives, the architectures employ cuckoo hashing, Bloom filters, sketches, and other probabilistic data structures and techniques to guarantee high detection performance and throughput while maintaining low resource requirements.

4.1 Flexible Network Monitoring

The first proposed architecture focuses on pre-filtering and reducing network traffic based on the characteristic heavy-tailed distribution of flow lengths. It follows the evidence that most network traffic (in terms of volume) is carried by a small number of relatively large network flows, which can be offloaded to achieve acceleration. We originally designed the architecture for use cases of network flow monitoring and application protocols analysis. We thus call it *Software Defined Monitoring (SDM)* even though we later adopted it for IDS acceleration too. We implemented the SDM acceleration concept using an FPGA-based network interface card to evaluate achieved performance metrics and resource requirements. We further deployed the system prototype in the network monitoring infrastructure at CESNET, the Czech NREN (National Research and Education Network) operator, to practically analyze SDM benefits and compare it to its non-accelerated counterpart.

Figure 4.1 shows the SDM architecture and flow measurement use-case evaluation conducted at the CESNET high-speed backbone network. We measured the performance curve after the system startup in heavy network traffic. These results are depicted in Figure 4.1a. When the system is activated (time 0), we immediately see a rapid increase in packets being offloaded. After about 5 minutes, the performance is stabilized at the offloaded percentage of packets in the range from 75 to 85 % of total traffic. Such constellation leads to considerable savings in system bus bandwidth. Moreover, it results in a corresponding decrease in CPU load, a positive effect of hardware acceleration based on the heavy-tailed distribution of network traffic. In Figure 4.1b, we further present the number of active rules, i.e., flows maintained in the hardware flow cache, compared to the number of active flows in the network during the day. Black dashed lines demarcate a desired flow cache load maintained by the adaptive selection technique of heavy flow detection threshold. The mechanism aims to maintain optimal hardware cache utilization depending on the actual amount of incoming



(a) Performance curve (offloaded packets) after system startup in heavy network traffic.

(b) 24 hour flow measurement use-case using adaptive threshold – active rules (flows).

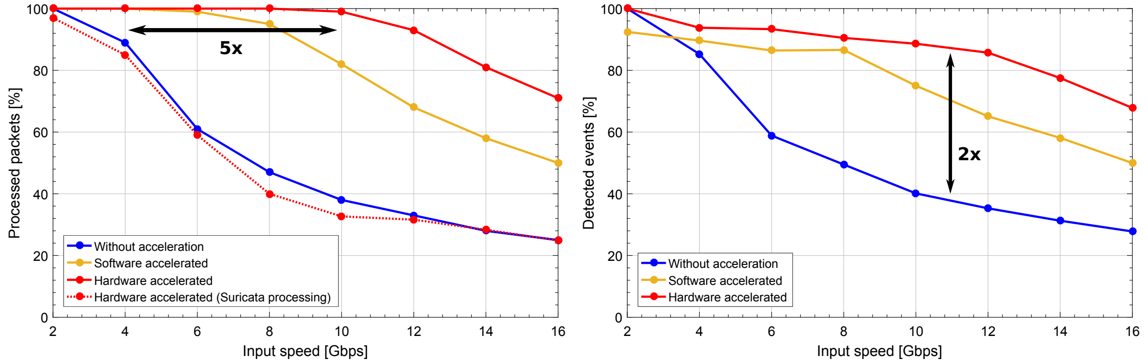
Figure 4.1: Evaluation of SDM architecture deployed at the CESNET backbone network.

network traffic. In the case of light load (night time), the threshold is gradually lowered to the minimum possible value of 5 packets to increase the proportion of hardware-processed network flows. Conversely, in the case of heavy traffic and full cache fill (the daytime), the threshold is increased to reduce the number of active flows in the cache. The adaptive threshold significantly improves the acceleration effect by efficiently managing the amount of collisions in the hardware cache. The complete analysis for other network monitoring use cases, e.g., application protocol processing, is provided in the included paper [I].

Further, we have adopted the exact implementation of SDM hardware architecture for general flow-based acceleration of intrusion detection systems. We demonstrated the proposed acceleration using Suricata IDS [86]. However, the concept is general enough to be deployable with any other software-based IDS regardless of the event detection mechanism it implements. Furthermore, the proposed SDM acceleration can also be implemented as a pure software system preprocessing the input network traffic. The graphs in Figure 4.2 show the IDS speedup achieved using both the software (orange) and hardware (red) implementations of the acceleration relative to the non-accelerated version (blue). Graph 4.2a shows the proportion of packets successfully processed, and graph 4.2b shows the proportion of events detected with regard to the input packet stream load. By reducing the uncontrolled packet loss at the input, up to five times higher processing throughput was achieved for the hardware-accelerated variant (see graph 4.2a). At the same time, the pure software performance of the IDS itself remained unchanged. From a different perspective, the results can also be seen as the detection quality of IDS is enhanced by the proposed acceleration more than twice at higher input speeds (see graph 4.2b).

From a hardware-oriented perspective, we implemented the SDM architecture specifically on top of COMBO-100G [33] network interface card, an FPGA accelerator equipped with Xilinx Virtex 7 XC7VH580T [2]. The firmware core implements a cuckoo hashing-based form [67] of the flow cache table with packet filtering capabilities. The flow cache is realized using external on-card QDR memories, and it has a total capacity of up to 2^{18} (~260 000) active filtering rules (flows). Because of the beneficial heavy-tailed character of network traffic and flow sizes, we offload only a very small percentage of the heaviest flows into the hardware. Hence, the cache capacity is not a limiting factor.

Table 4.1 presents the FPGA resources requirements. The SDM firmware core runs at 200 MHz and uses no more than 15 % of the Virtex-7 FPGA to achieve 100 Gbps throughput. In addition to the high-speed solution, we analyzed narrower data bus widths suitable



(a) The percentage of successfully processed packets. (b) The percentage of fully detected events.

Figure 4.2: IDS acceleration – the relation between processed packets and detected events.

Core module	LUTs	Regs	Throughput
512-bit bus	51 333 (14.15 %)	30 497 (4.20 %)	100 Gb/s
256-bit bus	42 793 (11.80 %)	25 866 (3.56 %)	50 Gb/s
128-bit bus	39 006 (10.75 %)	23 534 (3.24 %)	25 Gb/s
64-bit bus	37 233 (10.26 %)	22 384 (3.08 %)	12.5 Gb/s

NIC firmware	LUTs	Regs	Throughput
100 GbE port	249 214 (68.69 %)	197 758 (27.25 %)	1×100 Gb/s
40 GbE ports	178 984 (49.33 %)	134 172 (18.49 %)	2×40 Gb/s
10 GbE ports	222 745 (61.40 %)	184 084 (25.37 %)	8×10 Gb/s

Table 4.1: SDM firmware architecture resource utilization and achieved throughput.

for applications with lower throughput requirements. Finally, the table 4.1 shows the overall resources of the complete NIC firmware (including Ethernet, QDR, and PCI-Express controllers) for the COMBO network interface card in three different port configurations, i.e., one 100 GbE, two 40 GbE, and eight 10 GbE ports. The complete NIC firmware implementing the SDM concept occupies less than half of the available FPGA resources.

4.2 Event-triggered Data Plane

Following the research direction toward moving the detection of heavy network flows entirely in the data plane, we proposed an event-triggered and push-based approach for network monitoring. It informs the software controller only when a relevant network event is detected. Therefore, it significantly optimizes the data-control plane communication overhead. To demonstrate this, we proposed Unroller, the first proof-of-concept algorithm of the event-triggered approach that focuses on the real-time identification of routing loops. Later, we also proposed ElasticTrie, another event-triggered algorithm that allows even the detection of traffic aggregates, i.e., heavy hitters, superspreaders, and significant traffic changes, entirely in the data plane.

The Unroller algorithm maintains its state using a probabilistic approach to store visited switch IDs on packets. We implemented it in P4 and compared its loop detection capabilities against PathDump [87] and an especially crafted approach that, in contrast, adds a Bloom

Topology	# of Nodes (diameter)	PathDump Overhead	Bloom Filter Overhead	Unroller Avg Time	Unroller Overhead
Stanford	16 (2)	N/A	171 bits	1.74×	25 bits
BellSouth	51 (7)	N/A	189 bits	1.56×	25 bits
GEANT	40 (8)	N/A	608 bits	2.13×	27 bits
ATT-NA	25 (5)	N/A	608 bits	2.15×	27 bits
UsCarrier	158 (35)	N/A	2 466 bits	2.47×	28 bits
FatTree4	20 (4)	64 bits	414 bits	1.73×	28 bits

Table 4.2: Comparison of Unroller and state-of-the-art real-time solutions on real topologies.

filter [9] into packets to store the switch IDs. Other state-of-the-art solutions (as discussed in the included paper [III]) are either unable to detect loops in real-time or have a packet overhead that is linear in the number of hops.

To compare solutions fairly, we used several real WAN or data center topologies [102, 42]. We randomly picked two nodes in each topology in each run and selected the shortest path between them. Out of all possible loops that intersect with that path, we chose one at random to evaluate. The Bloom filter uses a probabilistic data structure to store switch IDs. Thus, it can introduce false positives, as Unroller also does. We evaluated over three million runs and measured the minimum overhead (in bits) needed in each packet so that each solution reports no false positives. Table 4.2 shows the results. PathDump [87] adds a fixed overhead of 64 bits on each packet. It does not experience false positives but can only be applied to a very limited set of topologies [87], e.g., FatTree. Unroller can detect loops without experiencing any false positives using a very small packet overhead. Depending on the topology, it requires 6x to 100x fewer bits than the Bloom filter counterpart. This comes at the expense of detection speed. While the Bloom filter can identify a loop as soon as a switch is hit twice by the same packet, Unroller might require one or two extra passes over the loop, as reported in the *Avg Time* column in Table 4.2. A naive approach using INT (In-band Network Telemetry) would require packets to store an increasing number of switch IDs at each hop, making this approach significantly more expensive (in terms of per-packet bit overhead) than those previously discussed.

To analyze Unroller's resources requirements, we compiled it into three different FPGA-based targets supporting 100 GbE ports: Xilinx Virtex 7 (model XC7VH580T) [2], Xilinx UltraScale+ (model XCVU7P) [92] and Intel Stratix 10 (model 1SG280HU). Table 4.3 shows the chip occupancy and the maximum frequency for all the platforms. Here, we can see that Unroller logic is lightweight, requiring less than 8 % of chip resources, and does not store any flow information at switches. Thus, it does not need any extensive on-chip memory at all. Unroller stores all the required data, i.e., the switch identification, with a minimal overhead directly on the incoming packet. Nevertheless, a small amount of BlockRAM memory is needed to store an arithmetic lookup table to implement specific operations, such as division or power evaluation, that are not natively supported by hardware (in case the base is not a power of two). Since the synthesized architecture is fully pipelined, i.e., capable of processing a new packet every clock cycle, the frequency can be directly correlated with the maximum achievable throughput: ~220 Mpps for Xilinx devices, and ~190 Mpps for the Intel platform. This is more than 100 Gbps for minimum-sized Ethernet packets. Given that the targets are dimensioned for 100 GbE processing, the Unroller logic does not introduce any throughput degradation. Moreover, the P4 implementation can potentially

Platform	LUTs	Regs	BlockRAM	Frequency
Virtex 7	26 234 (7.23 %)	29 944 (4.13 %)	396 kb (1.17 %)	224 MHz
Virtex US+	26 221 (7.23 %)	30 520 (4.21 %)	684 kb (2.02 %)	225 MHz
Stratix 10	21 917 (1.17 %)	45 907 (1.22 %)	301 kb (0.12 %)	189 MHz

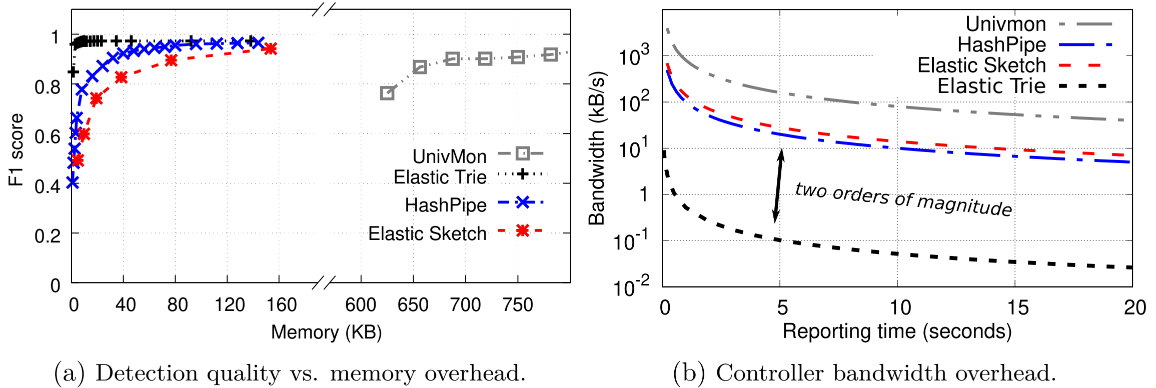
Table 4.3: HW resources utilized and frequency achieved by Unroller implementation.

scale up to much higher speeds if compiled into high-speed programmable switches that support Tbps-scale traffic.

While Unroller enables real-time identification of routing loops, Elastic Trie allows the detection of traffic aggregates within the data plane. In contrast to Unroller, the Elastic Trie algorithm also maintains flow state information at switches. However, it revisits precisely the same idea of triggering the controller only when a specific network event is detected. Elastic Trie can spot changes in traffic patterns and detect high-volume traffic clusters like heavy hitters and superspreaders. We implemented it in the P4 language, too. Its data structure can be well transformed into a hardware architecture based on a set of hash tables. Moreover, unlike similar solutions, Elastic Trie is entirely implemented within the data path without additional software control.

Figure 4.3a compares heavy hitter detection capabilities of Elastic Trie against other state-of-the-art solutions: HashPipe [79], Elastic Sketch [98] and UnivMon [51]. All of them employ probabilistic data structures. Thus, they can introduce false positives, as Elastic Trie does. We utilized real packet traces from CAIDA [3, 4], and to estimate detection quality, we adopted the F_1 score metric. To reach a F_1 score near 1.0, HashPipe needs a lower amount of memory (~100 KB) than Elastic Sketch (~140 KB), which is still much lower than the amount needed by UnivMon (more than 800 KB). Nevertheless, HashPipe can only detect heavy hitters, while UnivMon is not restricted to a single network event, but requires 90 % more memory to work. In contrast, Elastic Sketch ignores mice flows in its heavy flow mechanism, while Elastic Trie can aggregate these flows into shorter prefixes, which results in more accurate detection. At the same time, thanks to the adopted trie-based data structure, Elastic Trie enables traffic pattern change detection. Moreover, it significantly outperforms other solutions with less than 20 KB of the memory footprint.

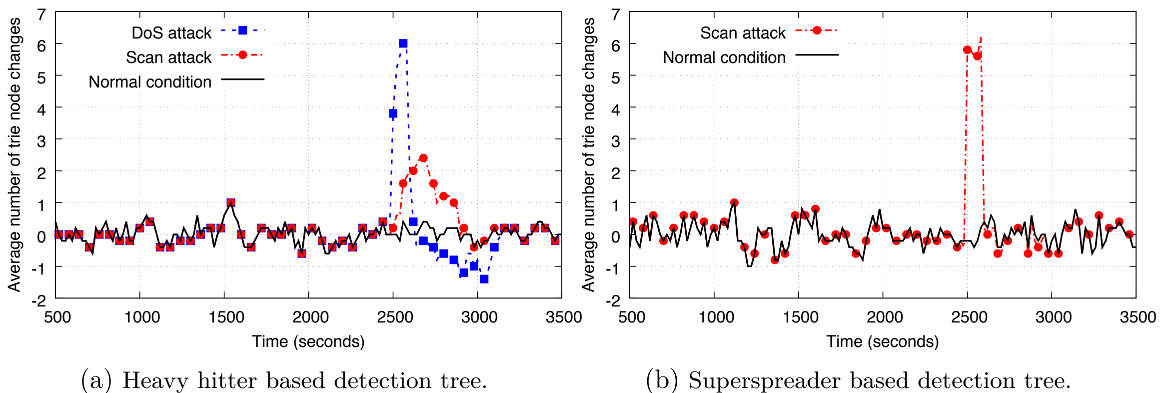
Figure 4.3b compares the data exchanged between a switch and an external controller when Elastic Trie, Univmon, Elastic Sketch, or HashPipe are in place. We calculated the



(a) Detection quality vs. memory overhead.

(b) Controller bandwidth overhead.

Figure 4.3: Comparing heavy hitter detection between Elastic Trie and other approaches.



(a) Heavy hitter based detection tree.

(b) Superspreader based detection tree.

Figure 4.4: Change detection capabilities of Elastic Trie data structure.

Platform	LUTs		Regs	Frequency
	Logic	Memory		
Virtex 7	11 088 (3.06 %)	2 880 (2.03 %)	14 104 (1.94 %)	172.4 MHz
Virtex US+	9 135 (1.16 %)	2 641 (0.67 %)	14 103 (0.89 %)	307.9 MHz

Table 4.4: HW resources utilized and frequency achieved by Elastic Trie implementation.

required bandwidth when varying the reporting time window (x-axis). The figure shows that compared to the other solutions, Elastic Trie, with the event-triggered approach, can save a significant amount of control plane traffic (more than two orders of magnitude).

Elastic Trie can also spot sudden traffic changes by tracking the number of nodes expanded or collapsed over time. Figure 4.4 demonstrates the change detection capabilities on a packet trace of a sudden DoS attack and a scanning (in time 2500 s). While the attack is a single source sending a vast amount of traffic to a designated destination, the scan is a source contacting many random destinations. Figure 4.4a shows the time on the x-axis and a moving average of trie changes (difference between number of expanded and collapsed nodes) on the y-axis. The tree is built based on heavy hitter detection. In the figure, we can distinctly see differences between normal conditions and the state under the DoS attack or scan. In Figure 4.4b, the tree is built on top of the superspreader detection. The DoS attack is undetected in this case because it represents a communication with only one distinct destination. On the other hand, the scan, as a typical case of superspreader, is much more significant now.

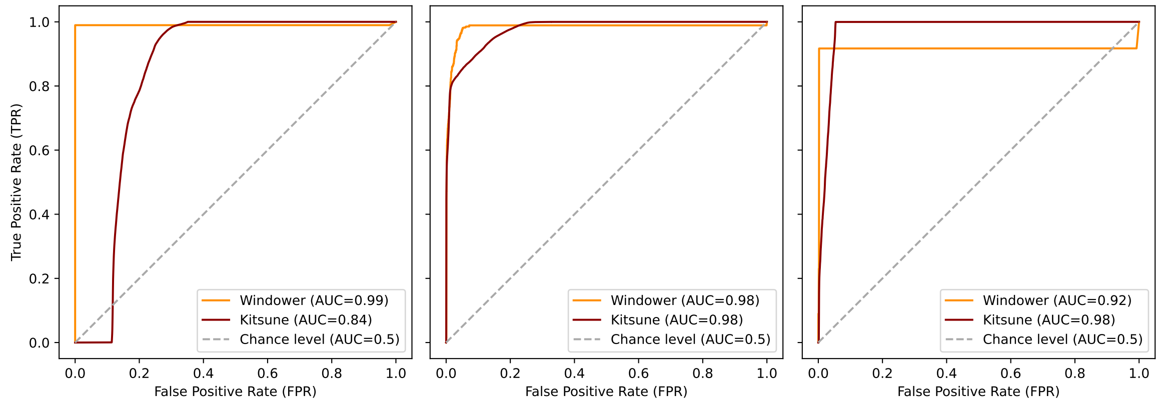
Finally, to quantify Elastic Trie's hardware resource requirements, we also created its VHDL implementation for two different FPGA platforms. Specifically, we synthesized the architecture for Xilinx Virtex-7 (model XC7VH580T) [2] and Virtex UltraScale+ (model XCVU7P) [92]. For FPGA target platforms, we mapped the LPM stage and main memory records into one memory block and utilized distributed memory to implement it as 32 parallel hash tables. Table 4.4 shows the chip occupancy and frequency of the design for both platforms. The total latency introduced by the design is seven clock cycles. The achieved operating frequency corresponds to 40.6 ns for Virtex-7 and 22.75 ns for UltraScale+. At the same time, the architecture can process a newly arrived packet every four clock cycles (the first three out of a total of seven stages are pipelined). This results in an overall throughput of 43.10 Mpps for Virtex-7, and almost twice as much 76.97 Mpps, for the UltraScale+ platform.

4.3 Real-time DDoS Mitigation

The interaction between the control and data plane can be expensive and imply undesirable delays for applications like DDoS detection and mitigation. Specifically, DDoS attacks are one of the increasing cybersecurity threats that intend to cause a target system to become unavailable and thus require a real-time response. Once the anomalous traffic can be detected autonomously in the data plane, it can also be quickly redirected or filtered appropriately without the need to communicate with the controller. In the rest of the thesis, we thus focused more extensively on the DDoS detection and mitigation use case using data plane algorithms and programmability.

We implemented three data plane-based mitigation techniques as a possible countermeasure against TCP SYN Flood, one of the most widespread DoS attack types. The proposed algorithms are used within the data plane only as a reactive defense against ongoing cyber-attacks, hence not affecting the traffic when no threats are detected. They utilize TCP three-way handshake to establish a security association with a client before forwarding its SYN data. The mechanism can efficiently distinguish a legitimate client from an attacker without storing any state information locally. Instead, the client's authentication is performed solely on the value stored in the SYN-ACK reply. The first SYN from a new non-authenticated client is dropped, and a spoofed reply is used to verify its validity. While the regular client will send a valid reply, the attacker without a valid TCP/IP stack will not. The validated client's IP address is then whitelisted, and SYN traffic originating such IP addresses is forwarded without further tampering. This comes at the cost of the delayed first connection (up to 1s), but all subsequent SYN segments experience no significant additional latency (< 0.2 ms). Though this delay may seem high, it is essential to realize that these methods are activated when only an ongoing attack is detected. Therefore, no delays occur during a regular operation, and a slight initial delay is highly preferable to service unavailability while an attack is in progress. Software-based implementation achieved a throughput of more than 97 Mpps on eight independent queues and can scale up to much higher speeds on multiple CPU cores.

Finally, a data plane-based feature extraction and an algorithm for real-time DDoS detection (Windower) was introduced. It employs a sliding window principle to aggregate statistical information from network packets. Summarized statistics are then fed into a ML-based attack discriminator. Windower achieves promising results on public datasets, detecting ongoing attacks and mitigating 99 % of flood and 92 % of slow attacks within the first six seconds while maintaining false positive rate bellow 1 %. Figure 4.5 compares Windower (orange) against Kitsune [58] (red) in terms of mitigation performance. In cases of CAIDA dataset (Figure 4.5a), Windower significantly outperforms Kitsune, which introduces a substantial amount of false positives (above 30%) to detect a similar number of malicious packets. The mitigation performance is more or less comparable for UNSW-NB15 (Figure 4.5b) and SUEE-2017 (Figure 4.5c) datasets. However, Windower still introduces fewer false positives with significantly fewer computational requirements. While Kitsune must still evaluate each incoming packet, Windower substantially lower the number of ML-model evaluations by using statistical features computed across multiple sliding windows. For this reason, Windower is fundamentally faster (the speed-up of more than an order of magnitude) and still it offers similar, or even better, mitigation performance known from packet-based solutions.



(a) CAIDA dataset. (b) UNSW-NB15 dataset. (c) SUEE-2017 dataset.

Figure 4.5: Windower performance on public datasets depicted using ROC curves.

Chapter 5

Discussion and Conclusions

This chapter summarizes all the thesis outcomes and describes the practical impact of the research findings into practice. Based on the results and contribution presented in the previous sections, we conclude that the thesis fulfilled the iniatial research objectives.

The thesis introduced multiple hardware architectures and algorithms for network data plane-based acceleration and optimization. It proposed a concept of flexible flow-based acceleration for network monitoring and intrusion detection and a concept of in-band network events detection and packet feature extraction, particulary for real-time DDoS mitigation.

The published journal and conference papers included in the thesis have focused on three network monitoring and security areas:

Flexible software-defined monitoring – presented architecture assembles the Software Defined Monitoring (SDM) concept. The architecture was presented in the IEEE Transaction on Computers journal with impact factor 3.7. General flow-based acceleration of intrusion detection systems built on top of the SDM architecture was introduced at the IEEE ICCD'18 conference. It enables processing of up to 5× higher link speeds while maintaining high detection quality compared to a non-accelerated system. The journal paper [I] has attracted 21 citations. The follow-up conference paper [II] has attracted three citations.

Event-triggered data plane monitoring – two architectures are presented. Unroller, a solution for real-time identification of routing loops in the data plane, requires 6x to 100x fewer bits added to packets than existing methods. It was presented at the CoNEXT'20 (Core A) conference. The paper [III] attracted 12 citations. Elastic Trie, a push-based approach for network monitoring and detecting network traffic aggregates within the data plane, reduces controller communication overheads by up to two orders of magnitude with respect to other solutions. It was introduced at the SIGCOMM SOSR'20 conference. The paper [IV] has attracted 23 citations so far.

Real-time DDoS mitigation – presented network-based mitigation algorithms provide an efficient countermeasure against DoS attacks. The TCP SYN authentication and SYN Flood mitigation techniques were presented at IFIP/IEEE IM'21 conference. The paper [V] has attracted three citations. Windower, a feature extraction method for real-time network-based DDoS detection using machine learning, outperforms state-of-the-art while improving the runtime performance by more than an order of magnitude. The corresponding paper [VI] was presented at the IFIP/IEEE NOMS'24 conference.

The research findings brought significant improvements in network security monitoring. The informed packet pre-filtering highly reduced the volume of network traffic that must be analyzed and allowed to concentrate the available resources better to achieve throughputs up to 100 Gbps. Thus, the implementation results were later practically integrated as a part of monitoring infrastructure at the CESNET association, the Czech NREN (National Research and Education Network) operator. The architecture implementing the Software Defined Monitoring (SDM) concept for enhanced monitoring of application protocols was tested and deployed in the real-life network infrastructure of the CESNET2 academic network. The live demonstration of application layer traffic monitoring at 100 Gbps [R2] was later presented at the IEEE LCN'15 conference. The implemented SDM architecture was subsequently licensed and commercialized by Netcope Technologies (now Magmio) and was available as a part of their packet capture solutions portfolio.

Moreover, the first implementation results inspired other research and development activities. At that time, there was a strong need not only passively to monitor the CESNET2 network but also to protect it against the threat of volumetric DDoS attacks actively. The original FPGA-based architecture and software implementation of a DDoS protection system was built at CESNET in the scope of the DCPro project (Technology for Protection of High-Speed Networks) funded by the Technology Agency of the Czech Republic. Its P4 implementation [R5] was later presented at the ACM/IEEE ANCS'19 conference. This research led to the publishing of the following papers included in this thesis and continued in AdaptDDoS (Adaptive Protection against DDoS Attacks) [75] and DoSIX (Distributed DDoS mitigation in Critical Infrastructure Environment) [76] projects funded by the security research programs of the Ministry of the Interior of the Czech Republic. I operated as the lead investigator and manager of both projects. In the scope of these projects, a unique system, DDoS Protector [25], was developed. It is a complex solution of protection against DDoS attacks that integrates many ideas and concepts initially proposed by the original research described in this thesis.

The system currently protects the Czech academic backbone network and is also deployed in operation at NIX.CZ, the Czech largest neutral Internet Exchange Point. Moreover, the research results have been commercialized by companies NetX Networks [74] and BrnoLogic [14]. Promising performance results achieved by the system have also sparked an interest in the potential deployment by other prominent internet service providers, critical information infrastructure operators, and organizations responsible for ensuring the state's cyber security.

5.1 Future Work

The speed of computer networks is continually increasing. In the last few years, development in this area has been more strongly motivated by the massive emergence of AI (Artificial Intelligence) and increasing demands for networking infrastructure performance at data centers. For example, NVIDIA recently announced [64] its new Quantum/Spectrum X800 technology of programmable network switches and ConnectX-8 smart network interface cards with the support of end-to-end 800 Gbps throughput for massive scaling of AI models and HPC (High-Performance Computing) in data centers.

The ONF (Open Networking Foundation), a consortium of the largest networking companies, considers the area of highly flexible programmable network devices to be the upcoming standard. Smart NICs and programmable switches are being applied to accelerate network functions, especially in large data centers. However, their use in the network

backbone infrastructure is not yet quite common. Thus, the current trend will lead to higher availability, lower cost, and gradual deployment in other segments of the network infrastructure. Therefore, it is essential to investigate this area further in future research.

From the cybersecurity perspective, the landscape of DDoS attacks evolves too. Anti-DDoS solutions still have difficulties catching up with the scale and rapid development of attack vectors. Hardware-based middleware boxes suffer from insufficient flexibility in terms of capacity and functionality. The data plane programmability in the cybersecurity segment thus can support future mitigation functionalities at Tbps speeds without additional hardware upgrades. However, P4 is still a relatively low-level language requiring deep expertise. For practical use, especially for dynamic defense mechanisms, it is necessary to create high-level primitives to describe the required filtering and mitigation functionality.

Such research will be partially addressed by the currently running HSPF (High-speed Packet Filtering) [77] project, which is part of the security research program for developing, testing, and evaluating new security technologies (SECTECH) funded by the Ministry of the Interior of the Czech Republic.

Bibliography

[1] *30 years online! On February 13th, 1992, our country joined the Internet. We celebrated the anniversary at a conference. Famous Vint Cerf also sent his greetings.* https://www.cesnet.cz/en/about-us/tiskove-zpravy-1/pred-30-lety-se-ceska-republika-pripojila-k-internetu-vyroci-dnes-pripomina-konference-na-ktere-na-dalku-promluvil-otec-internetu-vint-cerf-36. CESNET, Feb. 2022.

[2] *7 Series FPGAs Overview, Data Sheet 180 (v2.6).* https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. XILINX, Sept. 2020.

[3] *Anonymized Internet Traces 2009 – 17th September 2009.* http://www.caida.org/data/passive/passive_2009_dataset.xml. CAIDA, Jan. 2018.

[4] *Anonymized Internet Traces 2016 – 17th March 2016.* http://www.caida.org/data/passive/passive_2016_dataset.xml. CAIDA, Jan. 2018.

[5] *Application Layer Packet Classifier for Linux.* https://l7-filter.sourceforge.net/. ClearFoundation, Jan. 2009.

[6] Theophilus Benson et al. “MicroTE: Fine Grained Traffic Engineering for Data Centers”. In: *COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2011.

[7] Aanshi Bhardwaj et al. “Distributed denial of service attacks in cloud: State-of-the-art of scientific and commercial solutions”. In: *Computer Science Review*. Volume: 39. Elsevier, 2021.

[8] Monowar Bhuyam, Dhruba Kumar Bhattacharyya, and Jugal Kalita. “Network Anomaly Detection: Methods, Systems and Tools”. In: *Communications Surveys Tutorials*. Volume: 16, Issue: 1. IEEE, 2014.

[9] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Communications of the ACM (CACM)*. Volume: 13, Issue: 7. ACM, 1970.

[10] Pat Bosshart et al. “Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN”. In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

[11] Pat Bosshart et al. “P4: Programming Protocol-independent Packet Processors”. In: *Computer Communication Review*. Volume: 44, Issue: 3. ACM, 2014.

[12] Raouf Boutaba et al. “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities”. In: *Journal of Internet Services and Applications*. Volume: 9, Issue: 1. Springer, 2018.

[13] *BPF and XDP Reference Guide*. https://cilium.readthedocs.io/en/latest/bpf/. Apr. 2024.

[14] *Brno Design House for programmable logic and its verification*. https://brnologic.com/. BrnoLogic, July 2024.

[15] Christian Callegari et al. “Detecting Anomalies in Backbone Network Traffic: A Performance Comparison Among Several Change Detection Methods”. In: *International Journal of Sensor Networks*. Volume: 11, Issue: 4. Inderscience Publishers, 2012.

[16] Calin Cascaval and Dan Daly. “P4 Architectures”. In: *4th P4 Workshop*. https://opennetworking.org/wp-content/uploads/2020/12/p4-ws-2017-p4-architectures.pdf. 2017.

[17] J. D. Case et al. *Simple network management protocol (SNMP)*. RFC 1157. https://www.ietf.org/rfc/rfc1157.txt. IETF, 1990.

[18] Tomáš Čejka et al. “NEMEA: A Framework for Network Traffic Analysis”. In: *International Conference on Network and Service Management (CNSM)*. IEEE, 2016.

[19] Baek-Young Choi, Jaesung Park, and Zhi-li Zhang. “Adaptive random sampling for traffic load measurement”. In: *International Conference on Communications (ICC)*. IEEE, 2003.

[20] B. Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954 (Informational). https://www.ietf.org/rfc/rfc3954.txt. IETF, 2004.

[21] B. Claise, B. Trammell, and P. Aitken. *Specification of the IP Flow Information Export (IPFIX) protocol for the exchange of flow information*. RFC 7011 (Internet Standard). https://www.ietf.org/rfc/rfc7011.txt. IETF, 2013.

[22] Graham Cormode et al. “Finding Hierarchical Heavy Hitters in Streaming Data”. In: *Transactions on Knowledge Discovery from Data*. Volume: 1, Issue: 4. ACM, 2008.

[23] Andrew R. Curtis et al. “DevoFlow: Scaling Flow Management for High-performance Networks”. In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2011.

[24] David Day and Benjamin Burns. “A Performance Analysis of Snort and Suricata Network Intrusion Detection and Prevention Engines”. In: *International Conference on Digital Society (ICDS)*. IARIA, 2011.

[25] *DDoS Protector*. https://www.liberouter.org/technologies/ddos-protector/. CESNET, Apr. 2024.

[26] Luca Deri et al. “nDPI: Open-source high-speed deep packet inspection”. In: *International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2014.

[27] Damu Ding et al. “In-Network Volumetric DDoS Victim Identification Using Programmable Commodity Switches”. In: *Transactions on Network and Service Management*. Volume: 18, Issue: 2. IEEE, 2021.

[28] Rohan Doshi, Noah Apthorpe, and Nick Feamster. “Machine Learning DDoS Detection for Consumer Internet of Things Devices”. In: *Security and Privacy Workshops (SPW)*. IEEE, 2018.

[29] *DPDK: Data Plane Development Kit*. https://www.dpdk.org/. Apr. 2024.

[30] Juliette Dromard, Gilles Roudière, and Philippe Owezarski. "Online and Scalable Unsupervised Network Anomaly Detection Method". In: *Transactions on Network and Service Management*. Volume: 14, Issue: 1. IEEE, 2017.

[31] Cristian Estan and George Varghese. "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice". In: *Transactions on Computer Systems*. Volume: 21, Issue: 3. ACM, 2003.

[32] *Ethernet's Next Bar is Now – 800 Gb/s!* https://standards.ieee.org/beyond-standards/ethernets-next-bar/. IEEE, Apr. 2024.

[33] Štěpán Friedl et al. *Designing a Card for 100 Gb/s Network Monitoring*. Technical Report. https://www.cesnet.cz/wp-content/uploads/2014/02/card.pdf. CESNET, July 2013.

[34] Rick Hofstede et al. "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX". In: *Communications Surveys and Tutorials*. Volume: 16, Issue: 4. IEEE, 2014.

[35] Toke Høiland-Jørgensen et al. "The eXpress data path: fast programmable packet processing in the operating system kernel". In: *COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2018.

[36] Qun Huang, Patrick P. C. Lee, and Yungang Bao. "Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[37] *In-band Network Telemetry (INT) Dataplane Specification (Version 2.1)*. https://p4.org/p4-spec/docs/INT_v2_1.pdf. The P4.org Applications Working Group, Nov. 2020.

[38] *Intel Smart NICs*. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html. Intel Corporation, Apr. 2024.

[39] *Intel® Tofino™*. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html. Intel Corporation, Apr. 2024.

[40] Lavanya Jose, Minlan Yu, and Jennifer Rexford. "Online Measurement of Large Traffic Aggregates on Commodity Switches". In: *Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*. USENIX, 2011.

[41] Lukáš Kekely, Viktor Puš, and Jan Kořenek. "Software Defined Monitoring of Application Protocols". In: *International Conference on Computer Communications (INFOCOM)*. IEEE, 2014.

[42] Simon Knight et al. "The Internet Topology Zoo". In: *Journal on Selected Areas in Communications*. Volume: 29, Issue: 9. IEEE, 2011.

[43] Vlastimil Košař, Martin Žádník, and Jan Kořenek. "NFA reduction for regular expressions matching using FPGA". In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013.

[44] Balachander Krishnamurthy et al. "Sketch-based Change Detection: Methods, Evaluation, and Applications". In: *Internet Measurement Conference (IMC)*. ACM, 2003.

[45] Anukool Lakhina, Mark Crovella, and Christophe Diot. "Diagnosing Network-wide Traffic Anomalies". In: *Computer Communication Review*. Volume: 34, Issue: 4. ACM, 2004.

[46] Anukool Lakhina, Mark Crovella, and Christophe Diot. “Mining Anomalies Using Traffic Feature Distributions”. In: *Computer Communication Review*. Volume: 35, Issue: 4. ACM, 2005.

[47] Kun-Chan Lan and John Heidemann. “A measurement study of correlation of Internet flow characteristics”. In: *Computer Networks: The International Journal of Computer and Telecommunications Networking*. Volume: 17, Issue: 1. Elsevier, 2006.

[48] Yan Li and Yifei Lu. “LSTM-BA: DDoS Detection Approach Combining LSTM and Bayes”. In: *International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 2019.

[49] Yuliang Li et al. “FlowRadar: A Better NetFlow for Data Centers”. In: *Networked Systems Design and Implementation (NSDI)*. USENIX, 2016.

[50] Zaoxing Liu et al. “Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches”. In: *Usenix Security Symposium (USENIX Security)*. USENIX, 2021.

[51] Zaoxing Liu et al. “One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon”. In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[52] Ratul Mahajan et al. “Controlling high bandwidth aggregates in the network”. In: *Computer Communication Review*. Volume: 32, Issue: 3. ACM, 2002.

[53] Jianning Mai et al. “Is Sampled Data Sufficient for Anomaly Detection?” In: *Internet Measurement Conference (IMC)*. ACM, 2006.

[54] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *Computer Communication Review*. Volume: 38, Issue: 2. ACM, 2008.

[55] *Mellanox Smart NICs*. https://www.nvidia.com/en-us/networking/ethernet-adapters/. NVIDIA, Apr. 2024.

[56] Sebastiano Miano et al. “Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case”. In: *IEEE Access*. Volume: 7. IEEE, 2019.

[57] Oliver Michel et al. “The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications”. In: *Computing Surveys*. Volume: 54, Issue: 4. ACM, 2021.

[58] Yisroel Mirsky et al. “Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection”. In: *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[59] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. “Compiling PCRE to FPGA for Accelerating SNORT IDS”. In: *Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM, 2007.

[60] Meenakshi Mittal, Krishan Kumar, and Sunny Behal. “Deep learning approaches for detecting DDoS attacks: a systematic review”. In: *Soft Computing*. Volume: 27, Issue: 18. Springer, 2022.

[61] Biswanath Mukherjee, L.T. Heberlein, and K.N. Levitt. “Network intrusion detection”. In: *IEEE Network*. Volume: 8, Issue: 3. IEEE, 1994.

[62] *Netronome Agilio CX SmartNICs*. https://netronome.com/agilio-smartnics/. Netronome, Apr. 2024.

[63] *NPL – Open, High-Level language for developing feature-rich solutions for programmable networking platforms.* https://nplang.org/. Apr. 2024.

[64] *NVIDIA Announces New Switches Optimized for Trillion-Parameter GPU Computing and AI Infrastructure.* https://nvidianews.nvidia.com/news/networking-switches-gpu-computing-ai. NVIDIA, Mar. 2024.

[65] *P4.* http://p4.org/. Apr. 2024.

[66] *P4Runtime Specification (Version 1.3.0).* https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html. The P4.org API Working Group, Apr. 2021.

[67] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo Hashing". In: *Annual European Symposium on Algorithms (ESA)*. Springer, 2001.

[68] Vern Paxson. "Bro: A System for Detecting Network Intruders in Real-time". In: *Usenix Security Symposium (USENIX Security)*. USENIX, 1998.

[69] *PCIe boards based on FPGAs.* https://www.reflexces.com/pcie-boards. Reflex-CES, Apr. 2024.

[70] *Programmable FPGA Server Adapter – Connectivity Solutions.* https://www.silicom-usa.com/cats/server-adapters/programmable-fpga-server-adapter/. Silicom, Apr. 2024.

[71] Luigi Rizzo. "Netmap: a novel framework for fast packet I/O". In: *Annual Technical Conference (ATC)*. USENIX, 2012.

[72] Martin Roesch. "Snort – Lightweight Intrusion Detection for Networks". In: *Large Installation System Administration Conference (LISA)*. USENIX, 1999.

[73] Mohammad A. Salahuddin et al. "Chronos: DDoS Attack Detection Using Time-Based Autoencoder". In: *Transactions on Network and Service Management.* Volume: 19, Issue: 1. IEEE, 2022.

[74] *Security & DDoS Protection.* https://netx.as/netx-os#security. NetX Networks, Apr. 2024.

[75] *Security research project AdaptDDoS (Adaptive Protection against DDoS Attacks).* https://starfos.tacr.cz/en/projekty/VI20192022137. CESNET, Apr. 2024.

[76] *Security research project DoSIX (Distributed DDoS Mitigation in Critical Infrastructure Environment).* https://starfos.tacr.cz/en/projekty/VB01000015. CESNET, Apr. 2024.

[77] *Security research project HSPF (High-speed Packet Filtering).* https://starfos.tacr.cz/en/projekty/VB02000066. CESNET, Apr. 2024.

[78] *sFlow.* http://www.sflow.org/about/index.php. Apr. 2024.

[79] Vibhaalakshmi Sivaraman et al. "Heavy-Hitter Detection Entirely in the Data Plane". In: *Symposium on SDN Research (SOSR)*. ACM, 2017.

[80] *SmartNIC and IPU Hardware Portfolio – Product Overview.* https://www.napatech.com/support/resources/data-sheets/smartnic-and-ipu-hardware-portfolio/. Napatech, Apr. 2024.

[81] *Snort - Network Instrusion Detection & Prevention System.* https://www.snort.org/. CISCO, Apr. 2024.

[82] *Software-Defined Networking: The New Norm for Networks*. https://opennetworking.org/wp-content/uploads/2011/09/wp-sdn-newnorm.pdf. Open Networking Foundation, Apr. 2012.

[83] Haoyu Song et al. “Snort offloader: a reconfigurable hardware NIDS filter”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2005.

[84] A. Sperotto et al. “An Overview of IP Flow-Based Intrusion Detection”. In: *Communications Surveys Tutorials*. Volume: 12, Issue: 3. IEEE, 2010.

[85] Vincent Stoffer, Aashish Sharma, and Jay Krous. *100G Intrusion Detection*. http://go.lbl.gov/100g. Lawrence Berkeley National Laboratory, 2015.

[86] *Suricata – Open Source IDS / IPS / NSM engine*. https://suricata.io/. OISF, Apr. 2024.

[87] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. “Simplifying Datacenter Network Debugging with Pathdump”. In: *Operating Systems Design and Implementation (OSDI)*. USENIX, 2016.

[88] Lizhuang Tan et al. “In-band Network Telemetry: A Survey”. In: *Computer Networks*. Volume: 186. Elsevier, 2021.

[89] Ankit Thakkar and Ritika Lohiya. “A survey on intrusion detection system: feature selection, model, performance measures, application perspective, challenges, and future research directions”. In: *Artificial Intelligence Review*. Volume: 55, Issue: 1. Springer, 2022.

[90] *The Zeek Network Security Monitor*. https://www.zeek.org/. The Zeek Project, Apr. 2024.

[91] *Trident4 / BCM56880 Series*. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series. Broadcom, Apr. 2024.

[92] *UltraScale Architecture and Product Overview, Data Sheet 890 (v3.6)*. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. AMD, Mar. 2024.

[93] Shobha Venkataraman et al. “New Streaming Algorithms for Fast Detection of Superspreaders”. In: *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2005.

[94] Ke Wang and Salvatore J. Stolfo. “Anomalous Payload-Based Network Intrusion Detection”. In: *Recent Advances in Intrusion Detection (RAID)*. Springer, 2004.

[95] Nicholas Weaver, Vern Paxson, and Jose M. Gonzalez. “The Shunt: An FPGA-based Accelerator for Network Intrusion Prevention”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2007.

[96] *What Is a SmartNIC?* https://blogs.nvidia.com/blog/what-is-a-smartnic/. NVIDIA, Oct. 2021.

[97] Yinglian Xie et al. “Worm Origin Identification Using Random Moonwalks”. In: *Security and Privacy (SP)*. IEEE Computer Society, 2005.

[98] Tong Yang et al. “Elastic Sketch: Adaptive and Fast Network-wide Measurements”. In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[99] Minlan Yu. “Network Telemetry: Towards A Top-Down Approach”. In: *Computer Communication Review*. Volume: 49, Issue: 1. ACM, 2019.

[100] Minlan Yu, Lavanya Jose, and Rui Miao. “Software Defined Traffic Measurement with OpenSketch”. In: *Conference on Networked Systems Design and Implementation (NSDI)*. USENIX, 2013.

[101] Xiaoyong Yuan, Chuanhuang Li, and Xiaolin Li. “DeepDefense: Identifying DDoS Attack via Deep Learning”. In: *International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2017.

[102] James Hongyi Zeng and Peyman Kazemian. *Mini-Stanford Backbone*. https://reproducingnetworkresearch.wordpress.com/2012/07/11/atpg/. 2012.

[103] Menghao Zhang et al. “Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches”. In: *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[104] Zhipeng Zhao et al. “Achieving 100Gbps intrusion prevention on a single server”. In: *Conference on Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

[105] Noa Zilberman et al. “NetFPGA SUME: Toward 100 Gbps as Research Commodity”. In: *IEEE Micro*. Volume: 34, Issue: 5. IEEE, 2014.

Appendix A

Included Papers

A.1 Paper I

Software Defined Monitoring of Application Protocols

Lukáš KEKELY, Jan KUČERA, Viktor PUŠ, Jan KOŘENEK and Athanasios V. VASILAKOS. "Software Defined Monitoring of Application Protocols". In: *IEEE Transactions on Computers* 65.2 (2016), pp. 615–626. ISSN: 0018-9340.

Software Defined Monitoring of Application Protocols

Lukáš Kekely, Jan Kučera, Viktor Puš, Jan Kořenek, and Athanasios V. Vasilakos

Abstract—With the ongoing shift of network services to the application layer also the monitoring systems focus more on the data from the application layer. The increasing speed of the network links, together with the increased complexity of application protocol processing, require a new way of hardware acceleration. We propose a new concept of hardware acceleration for flexible flow-based application level traffic monitoring which we call Software Defined Monitoring. Application layer processing is performed by monitoring tasks implemented in the software in conjunction with a configurable hardware accelerator. The accelerator is a high-speed application-specific processor tailored to stateful flow processing. The software monitoring tasks control the level of detail retained by the hardware for each flow in such a way that the usable information is always retained, while the remaining data is processed by simpler methods. Flexibility of the concept is provided by a plugin-based design of both hardware and software, which ensures adaptability in the evolving world of network monitoring. Our high-speed implementation using FPGA acceleration board in a commodity server is able to perform a 100 Gb/s flow traffic measurement augmented by a selected application-level protocol analysis.

Index Terms—Network monitoring, acceleration, security, FPGA, L7

1 Introduction

Modern network engineering and security heavily rely on the network traffic monitoring. The requirements imposed on the quality of network security monitoring information often lead to the requirement to process unsampled network traffic. That ability is crucial in order to detect even single-packet attacks. A golden standard in the area of network monitoring is a flow measurement. A monitoring device collects basic statistics about the network flows at the Internet and Transport layers and reports them to a central storage collector using a handover protocol such as NetFlow [1] or IPFIX [2]. Flow measurement is a stateful process, because for each packet the flow state record is updated in the device (e.g. packet counters are incremented), and only the resulting numbers are exported. This also implies that some information is lost in the monitoring process and that the flow collector (where further data processing is usually done) has a limited view on the network. While a number of researchers focus on harvesting knowledge from the existing flow data, we argue that the ability to analyze *application layer* in the monitoring process is crucial for improvement of the quality and flexibility of network monitoring. This is illustrated by the recent infamous Heartbleed bug in the SSL implementation. While it is impractical (if not impossible at all) to detect the Heartbleed attack by analyzing the transport layer flow data, its detection at the application layer is trivial.

The ongoing trend in the field of application layer monitoring is towards creation of richer flow records [3], [4], [5], carrying some extra information in addition to the basic flow size and timing statistics. The added information often include values from the application layer protocol headers, such as HTTP, DNS etc. It seems that the ability to analyze application layer in the monitoring process is crucial for improvement of the quality of network threat detection, because more and more of the network functionality is being shifted up in the protocol stack.

Implementation of the application level flow monitoring with a commodity CPU is certainly possible, yet its throughput is limited mainly by the performance of the processors [6]. It should be noted that every newly arrived packet is inevitably a cache miss in the CPU. On the other hand, ASICs and FPGAs offer much better possibilities in terms of throughput. However, a fixed solely hardware implementation may face the flexibility issues, since the evolving nature of network threats implies the need for fast changes of the monitoring process, quickly making fixed hardware devices obsolete. Many papers proposing high-speed hardware architectures for the most timing-critical operations necessary in flow monitoring were published. Those operations include packet header parsing, packet classification, counters management and pattern matching. However, most of the proposed architectures have never been practically deployed. We conceive that this is because the effort is usually spent only on the improvement of the performance features, but flexibility, ease of use and speed of response to newly emerged problems are neglected.

The aim of this paper is to (1) strike a balance between the system throughput and flexibility/programmability and to (2) offer a configurable trade-off to the above, but

- *L. Kekely, J. Kučera and V. Puš are with CESNET a. l. e., Zikova 4, 160 00 Prague, Czech Republic. E-mail: {kekely, jan.kucera, pus}@cesnet.cz.*
- *J. Kořenek is with the IT4Innovations Centre of Excellence, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic. E-mail: korenek@fit.vutbr.cz.*
- *A. V. Vasilakos is with the Department of Computer Science, Electrical and Space Engineering, Lulea University of Technology, Sweden. E-mail: athanasios.vasilakos@ltu.se.*

Manuscript received 17 June 2014; revised 20 Feb. 2015; accepted 6 Apr. 2015. Date of publication 15 Apr. 2015; date of current version 15 Jan. 2016.
Recommended for acceptance by Y.-D. Lin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2423668

0018-9340 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

mainly to (3) endorse a progressive adoption of network monitoring subtasks to the hardware accelerator, motivated solely by the real needs of the networking community.

Our key idea is that even the advanced application-layer processing task usually need to observe only some network flows, representing only a small fraction of traffic. An example can be a DNS analyzer, since DNS traffic typically represents no more than 1 percent of all network packets. Other applications may utilize even better offload, since they need to observe only a small amount of packets within each flow for their full functionality. Let a HTTP header analyzer be an example, since the HTTP header is typically located in the first few packets of the network flow. Please note that our method never discards packets that are relevant for the particular monitoring application.

We only offload the processing of bulk traffic that is not (or no longer) interesting for the application-layer processing tasks into the hardware accelerator. The offload of measurement is controlled on a per flow basis by the monitoring software and adjusted in real time to its current needs. Offload control is realized through unified interface by dynamically specifying a set of rules. These rules are installed into the accelerator to determine the type of packet offload (=preprocessing acceleration) used for individual network flows. The preprocessing method that best aids the performance and does not decrease the required precision of advanced software processing is selected. Due to the unified control interface the proposed system is very flexible and can be used for a wide range of network monitoring applications.

Furthermore, the whole system is designed to be easily extensible at two different levels. At the software side, monitoring plugins can be easily added to the system. This brings the possibility of rapid development and deployment of new monitoring applications, for example as a reaction to a new network security threat. Once the functionality of software task is verified and stable enough, the second level of system extensibility can be employed to further speed-up the task. Various packet processing and data aggregation routines can be relocated directly into the hardware accelerator.

The paper is structured as follows: The following section provides analysis of real-life network traffic from the application monitoring point of view. In Section 3 we make use of the analysis outcomes to design the concept of hardware accelerator tightly coupled to monitoring software applications. Section 4 provides experimental results of our work. Section 5 presents notable related work and Section 6 concludes our paper.

2 Analysis

We start the paper with an analysis of traffic properties in a real high-speed backbone network. Based on the measured characteristics we then optimize the design of our SDM system to achieve optimal performance when deployed in real networks. All of the measurements in this paper were conducted in the high-speed CESNET backbone network. CESNET is Czech National Research and Educational Network which has optical links operating at speeds up to 100 Gbps and routes mainly IP traffic. It serves around 200,000 users. We conduct all of our measurements during the standard working hours. To get a basic view of the network traffic character, we measure mean size of packets in bytes, mean size of flows in packets and mean time duration of flows. Because we aim for the application protocols, we measure these characteristics not only for the whole network traffic on the link, but also for the selected applications. We select some of the most commonly used application protocols and services such as HTTP, HTTPS, DNS, email (SMTP, POP3 and IMAP), SSH, RTMP, FTP and others. Furthermore, we measure the percentage of these protocols in the captured traffic in terms of flows, packets and bytes.

TABLE 1
Basic Statistical Characteristics of Network Data Grouped by the Service

	Traffic portion in			Average		
	flows [%]	packets [%]	bytes [%]	flow size [packets]	flow time [s]	pckt size [Bytes]
HTTP	26.62	48.33	51.81	59.2	7.137	983.0
HTTPS	18.18	31.12	29.75	51.3	8.591	816.7
SSH	2.66	1.42	1.09	11.7	17.167	241.2
RTMP	0.02	1.01	1.24	2,066.8	57.432	1,001.2
DNS	24.10	0.79	0.19	1.1	0.153	205.9
Email	1.00	0.72	0.56	16.8	2.957	581.6
ICMP	1.91	0.60	0.50	1.9	3.206	91.3
RDP	3.37	0.53	0.31	4.7	2.731	468.4
NTP	1.53	0.41	0.21	8.8	4.142	359.5
FTP	0.38	0.01	0.01	2.3	1.234	75.8
SIP	0.00	0.00	0.00	5.0	23.611	421.1
others	20.23	15.06	14.33	27.2	7.536	839.6
all				32.0	6.432	872.2

Table 1 shows the results of the basic network traffic analysis. The table shows that the observed statistics differ greatly depending on the specific service. The largest portion of network traffic is conveyed by the HTTP protocol which accounts for more than a quarter of all flows and around half of all packets and bytes. Moreover we can see that HTTP flows and packets are generally larger (heavier) in number of packets and bytes and longer in time than average. Another large amount of total traffic belongs to HTTPS, which has very similar observed characteristics as HTTP. These two protocols (HTTP and HTTPS) together cover majority of all network traffic—nearly a half of all flows and around four fifths of the data. Therefore, the possibility of their further analysis is certainly desirable. A large amount of flows also belong to the DNS protocol (nearly one quarter), but this number is highly disproportional to the DNS total packet and bytes percentage. DNS flows are generally very small (light) with majority of them consisting of only one small packet. Also ICMP, which covers majority of non-TCP/non-UDP flows on the network, has similar character of flows as DNS with very small and short flows. The opposite type of disproportional flows and packets percentages as DNS and ICMP has RTMP protocol (Flash multimedia streaming), which covers only a tiny portion of flows, but they are all extremely heavy and long.

The *distribution* of packet lengths is another interesting characteristic of the network. The majority of packets are either very long (over 1.300 B: 57 percent) or very short (under 100 B: 35 percent). Especially dominant are both extremes from the range of lengths supported by the

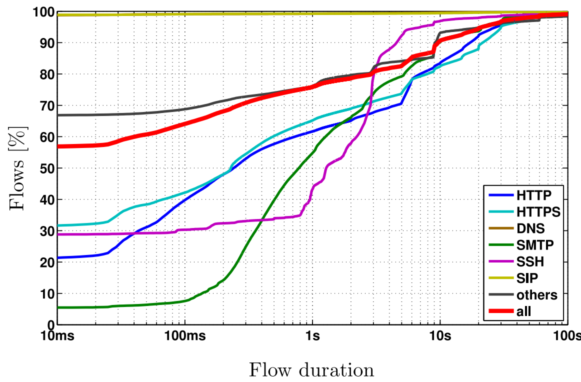


Fig. 1. Cummulative distribution functions of flow durations.

Ethernet standard—42 and 1,500 B. Medium sized packets are not very common.

In Table 1 we have already shown basic information about mean flow durations. Further information about the flow time durations for the selected application protocols can be seen in Fig. 1. Each line in the graph shows the percentage of flows that last shorter than the given duration. Generally (red thick line) over $\frac{2}{3}$ of all flows are shorter than 100 ms and only a tenth of them exceed 10 s. Also majority of DNS and SIP flows have a duration under 10 ms.

While Fig. 1 shows further information about flow duration, it does not say anything about time distribution of packets inside the flows. Weights of individual flows are also not considered. A better look at packet timing inside the flows can be shown by measuring the relative arrival times of packets from the start of the flow. Thus, the first packet of each flow has the zero relative arrival time and its absolute arrival time marks the starting time of that flow. Then, each subsequent packet has a relative arrival time equal to the difference of its absolute arrival time and the marked start of the flow. Results of this measurement are shown in Fig. 2. The graph shows that on average (red thick line) only a small portion of all packets arrive right after the start of the flow—;only a fifth of all packets arrive during the first second of the flow. This fact leads to the conclusion that flows with short duration carry only a very few packets. The conclusion is further strengthened by the fact that the majority of flows have a very short duration.

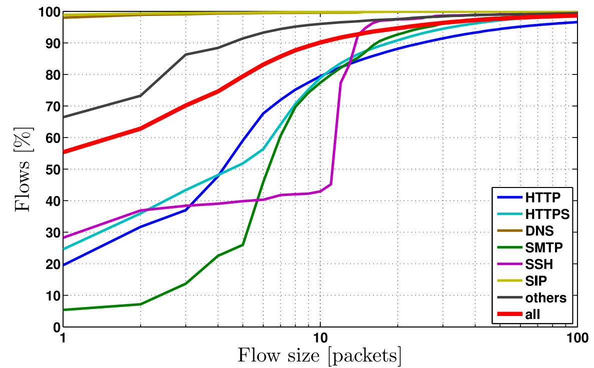


Fig. 3. Cummulative distribution functions of flow sizes.

Table 1 contains the information about mean flow sizes for selected application protocols and services. Further information about flow sizes can be seen in Fig. 3. Each line of the graph shows the percentage of flows that consists of fewer packets than a given number. On average (red thick line) only a tenth of all network flows have more than 10 packets. Also, virtually all DNS and SIP flows consist of a single packet.

Fig. 3 does not clearly say anything about the percentage of all packets carried by flows of different sizes. It is known that high-speed network traffic has a heavy-tailed character of flow size distribution [7], [8]. The heavy-tailed character of flow size distribution derived from the measured values is shown in Fig. 4. The graph shows the portions of all packets carried by the specified percentage of the heaviest flows on the network. It can be seen that on average (red thick line) 0.1 percent of the heaviest flows carry around 60 percent of all packets and 1 percent carry even around 85 percent. An exception to the heavy-tailed distribution of flow sizes is the DNS protocol. On the other hand, SIP and SSH protocols have a heavier tail than average.

Our work relies on the following consequence of the heavy-tailed character of network traffic: by selecting a small percentage of the heaviest flows, we can cover the majority of packets. The problem then lies in an effective prediction of which flows are the heaviest. More accurately, it lies in a capability to recognize the heaviest flows only from the properties of their first few packets. The simplest method of

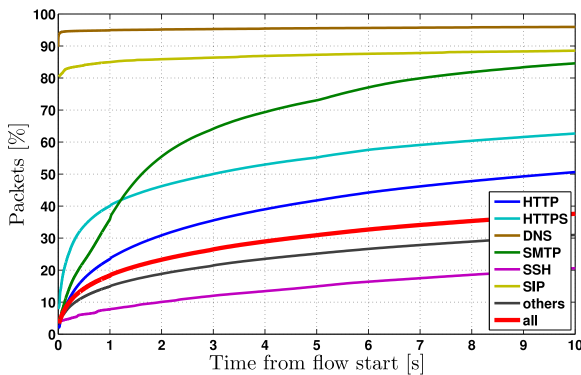


Fig. 2. Cummulative distribution functions of packet arrival times.

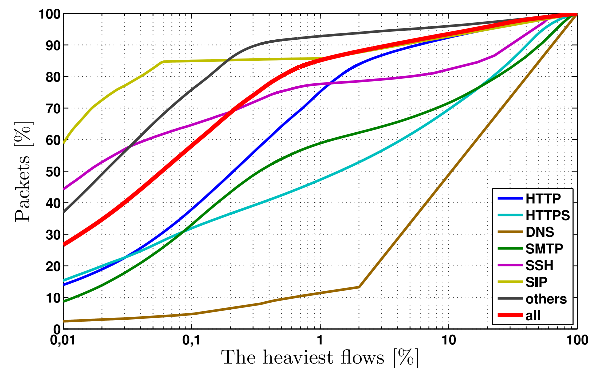


Fig. 4. Portions of packets carried by the percentage of the heaviest flows.

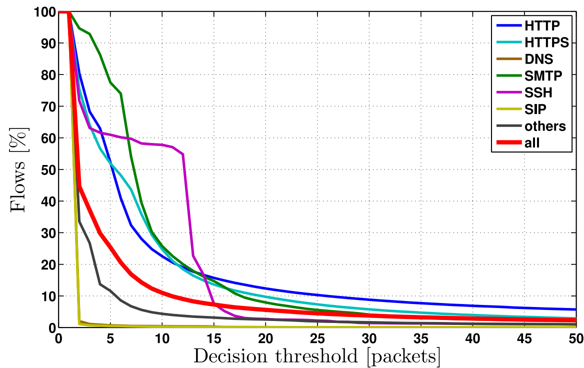


Fig. 5. Heavy flow detection using the simple method—portions of selected flows.

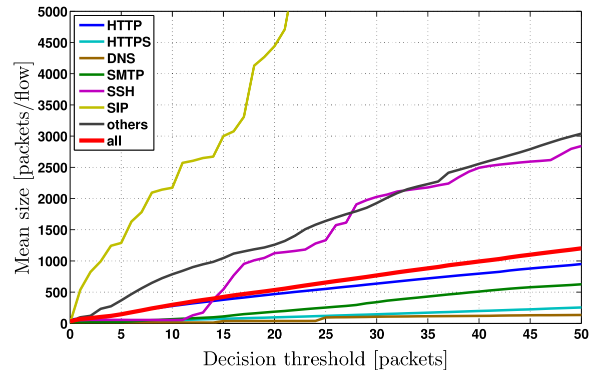


Fig. 7. Mean number of captured packets per flow in flows selected using the simple method.

this recognition is based on a rule that every flow is considered heavy after arrival of its first k packets for some selected decision threshold k. The main advantage of this method is just its simplicity—no additional packet analysis nor advanced stateful information for the flows is needed.

Figs. 5 and 6 show the measured accuracy of the heaviest flow selection by the described simple method. These graphs show the relations between the value of threshold k to the portion of heavy marked flows (first graph) and packets covered by them (second graph). By a combination of values from both graphs we can see that with the rising decision threshold the portion of flows marked heavy dramatically decreases, but the percentage of covered packets decreases rather slowly. For example, decision threshold $k = 20$ leads to only 5 percent of heavy marked flows while covering around 85 percent of all packets on average. Exceptions are DNS and to some extent also HTTPS and SMTP protocols, where the percentage of covered packets decreases quickly.

Fig. 7 shows a different view on the simple heavy flow prediction method effectiveness. It shows the average number of packets covered by one heavy marked flow for different values of the decision threshold k. Values shown in the graph rise with the decision threshold to a considerably higher number than the average sizes of the flows from Table 1. For example the average size of flow with more than $k = 20$ packets is over 500 packets, while Table 1

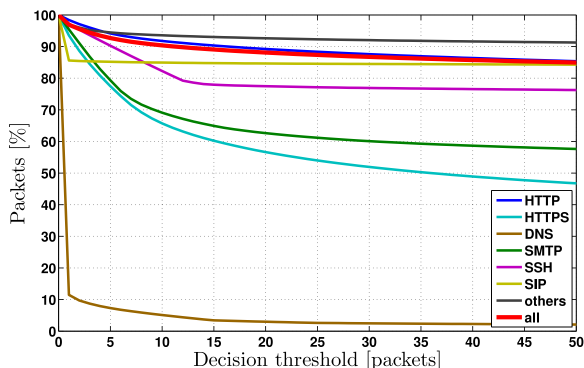


Fig. 6. Heavy flow detection using the simple method—portions of captured packets.

reports overall average of 32.0 packets per flow. This clearly proves that even our simple heavy flow prediction method effectively predicts the heaviest flows. Certainly there are many more advanced methods of heavy flow prediction, but these are out of scope of this paper.

3 System Design

Many 10 Gbps flow measurement systems have adopted a common scheme. A hardware network interface card performs packet capture, sometimes enhanced by packet distribution among several CPU cores. The captured traffic is then sent over the host bus to the memory, where packets are processed by the software applications running at the CPU cores [6]. This model cannot be applied to 100 Gbps networks due to major performance bottlenecks. The main bottleneck lies in limited computational power of CPU which is insufficient for advanced monitoring tasks.

We propose a new acceleration model that overcomes the above-mentioned bottlenecks by a well-designed hardware/software system. The main idea is to give the hardware the ability to handle basic traffic processing. Only the control of the HW and advanced processing of a fraction of the traffic are left for the software. Although the preprocessing is done by the firmware in FPGA, it is fully controlled by the software applications. Therefore, the first few packets of each new flow are sent to the software, which selects a type of hardware preprocessing used for the subsequent packets of that flow. Complete software control of the monitoring process is also the reason why we called the proposed model Software Defined Monitoring (SDM).

The types of data preprocessing in the SDM hardware suitable for the area of network monitoring can be divided into three basic groups:

- *Extraction* of the interesting data from packets and sending only those data to the software in a predefined format, which we call a Unified Header (UH). Then only a few bytes for each packet are transferred from hardware to software, thus reducing the PCIe utilization. Also the CPU has lower load, because the packet parsing is done in the hardware.
- *Aggregation* of packets into flow records directly in the hardware, which brings even higher performance savings for the CPU. This aggregation may range

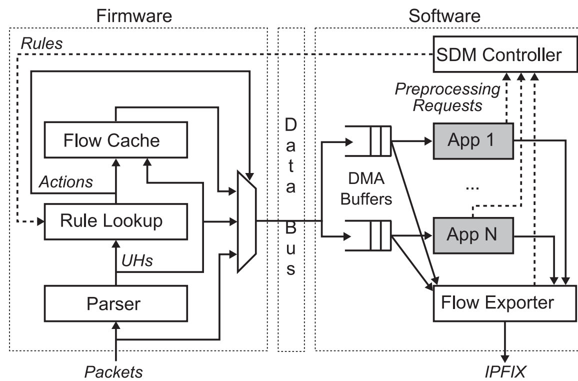


Fig. 8. Conceptual top-level scheme of SDM system.

from basic flow statistics to very specific actions according to the needs of particular applications.

- *Filtration* of unnecessary packets and forwarding only the interesting ones to the software. This can aid advanced monitoring applications, which perform various analyses and detections oriented only to some specific subgroup of network traffic (e.g. DNS threat detector or HTTP header analyzer).

Top-level conceptual scheme of the proposed SDM model is shown in Fig. 8. Forward path is represented by solid arrows and an offload control feedback path by dashed arrows. The system is composed of two main parts (FPGA firmware and software on general CPU) connected together through a data bus. The bus can be PCI Express in case of using commodity PC with a hardware accelerator, or any other interface (e.g. ATCA backplane or internal bus of a single die CPU+FPGA chip).

The processing of all incoming packets starts with parsing a header and extracting packet's metadata (Parser). Extracted metadata is then used to classify the packet based on a software defined set of rules (Rule Lookup). Each rule identifies one concrete flow and specifies the type of packet preprocessing and the target software channel for packets of that flow. Packets can be processed in a firmware flow cache (i.e., aggregated to the selected type of flow record), dropped, trimmed or sent to the software unchanged or in the form of a Unified Header. Flow records residing in the firmware flow cache are periodically exported to the software. The periodic checking is not shown in Fig. 8 for clarity. The data from the firmware is sent over the bus to the software using multiple independent channels. Data for each channel is stored in a software buffer in the form of whole packets, Unified Headers or flow records.

This data is processed by the set of user specific software applications such as the flow exporter [1] which analyzes the received data and exports the flow records to a collector. User applications read the data from the selected channels. They also specify which types of traffic they want to inspect and which flows can be preprocessed in hardware. Definitions of uninteresting traffic from all applications are passed to a software SDM controller daemon. The SDM controller aggregates the definitions (requests) into rules and configures the firmware preprocessing in order to achieve the maximal possible reduction of traffic while preserving the required level of information so that not a single piece of application interesting information is lost. This mechanism realizes the feedback control loop, which is the important concept in our work.

Network traffic preprocessing in the firmware is entirely controlled from the software and the core of the controlling software consists of the monitoring applications (App 1..N). Each monitoring application has the form of a software plugin. The main input to the plugin is the data path carrying the packets, extracted UHs or aggregated flow records. The plugin output is whichever data that the plugin has parsed/detected/measured. This output data is added to the exported IPFIX flow record, so it is *enriched* by the information from the plugin. Each monitoring application also has the interface to the SDM controller.

3.1 SDM Controller

SDM controller accepts the preprocessing requests from multiple applications and aggregates them into rules for the firmware. It also manages timed expiration of application requests and periodical export of aggregated flow records from hardware. The aggregation of preprocessing rules is based on different degrees of data reduction. Ordered from the lowest degree of data reduction the preprocessing types are: none (whole packets), trim (shortened packets), partial (UH), complete (flow record) and elimination (packet drops). Therefore, aggregation of rules in the SDM controller is done simply by the selection of the lowest preprocessing degree (highest data preservation) for particular flows which satisfy the information level requirements of all monitoring applications. In order to maintain a proper functionality of the SDM firmware, the controller must carry out the following operations:

- Management of rules activated in the firmware (rule add/delete/update) based on the application demands.
- Decision about offloading particular flows based on the estimated flow size and the free space in the firmware flow cache.
- Cyclic export of active flow records computed in the firmware flow cache.
- Allocation of records in the firmware flow cache.

In the previous section we have presented the method of heavy flow estimation based on the simple packet count threshold. In the design for practical implementation, we further extend this idea by using *adaptive threshold* that automatically reacts to the changing characteristics of network traffic in time. The adaptation is based on current load of the firmware flow cache, which has a limited size. For the best offload ratio, it is advantageous to keep the flow cache nearly full at all times. That way, there is still some space left for the new flows, while the amount of offloaded traffic is maximized. Therefore, SDM controller periodically checks the flow cache state, decreases the heavy flow decision threshold when the flow cache utilization drops below a specified point, and increases the threshold when the flow cache utilization rises.

3.2 SDM Firmware

Top level implementation scheme of the SDM accelerator firmware for FPGA is shown in Fig. 9. The main firmware

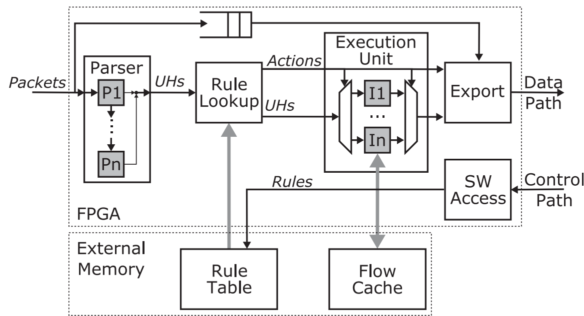


Fig. 9. Detailed firmware scheme.

functionality is realized by a processing pipeline that processes incoming network traffic and creates an outgoing data flow for the software. Packets do not flow directly through the processing pipeline, but are rather stored in a parallel FIFO buffer. The processing pipeline uses only meta-information (UH) extracted from packet headers by Parser. Whole software control of the processing pipeline is realized through SW Access module which conveys the preprocessing rules to be used in Flow Search unit.

The SDM firmware is realized by five main modules:

Parser extracts interesting information from headers of packets, especially fields that clearly identify network flows. To identify the flows, we use the five-tuple: IP addresses, TCP/UDP ports and protocol. Furthermore, our implementation is modular and enables easy extensions of default packet parsing process by additional application-specific parser modules (P1..Pn). This way, the information extracted from each packet can be enhanced when required. Further information about this parser can be found in [9].

Rule Lookup assigns an action (processing instruction) to every packet based on its flow identifier and a set of software defined rules. Management of the rule set is done through a control interface capable of atomic on the fly add, remove and update of the rules.

Execution Unit manages stateful flow records in Flow Cache. It mainly actualizes their values by execution of instructions from flow associated actions. Every action specifies an instruction that should be executed and the address of the flow record to work with. Furthermore, the instruction has access to data extracted from packet (UH). Special type of instruction is an export of the record values, possibly followed by a reset of the record. Records can be exported not only at the flow end but also in a periodical manner, so that the software applications can have actual information about flows in the firmware. Control of memory allocation for records and their periodical export is left to the SDM controller. The Execution Unit supports multiple user-defined instruction sub-modules (I1..In). More details about the execution and implementation of instructions are in Section 3.3.

Export pairs together corresponding UH transaction with frame data from FIFO buffer. Then it chooses the required channel and format for the data based on action assigned by Rule Lookup module.

SW Access is the main configuration access point into the SDM firmware from the software side. Its primary function is to manage the rules and to initiate export of the flow records based on controller commands. Besides, it contains all configuration and control registers.

3.3 Execution Unit Functionality

As already mentioned, Execution Unit realizes the main stateful behavior of the hardware by execution of flow record updating instructions. To improve the overall flexibility of the system, we use modular architecture within the Execution Unit that allows us to implement custom read-modify-write aggregation operations (instructions). Thanks to these custom instructions, the nature of the flow records maintained by the hardware in Flow Cache can be customized according to a target application. Furthermore, we use high-level synthesis (HLS) tools to generate custom hardware modules from an instruction description in C or C++. Thanks to that, SDM hardware functionality can be customized faster and even without the knowledge of HDL programming (e.g. by network security experts). Also an incremental, performance driven design of new hardware accelerated applications is much easier. The process starts with a software implementation of the application, accelerated only by the default SDM instructions. Then the performance bottleneck is identified and the critical piece of code is moved into the FPGA as a new instruction with minimal extra effort.

We have already implemented and evaluated five different Execution Unit instructions to test the feasibility of the described concept with HLS usage:

- *NetFlow* instruction is used for standard NetFlow aggregation. Its execution increases flow packet and byte counters, updates flow end timestamp and computes logical OR of the observed TCP flags.
- *NetFlow Extended* instruction has the same basic functionality as NetFlow. In addition, it stores TCP flags of the first five packets. This additional information may become very useful for analysis of TCP handshake or for detection of network attacks like DoS (Denial of Service).
- *TCP Flag Counters* instruction performs increment of counters of individual observed TCP flags. For example, one can see the number of ACK flags transmitted during the whole TCP connection. Information from this aggregate can be used to support advanced flow analysis [10].
- *Timestamp Diff* instruction maintains records of inter-arrival times of the first 11 packets of the flow. These times have nanosecond precision and can be used as network discriminators for flow-based classification [10] or for identification of application protocol [11].
- *CPD* instruction (Change-Point Detection) shows implementation of more complex operation. CPD is an algorithm designed to detect an anomaly in the processed network flow. Description of this method is out of scope of this paper, more details can be found in [12], [13].

4 Results

We have implemented the whole SDM prototype in order to verify the proposed concept. The hardware part of the prototype is realized on an accelerator board with a powerful Virtex-7 H580T FPGA (Fig. 10). The FPGA firmware realizes

Fig. 10. COMBO-100 G accelerator used for our prototype implementation.

the SDM functionality, such as packet header parsing and NetFlow statistics updating, but also 100 Gbps Ethernet, PCI-Express and QDR external memory interface controllers. The software is realized as a set of plugins for the Invea-Tech's Flowmon exporter software [14]. This exporter allows us to modify its functionality to the extent required by the SDM concept.

We follow by measurement of the real effectivity of the heavy flow detection. Control of the hardware preprocessing is mainly realized by the monitoring applications through on the fly defined dynamic rules for particular flows. These rules are generated as a reaction to the first few packets of the flow. Therefore, there is some delay between the flow start and offload rule application. The duration of this delay influences a portion of packets affected by the rules. The basic view of achievable SDM effectiveness can be gained from an examination of an achievable portion of packets whose preprocessing was influenced by the dynamic flow rules.

We have created a simple use case in order to test the described ability of the SDM concept. In this use case, only a specified number of the first packets from each flow are interesting to the software. All packets from unknown (new) flows are, therefore, forwarded into the software application by default. SDM controller software counts the number of packets in each flow. Right after the reception of the specified number of packets for a flow, the application creates a rule for the firmware to drop all the following packets from this flow. This decision method is absolutely the same as the simple heavy flow detection method defined in the previous section, but the adaptive threshold is not employed in this use case.

We have measured the portion of packets dropped by the SDM firmware in the described test case. The results are projected into the graph in Fig. 11. The graph shows the percentage of dropped (influenced) packets (solid lines) and the percentage of flows for which the rule was created (dashed lines). For comparison, analytical results from graphs 5 and 6 in the previous section are also shown (red). The result is that the SDM can influence preprocessing of

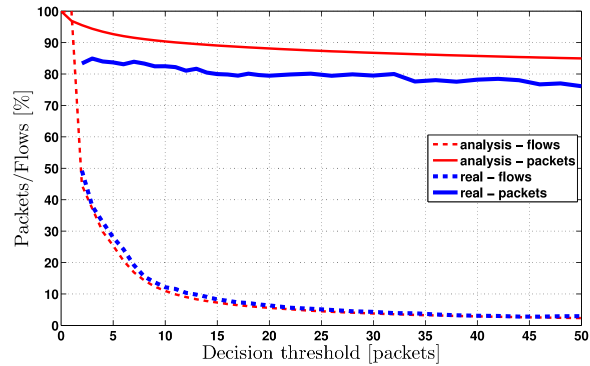


Fig. 11. Portions of offloadable packets and flows using the simple heavy flow detection method.

up to 85 percent of all packets from real network traffic by dynamic flow rules. A visible difference of about 10 percent of influenced packets between analytical and real results is caused by neglecting the duration of rule creation and activation process in the analytical result.

The portions of offloaded packets and flows are similar to the analysis in Section 2—there is a considerably faster decline in the percentage of flows than in the percentage of packets. A different view is provided in Fig. 12. There, a relation of the mean number of packets influenced by one created rule over the decision threshold value is shown (blue). The red line is analytical result of simple heavy flow detection method effectiveness taken from Fig. 7. The graph shows that real measured effectiveness of this method is slightly worse than the analysis suggests, but still suitable for real usage.

We also provide a test of SDM acceleration abilities in more realistic use cases. We test the performance of the concept prototype in the following four cases:

- *Standard NetFlow measurement.* In this use case, all packets from a network line are taken into account. Since NetFlow measurement is based on counting statistics of packet headers only, the packets are sent to the software in the form of UH by default. The software then adds dynamic rules to offload the NetFlow measurement of heavy flows (predicted by our simple method) into the hardware accelerator.

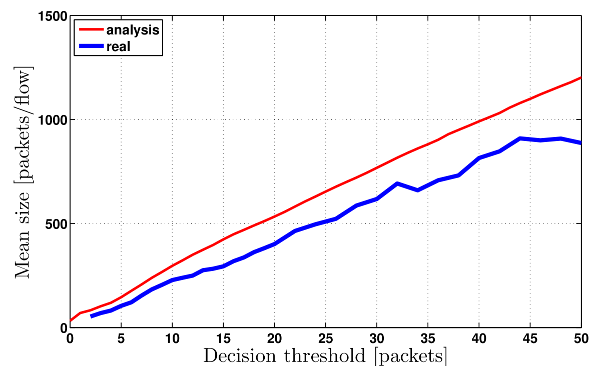


Fig. 12. Mean number of offloadable packets per flow in flows selected using the simple heavy flow detection method.

TABLE 2
Usage of Hardware Preprocessing

Use case	Preprocessing method [% of packets]			
	Packet	Header	NetFlow	Drop
NetFlow	–	20.55	79.45	–
Port scan	–	17.54	–	82.46
Heartbleed	4.91	–	–	95.09
HTTP	22.82	–	–	77.18
HTTP+NetFlow	23.34	10.56	66.10	–

- *Port scan detection.* This use case demonstrates a measurement that is flow-based, yet not directly NetFlow. The software plugin observes UHs of first several packets of each flow and installs drop rules for the subsequent packets of heavy flows. This information is typically enough to detect port scan attacks through various methods.
- *Heartbleed detection*. clearly demonstrates the need for application-layer processing in the network security monitoring. The software application first instructs the accelerator to drop all non-SSL packets (i.e., other than TCP port 443). Then further rules to drop packets of heavy SSL flows are installed in the runtime, because the Heartbleed attack can be detected by observing first few packets of each flow.
- *HTTP header analysis.* From application layer protocols we choose HTTP because our network analysis in Section 2 shows that HTTP traffic is dominant in current networks. Therefore, acceleration of its analysis is of high importance. In this use case we test an application that parses HTTP headers and extracts some interesting information (e.g. URL, host, user-agent) from them. Because the application works with the data of HTTP packets, only the packets with a source or destination port 80 are sent into the software by default. Others are dropped in the hardware. Furthermore, the application adds dynamic rules to drop the packets of HTTP flows in which it already detected and parsed the HTTP header.
- *Standard NetFlow enriched by HTTP analysis.* This case combines two of the previous use cases. Both NetFlow exporter and HTTP parser are active at the same time without the need of any changes in them. Their traffic preprocessing requirements are automatically combined by the SDM controller.

Tables 2 and 3 show the results of the SDM system testing in the described use cases. The tables show portions of all incoming packets and bytes preprocessed in the hardware by each preprocessing method. These hardware preprocessing utilizations lead to a reduction of software application load displayed in Table 4. The table shows portions of incoming packets and bytes that are processed by software applications in each use case relative to the state without the SDM accelerator. It also shows a percentage of flows for which a rule was created in the hardware.

TABLE 3
Usage of Hardware Preprocessing

Use case	Preprocessing method [% of bytes]			
	Packet	Header	NetFlow	Drop
NetFlow	–	12.03	87.97	–
Port scan	–	10.35	–	89.65
Heartbleed	3.77	–	–	96.23
HTTP	27.82	–	–	72.18
HTTP+NetFlow	28.50	3.63	68.87	–

TABLE 4
Software Applications Load Using SDM in Tested Use Cases, Relative to the State without the SDM Accelerator

Use case	SW load [%]		Flows covered by rules [%]
	Packets	Bytes	
NetFlow	20.66	0.98	6.37
Port scan	17.54	0.86	6.53
Heartbleed	4.91	3.77	0.95
HTTP	22.82	27.82	1.98
HTTP+NetFlow	34.02	29.00	6.04

Standard NetFlow measurement is significantly accelerated by the hardware flow cache. In this way, the software application load is reduced to one fifth of all packets (in the form of UH or flow record). Further acceleration rises from the fact that only UHs and flow records are sent to the software, instead of complete packets. The software, therefore, does not parse packets anymore and the PCI Express bus load is reduced to less than one percent.

The unnecessary packets are dynamically dropped in the Port scan scenario. Furthermore, the detector do not require whole packets—UHs are sufficient. This constellation leads to considerable savings of both bus bandwidth and CPU load.

Dropping the packets based on static and dynamic rules is also the preferred method of acceleration in both Heartbleed detection and HTTP protocol analysis scenarios. This leads to the HTTP parser load being reduced to only about a quarter of all packets and bytes and even more significant reduction in the Heartbleed detection. Due to the fact that both static and dynamic rules are used, the percentage of dropped packets is split in two parts. In the HTTP use case 51.84 percent of all packets were dropped by a static TCP port 80 check, and 21.34 percent of packets belonged to heavy TCP port 80 flows for which the dynamic rule has been installed by the SDM controller.

In the standard NetFlow measurement together with the application protocol parsing scenario, the load of the application protocol parser is the same as when used alone thanks to the DMA channel traffic splitting supported by SDM. The HTTP parser software still receives only the packets on the TCP port 80. The load of the software NetFlow measurement slightly rises compared to the NetFlow only measurement, because of the packets that are sent to the software for the HTTP analysis (NetFlow measurement sees also the HTTP packets).

Graphs in Figs. 13, 14, and 15 show results of SDM prototype testing in the NetFlow use case in more details. In the graphs we can see courses of various parameters of SDM system during whole day of NetFlow measurement. Packets preprocessing ability of the accelerator is presented in the first graph. During the whole day, the majority of all

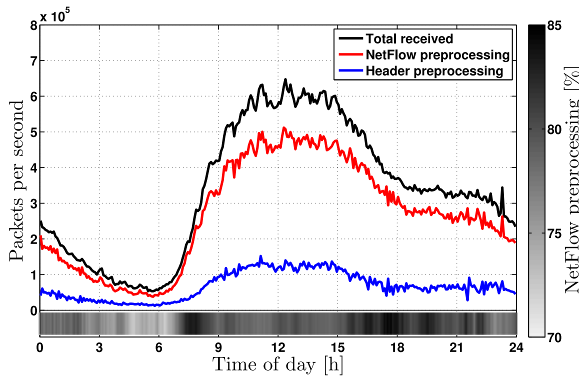


Fig. 13. 24 hours NetFlow measurement with SDM—processed packets.

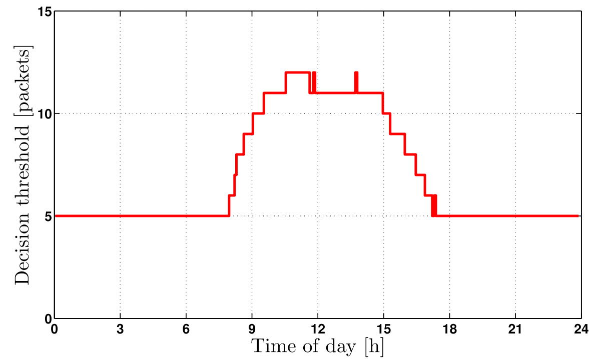


Fig. 15. 24 hours NetFlow measurement with SDM—decision threshold.

received packets (black line) are processed in the firmware flow cache (red line), leaving only a small portion for software processing (blue line). Offloaded percentage of packets is always in the range from 70 to 85 percent of total traffic and is shown in gray shade bar at the bottom of the grid. The second graph shows the number of active rules maintained in the SDM firmware compared to the number of active flows in the network. Black dashed lines demarcate a desired flow cache load maintained by the adaptation of heavy flow decision threshold. There is a significant spike in the total number of flows in the network at around 10:55 pm. After further analysis, we have found that the spike was caused by a mid-sized DoS attack with randomly generated port numbers. Each attacking packet represented a separate flow and was therefore not offloaded to the accelerator. That is a desired behavior, since we want to retain as much information about the attack as possible.

The adaptation of the threshold value during the measurement is illustrated in the third graph. During heavy network load, the threshold value raises to keep the number of offloaded flows within the given range. When the load starts to decline at around 3 pm, the threshold value follows until it reaches a chosen reasonable minimal value (five packets).

For the NetFlow use case, we have also measured a SDM performance curve after system startup in heavy network traffic. The results are depicted in Fig. 16. At the start of the test, the SDM functionality is disabled and all packets are sent for processing into CPU (0 % offload). When SDM is enabled (time 0), we immediately see quick increase in the percentage of offloaded packets as the accelerator is swiftly learning the active heavy flows from the software controller. Around one minute mark, the rise starts to slow down, but still steadily continues for 4 more minutes. After that, the SDM performance is stabilized. Described trend of SDM startup performance curve is very similar also in other tested use cases.

Finally, in Fig. 17 we examine the trade-off in CPU load, since the management of rules in SDM controller represents an additional load to the CPU. We show the effect of SDM acceleration on CPU utilization savings. For this purpose we use the most difficult of our use cases—Netflow measurement together with HTTP analysis. Left half of the graph in Fig. 17 shows measured CPU load with enabled SDM in a stabilized state, right half shows CPU load after SDM was disabled (all processing starts to be done on CPU). According to Table 4, the software load in HTTP +NetFlow use case is up to one third of received packets and bytes when using SDM. This perfectly corresponds to the observed increase in CPU load for packet processing (red line) from 20 to 60 percent after SDM disabling. However, when SDM functionality is enabled, the SDM controller brings some additional overhead (blue line) for configuring the accelerator and aggregating the applications requests. In the end then, total CPU load is two-times lower when using SDM in HTTP+NetFlow use case (black line). This graph also suggests that SDM is best suited for highly

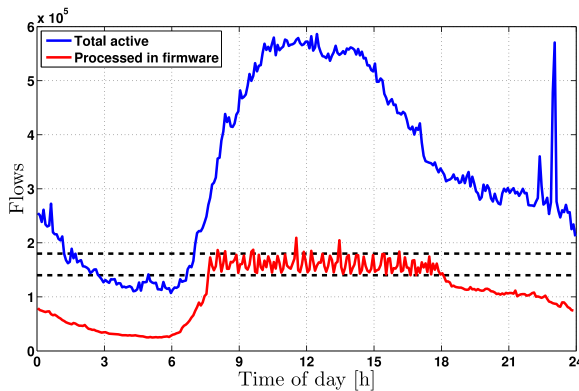


Fig. 14. 24 hours NetFlow measurement with SDM—active rules.

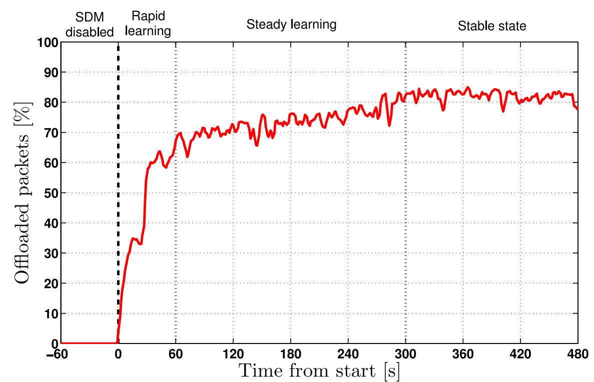


Fig. 16. SDM performance after heavy duty startup.

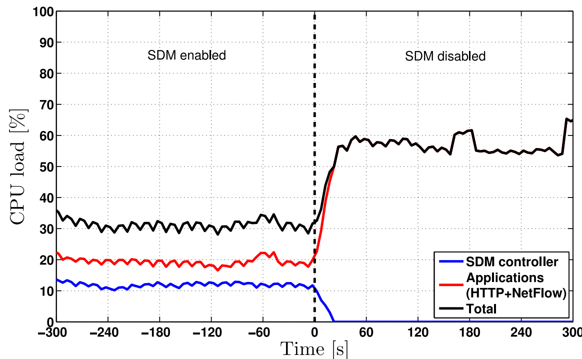


Fig. 17. CPU load in HTTP+NetFlow use case with and without SDM support.

advanced software tasks which consume significant CPU resources. Due to the fact that SDM controller CPU load (blue line) is independent on the application, its share in the total CPU load decreases with the complexity of the application (red line).

4.1 FPGA Implementation Results

Our high-speed SDM FPGA firmware runs at 200 MHz and occupies less than half of the available FPGA (Virtex-7 H580T) resources. Closer look at the FPGA resources of the firmware is shown in Table 5. Using the same SDM core with a data width of 512 bits and throughput of 100 Gbps, we have created three different FPGA architectures for boards with three different arrangements of Ethernet ports: one 100 GbE port, two 40 GbE ports and eight 10 GbE ports.

In addition to the high-performance 100 Gbps solution, we also provide an analysis of the SDM core with narrower data width. These solutions can be used in applications with lower throughput requirements, e.g. in embedded 1 or 10 Gbps probes. Note that the results for data widths other than 512 bits were obtained by simple downscaling of the SDM core. Further optimizations are certainly possible to achieve significantly lower FPGA resource utilization for lower throughputs.

Table 6 shows the resource utilization of the individual instruction sub-modules for the Execution Unit. It can be seen that the additional instruction sub-modules are relatively small, compared to the whole firmware, and therefore adding new instruction should not involve any major refinements of the FPGA firmware. Furthermore, a comparison between high-level synthesis and handmade implementation can be seen from the first two rows of the table. Handmade implementation occupies less than a half of LUTs and a bit less registers compared to HLS result. On the other hand, the creation of C implementation of the instruction and its subsequent automatic synthesis to HDL is much faster and simpler than HDL implementation.

TABLE 5
Resources of the SDM Firmware

Firmware/Module		Regs	LUTs	Throughput
Complete SDM		197,758	249,214	1 × 100 Gbps
		134,172	178,984	2 × 40 Gbps
		184,084	222,745	8 × 10 Gbps
SDM core	512 b	30,497	51,333	100 Gbps
	256 b	25,866	42,793	50 Gbps
	128 b	23,534	39,006	25 Gbps
	64 b	22,384	37,233	12.5 Gbps
	32 b	21,908	36,803	6.25 Gbps
Virtex-7 H580T FPGA		725,600	362,800	

TABLE 6
Resources of the Instruction Blocks

Instruction	Regs	LUTs
NetFlow (handmade VHDL)	1,754	325
NetFlow	1,846	824
NetFlow Extended	2,070	1,113
TCP Flag Counters	0	1,046
Timestamp Diff	5,199	2,556
Change-Point Detection	5,296	3,919

5 Related Work

We discuss several approaches that may to some extent resemble the SDM concept. However, we show that our work has significant differences to those works.

Snort [15] is an open source software network intrusion prevention and detection system. It relies heavily on regular expression matching, while our work does not enforce nor assume any particular type of software processing. While many papers dealing with hardware acceleration of Snort have been published, they typically restrict their focus to regular expression matching only. We argue that network security monitoring is much more complex task than that and the limitation to RE matching makes those systems unfeasible for practical use. L7-filter [16] is a Linux packet classifier software aiming at protocol identification. It resembles Snort as it also relies on regular expressions.

A good example of a complex software library for application layer traffic processing (showing that RE matching is not sufficient) is nDPI [17]. While this open source library is probably too complex to be hardware accelerated, we envision that similar software can be used as a basis of a SDM plugin.

The OpenSketch architecture [18] defines a configurable pipeline of hashing, classification and counting stages. These stages can be configured to perform the computation of various statistics. OpenSketch is tailored to compute *sketches*—probabilistic structures allowing to measure and detect various aspects of the network communication with a defined error rate. It is not intended for complete NetFlow-like monitoring, nor for exact, error-free measurements. Also, OpenSketch does not allow for application level protocol parsing.

FlowContext system [19] provides a flexible way to implement stateful network traffic processing in an FPGA. NetFlow monitoring is among the examples of its use. However, it does not provide tight control feedback loop to a software application, and therefore cannot be effectively used for problems exceeding the capabilities of a single FPGA.

There have been efforts to implement NetFlow traffic monitoring in FPGAs, most recently even as an open source project [20] for the NetFPGA platform. Our work is however more flexible by allowing application protocol processing in

the software and further acceleration through extensions of the Execution Unit.

The Shunt system [21] is a hardware accelerator with support to divert a suspicious/interesting traffic to a software for further analysis. To this end it resembles our work, however, Shunt accelerates only packet forwarding and does not include any possibilities to offload/accelerate the flow measurement tasks. Our work is also more complete by defining the software architecture with the plugin support.

Xilinx has recently announced SDNet [22] environment for software defined, hardware accelerated networking. The system uses high level language(s) to describe a network application, which is then compiled to a form of hardware accelerator for a Xilinx FPGA. From the limited information available at the time of writing, we envision that SDNet could be used to improve SDM by custom application parsers or instruction modules.

The proposed arrangement of SDM resembles OpenFlow [23]: Packets of an unknown flow are passed from a data path to a control software, which in turn may choose to install processing rules into the data path. Similar to plugins for an OpenFlow controller, SDM is also designed to support various software plugins. In addition to that, newer versions of OpenFlow standard define monitoring primitives for the data path. The main difference with OpenFlow is that, for the sake of performance, our system is not distributed, but our controller is rather very tightly coupled with the hardware accelerator—within the same box, or even at the same chip. That allows implementing applications which would be impractical when built as a distributed system. We also propose user-defined modifications to the data plane through the modular Execution Unit—a concept that is unparalleled in OpenFlow. Our system is an instance of Software Defined Networking in a broader sense, yet it is different from OpenFlow.

6 Conclusion

We propose a new concept of application level flow monitoring acceleration called Software Defined Monitoring. The concept is able to support application level monitoring and high-speed flow measurements at speeds over 100 Gbps at the same time. Our system focuses on a high speed and high quality flow based measurement with the support of a hardware accelerator. The accelerator is fully controlled by the software feedback loop and offloads the simple monitoring tasks of bulk, uninteresting traffic. The software, on the other hand, decides about the traffic processing on a per-flow basis and performs the advanced monitoring tasks such as application protocol parsing. The software works with monitoring plugins, therefore, SDM is *by design* ready for extensions by new high-speed monitoring tasks without the need to modify its hardware. Moreover, the FPGA accelerator itself can also be improved to support new types of offload.

Our detailed analysis of the backbone network traffic parameters demonstrates the feasibility of the concept. We have also implemented the whole SDM system using the Virtex-7 FPGA accelerator board, including some extensions to the firmware offload engine. The system is ready to handle 100 Gbps traffic. Using the SDM prototype, we have evaluated several use cases for SDM. It is clear from the obtained results that SDM is able to offload a significant part of network traffic to the hardware accelerator and therefore to support a much higher throughput than a pure software solution. The results show a major speed-up in all test cases.

Acknowledgments

This research has been partially supported by the "CESNET Large Infrastructure" project no. LM2010005 funded by the Ministry of Education, Youth and Sports of the Czech Republic, the research programme MSM 0021630528, the grant BUT FIT-S-14-2297 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070. V. Puš is the corresponding author.

References

[1] B. Claise, "Cisco systems netflow services export version 9," RFC 3954, Internet Engineering Task Force, Oct. 2004.
[2] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information," RFC 7011, Internet Engineering Task Force, Sep. 2013.
[3] L. Deri, L. Trombacchi, M. Martinelli, and D. Vannozzi, "A distributed dns traffic monitoring system," in *Proc. 8th Int. Wireless Commun. Mobile Comput. Conf.*, 2012, pp. 30–35.
[4] M. Elich, P. Velan, T. Jirsik, and P. Celeda, "An investigation into teredo and 6to4 transition mechanisms: Traffic analysis," in *Proc. 38th Conf. Local Comput. Netw. Workshops*, 2013, pp. 1018–1024.
[5] P. Velan, T. Jirsik, and P.Čeleda, "Design and evaluation of http protocol parsers for ipfix measurement," in *Proc. Adv. Commun. Netw.*, 2013, vol. 8115, pp. 136–147.
[6] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 218–224.
[7] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, Aug. 2003.
[8] K.-C. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Comput. Netw.*, vol. 50, no. 1, pp. 46–62, Jan. 2006.
[9] V. Puš, L. Kekely, and J. Kořenek, "Low-latency modular packet header parser for FPGA," in *Proc. 8th ACM/IEEE Symp. Arch. Netw. Commun. Syst.*, 2012, pp. 77–78.
[10] A. W. Moore, D. Zuev, and M. L. Crogan, "Discriminators for use in flow-based classification," Department of Computer Science, Queen Mary University of London, London, U.K., Tech. Rep. RR-05-13, 2005.
[11] P. Piskac and J. Novotny, "Using of time characteristics in data flow for traffic classification," in *Managing Dynamics Netw. Serv.*, 2011, vol. 6734, pp. 173–176.
[12] A. Tartakovsky, A. Polunchenko, and G. Sokolov, "Efficient computer network anomaly detection by changepoint detection methods," *Sel. Topics Signal Process.*, vol. 7, no. 1, pp. 4–11, 2013.
[13] R. B. Blazek, H. Kim, B. Rozovskii, and A. Tartakovsky, "A novel approach to detection of "denial of service" attacks via adaptive sequential and batch-sequential change-point detection methods," in *Proc. 2nd IEEE Workshop Syst., Man, Cybernetics*, 2001, pp. 220–226.
[14] INVEA-TECH a.s.. (2014). FlowMon Exporter-Community Program. [Online]. Available: http://www.invea.cz
[15] Snort. (2014). [Online]. Available: http://www.snort.org
[16] l7-filter. (2014). [Online]. Available: http://l7-filter.clearfoundation.com/
[17] ntop, nDPI. (2014). [Online]. Available: http://www.ntop.org/products/ndpi/
[18] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. 10th USENIX Conf. Networked Syst. Des. Implementation*, 2013, pp. 29–42.
[19] M. Košek and J. Kořenek, "Flowcontext: Flexible platform for multigigabit stateful packet processing," in *Proc. Int. Conf. Field Programm. Logic Appl.*, 2007, pp. 804–807.

[20] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and flexible flow-based monitoring for high-speed networks," in *Proc. 23rd Int. Conf. Field Programm. Logic Appl.*, Sep. 2013, pp. 1–4.
[21] N. Weaver, V. Paxson, and J. M. Gonzalez, "The shunt: An FPGA-based accelerator for network intrusion prevention," in *Proc. 15th Int. Symp. Field Programm. Gate Arrays*, 2007, pp. 199–206.
[22] Xilinx Inc., SDNet. (2014). [Online]. Available: http://www.xilinx.com/applications/wired-communications/sdnet.html
[23] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

Lukáš Kekely is working towards the PhD degree at the Faculty of Information Technology, Brno University of Technology since 2013 and also a researcher at CESNET since 2011. His research is focused mainly on hardware accelerated solutions for high-speed networks, particularly in the area of monitoring and security. He is an author of several research papers published at renowned international conferences.

Viktor Puš received the PhD degree from the Faculty of Information Technology, Brno University of Technology in 2012. He is a researcher with focus on hardware acceleration of timing-critical operations in the network, particularly in the network security monitoring. He is an author of one US patent and many research papers published at renowned international conferences.

Jan Kořenek received the PhD degree in 2010 from the Brno University of Technology, Czech Republic. He has substantial experiences in the hardware acceleration of network applications which was obtained by working on a number of European and locally funded projects. He is an author of many papers and novel hardware architectures. In May 2007, he co-founded INVEA-TECH which is a successful spin-off company focused on high speed network monitoring and security applications. In 2009, he formed Accelerated Network Technologies (ANT) research group at Brno University of Technology.

Jan Kučera graduated with a bachelor's degree at the Faculty of Information Technology, Brno University of Technology in 2014. He continues his studies in the follow-up master's degree programme. Since 2012, he works as a researcher at CESNET and is interested in FPGA design, especially in hardware acceleration for the purpose of high-speed networks monitoring.

Athanasios V. Vasilakos served or is serving as an editor or/and guest editor for many technical journals,such as the *IEEE Transactions on Network and Service Management*; *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Cybernetics*; *IEEE Transactions on Nanobioscience*; *IEEE Transactions on Information Technology in Biomedicine*; *ACM Transactions on Autonomous and Adaptive Systems*; the *IEEE Journal on Selected Areas in Communications*. He is also general chair of the European Alliances for Innovation (www.eai.eu).

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

A.2 Paper II

General IDS Acceleration for High-Speed Networks

Jan KUČERA, Lukáš KEKELY, Adam PIECEK and Jan KOŘENEK. "General IDS Acceleration for High-Speed Networks". In: *Proceedings of the 36th IEEE International Conference on Computer Design.* ICCD 2018. Orlando, FL, USA: IEEE, 2018, pp. 366–373. ISBN: 978-1-5386-8477-1.

2018 IEEE 36th International Conference on Computer Design

General IDS Acceleration for High-Speed Networks

Jan Kučera, Lukáš Kekely
CESNET, a. l. e.
Prague, Czech Republic
jan.kucera@cesnet.cz
kekely@cesnet.cz

Adam Piecek
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
xpiece00@stud.fit.vutbr.cz

Jan Kořenek
IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
korenek@fit.vutbr.cz

***Abstract*—Network Intrusion Detection Systems have gained popularity as one of the key technologies to secure communication infrastructures. However, their high computational complexity poses performance challenges for practical deployment in modern high-speed networks. To achieve the highest quality of detection, IDS should process as much relevant data as it can without becoming the bottleneck of a network connection. At the same time, IDS implementation should be flexible enough to accommodate detection methods of ever emerging new security threats.**

This paper aims at an acceleration of IDS by means of informed packet discarding, effectively focusing the available resources of overloaded IDS to the most relevant parts of analyzed traffic. Unlike previous works, the proposed scheme does not move the IDS nor any specific portion of it into the hardware accelerator. Rather it uses smart software based or hardware accelerated offload (bypass) of the traffic parts that are not likely to represent a security threat. The flexible nature of software-based IDS is therefore fully maintained, while the quality of threat detection remains sufficiently high even when processing high-speed traffic. We show that controlled (informed) discarding of well-defined portions of input traffic yields better detection rates, compared to the default uncontrolled (blind) buffer overflow discarding in high throughput scenarios. Our results show that it is entirely possible to run an IDS on a high-speed network link using single CPU with an FPGA accelerated packet pre-filtering.

I. Introduction

Intrusion Detection Systems significantly contribute to network security by providing a deeper insight into transferred packets and their payloads. These systems often use some form of deep packet inspection, such as pattern matching or other methods, to detect characteristic signatures of malicious activity present in the network data. A common property of these inspection methods is their overwhelming computational complexity, leading to challenges in meeting the performance requirements of modern high-speed networks. Running a practical pattern matching algorithm at 100 Gbps or even tens of Gbps is an unreachable goal for current CPUs.

On the other hand, the ever-changing nature of security landscape requires the IDS to be able to react quite rapidly to newly emerging threats and attack vectors. The flexibility of software implementation is therefore highly desirable and the use of considerably less flexible hardware processing offload that accelerates only a specific IDS application directly is therefore not so feasible.

To aid the performance of a general software-based IDS without hindering its flexibility, we propose and explore a different approach to IDS acceleration. The key idea of our concept is not to directly accelerate the speed of data processing in the IDS, but rather to intelligently reduce the amount of input traffic that the IDS must process. Based on a few basic characteristics some packets are deemed interesting and selected for processing, while others are discarded. Also, this selection of packets for processing/discarding is done in such a controlled way so that negative impacts on IDS detection abilities are minimized. We expect and experimentally prove that only a considerably small percentage of all threats on the network is overlooked this way, while the IDS can operate at much higher speeds as originally possible.

We present an IDS acceleration based on these assumptions:

1) The packet rate performance of software-based IDS is limited and insufficient. The IDS is not fast enough to sufficiently process all of the packets that are transmitted in a monitored high-speed network, such as 40 Gbps or 100 Gbps Ethernet line.
2) The default packet discarding mechanism is a blind input buffer overflow that behaves in an effectively random manner. In an overloaded IDS, incoming packets are discarded right after they arrive without any chance of further examination. As a result, a number of threats are overlooked and remain unreported.
3) The most relevant information regarding security is present in several packets at the beginning of each network connection (flow). These packets should be, therefore, preferred for processing over others, when IDS becomes overloaded.
4) Basic packet examination can be performed and the obtained information subsequently used to selectively discard some of the following packets. Furthermore, such packet discarding mechanism exists, that lets the IDS yield a better quality of detection than the default (blind) discarding. This holds, even though the raw packet rate of the IDS itself stays unchanged.
5) Majority of the network traffic is carried by a quite small number of relatively large flows. So, selection of only a few flows for discarding (bypass) can significantly reduce input packet rate of IDS.

This research has been supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 by the MEYS of the Czech Republic; the IT4Innovations excellence in science project IT4I XS – LQ1602; and by the Ministry of the Interior of the Czech Republic project VI20172020064.

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00062

In the following text, we basically assume that points (1) and (2) hold true based on our previous experiences with IDS deployment (more in section III). The validity of point (5) has been already shown in several other papers, so we simply check if this feature is also present in our network data. The main focus of this paper is placed on thoroughly exploring and proving points (3) and (4).

The main contribution of this paper is three-fold:

- Design of an input acceleration concept (heuristic) that considerably reduces the amount of data sent to any general IDS in a controlled and beneficial manner. The concept idea enables that both software and hardware-based implementations are possible.
- Examination of real network traffic traces to show that the overall quality of threat detection remains sufficiently high even when only short flows and several initial packets of large flows are observed. In other words, proving that only a minute fraction of network threats is present in the latter parts of the connection contents.
- Implementation and experimental evaluation of the proposed informed discarding concept to demonstrate its effectivity under real network deployment conditions.

II. Related Work

There are several commonly used software implementations of IDS, which we list here with a brief description. Snort [1] is an open source software network intrusion detection and prevention system. It relies heavily on regular expression matching. Similarly to Snort, L7-filter [2] also operates with regular expressions. It is a Linux based packet classification software aiming primarily at application layer (L7) processing. A software library for application layer traffic processing called nDPI [3] can serve as an example showing that regular expression matching alone is not enough. It should be only one component of a more complex set to form a robust IDS. Bro [4] is a flexible framework that allows specification of custom detection rules using its own scripting language. This feature makes it very powerful, but also rather complex. Suricata [5] is functionally quite similar to Snort, but supports multithreaded processing and is, overall, built to achieve higher performance.

Apart from software implementations, there is a large number of papers proposing partial or full IDS functionality offload to a hardware accelerator, for example [6], [7], [8]. The most common way of doing that is by converting regular expressions to an FPGA firmware structure, thus offloading the time-consuming pattern matching from the CPU. Such approach poses a disadvantage of lowered flexibility. Most proposed methods require recompiling of the FPGA firmware when the regular expression set changes. That may take hours to complete and meeting FPGA timing constraints is never guaranteed. Also, advanced techniques like TCP stream reassembling are often missing in FPGA-based IDS accelerators, leaving open back doors for covert attacks. Finally, for IDS that use more complex threat detection methods than just pattern matching, such acceleration is of little use.

An example of a more flexible acceleration is provided by The Shunt system [9]. It is a hardware accelerator that cab divert a suspicious (interesting) traffic to the software for further analysis. To this end, it somehow resembles our work. Of course, our work uses a more powerful accelerator, resulting in throughput of tens of Gbps, while the Shunt was demonstrated at only 1 Gbps links. The main differentiation of our work is that it is much more complete. While the Shunt paper describes almost exclusively the hardware architecture and its implementation, we primarily aim to provide analysis of real network traces together with extensive experimental results of achieved IDS acceleration. Moreover, we provide results showing the benefits of a pure software implementation of our proposed controlled packet discarding method.

Another more sophisticated hardware acceleration concept is our previous work – Software Defined Monitoring (SDM) system [10]. It is a hardware accelerator aimed primarily at flow-based network monitoring. It supports offloading of NetFlow statistics computation of uninteresting and large flows into the hardware, while sending packets of short, interesting and unknown flows to the CPU for a more detailed analysis. SDM also includes a software controller with easy to use API, which accepts offload requests from the monitoring (security) applications and commands the hardware accelerator accordingly. It has been shown to effectively accelerate network flow measurement up to application layer processing, but no clear benefits have been demonstrated when used to offload traffic from more computationally complex IDS.

A research published in [11] proposes a Time Machine concept. This concept exploits the heavy-tailed nature of network traffic and also assumes that the most relevant information is present at the beginning of each network flow. But, their approach aims at a packet capture system which enables storage of suspicious traffic for later offline (i.e. retrospective) forensics. Also, the Time Machine system is only controlled by an IDS and do not in any way accelerates its operation. That is distinctly different from our proposed concept. We deal with an online analysis of suspicious traffic and employ an accelerated pre-filter to directly aid the IDS performance by reducing the amount of data on its input.

An attempt at creating a high-speed IDS was performed at Berkeley Lab [12] using Bro. In this approach, the traffic is distributed to multiple IDS servers by a pair of switches. To achieve high throughput, five servers with 10 Gbps input lines each are running Bro in parallel. If some Bro instance detects a bulk transfer, the switches are set to discard the subsequent packets of that flow. Therefore, the amount of input traffic to Bro servers is reduced. Our work is similar to [12] in that it uses software IDS and a controlled packet discarding. However, our work aims at achieving similar IDS performance in a single box deployment. Furthermore, our paper also focuses on a detailed analysis of the controlled packet discarding influence on the achieved quality of detection, which is completely missing in [12]. Finally, our discarding algorithm is much finer and IDS agnostic as it does not rely solely on the information from the IDS.

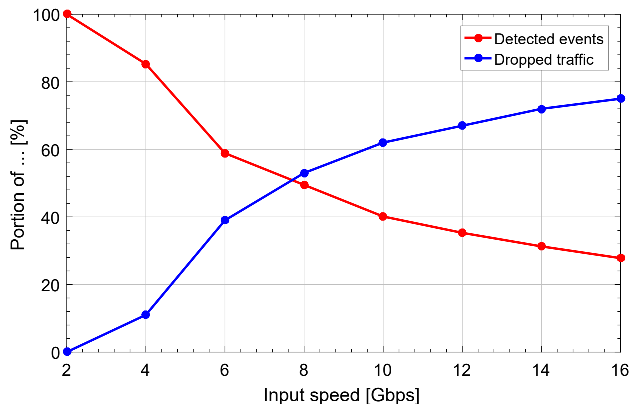


Fig. 1. Percentage of dropped packets and detected events at different speeds.

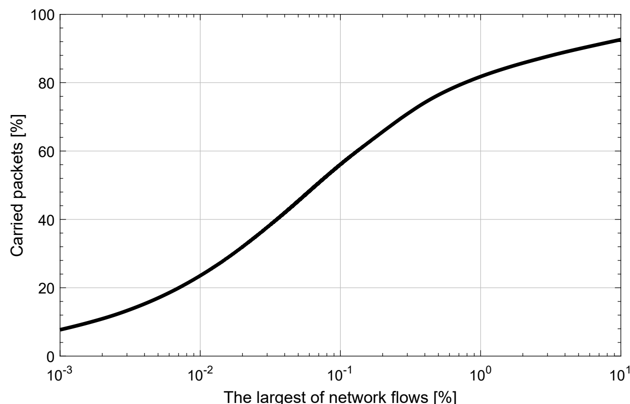


Fig. 2. Percentage of packets carried by the largest flows on the network.

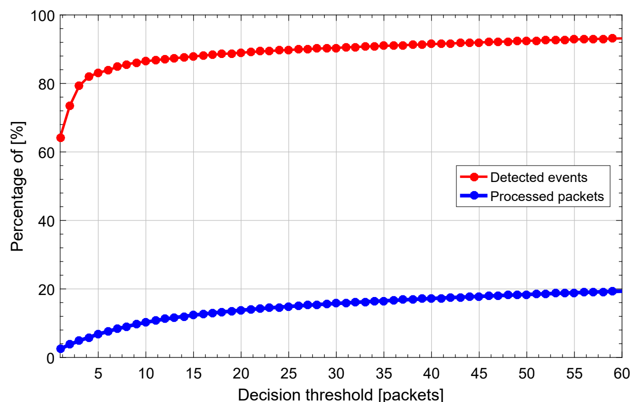


Fig. 3. Processed packets and detected events in them for different N.

III. Proof of Concept

In this section, we analytically support our claim that *controlled* packet discarding results in sufficiently high IDS detection quality, while IDS can handle traffic at much higher speeds. For this evaluation, we use unsampled packet data from one of the lines in our nation-wide network. The captured PCAP file contains 285 833 947 packets of 8 642 744 flows in over 200 GB of data. It was captured at around 2 Gbps link utilization over a duration of 826 seconds. The traffic is replayed on its original capture speed. To perform measurements with higher bandwidths, we still replay the PCAP at the original speed (to maintain flow timing characteristics), but replicate each packet several times. To avoid changing flow data, we deterministically modify each packet's IP address when replicating, which effectively results in new flows being created. As an IDS, we choose Suricata [5] because of its affinity to high-performance multi-threaded implementation. We use 13 642 detection rules from the public EmergingThreats database [13].

To support our claims of insufficient IDS performance or detection quality for high-speed deployment, we provide some basic results in Fig. 1. An out-of-the-box version of Suricata is used. A packet drop rate on its input (blue) and detection accuracy (red) are measured for different traffic speeds. We can see that a packet loss of around 10 % is present even at 4 Gbps. The loss is consistently rising with input speed and at 10 Gbps it already reaches more than 60 %. For the overall detection quality of tested IDS at various input speeds and related drop rates, the 100 % baseline is given by offline (non-discarding) analysis of the acquired network data. All of the expected events are reliably detected only at the speed of 2 Gbps. The presence and increase of *uncontrolled* packet discarding for higher speeds causes the percentage of successfully detected events to drops rapidly. Even at 6 Gbps (40 % drop) only around 60 % of all events are detected and at the speeds of 10 Gbps and more (above 60 % drop) only fewer than 40 % of events is detected. Therefore, there is definitely a reason to utilize some kind of IDS acceleration.

Findings already presented in various papers like [10] suggest that high-speed network traffic has a heavy-tailed character of flow (communication) size distribution. The heavy-tailed character of flow size distribution derived from the captured PCAP file is shown in Fig. 2. The graph shows the percentage of all packets carried by the specific portion of the largest (heaviest) flows from the file. It can be seen that even 0.1 % of the largest flows carry as many as nearly 60 % of all packets and 1 % carry more than 80 %. The observed heavy-tailed character of flow sizes has a potentially positive consequence for an achievable efficiency of the proposed controlled discarding concept. Even if only a small percentage of all flows is selected for controlled discarding, processing of majority of the packets by IDS is still avoided. In other words, this enables to focus the IDS's effort primarily to short flows and several initial packets of larger flows with potential for considerable benefits in achievable performance.

Fig. 3 shows (in blue) the percentage of packets that have to be processed by IDS when only the first N packets of each network flow are analyzed and the rest is discarded (flows shorter than N are analyzed entirely). The discarding decision threshold N is shown on the horizontal axis, while the percentage of packets is drawn in blue, one dot for each considered value of N. We can see that the first 50 packets of all flows carry less than 20 % of all packets, therefore the remaining 80 % of packets are found in later packets of flows larger than 50 packets. These results only confirm the expected implications of the heavy-tailed character of flow size

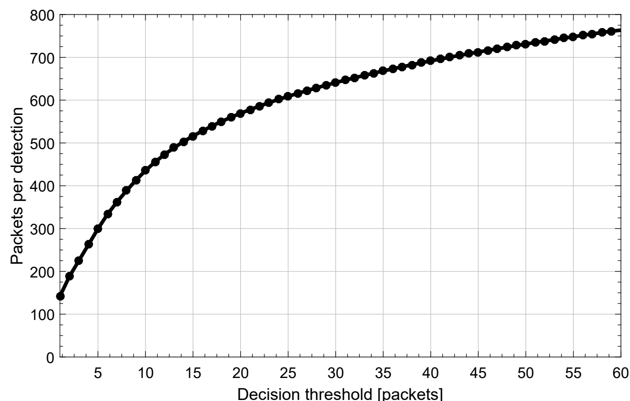


Fig. 4. A number of processed packets per detected event for different N.

distribution stated in the previous paragraph.

Fig. 3 also shows (in red) the number of events detected by Suricata when the processing of the input file is reduced to N initial packets of each flow. The results are shown in relative form (as a percentage), where the base value (100 %) was obtained similarly as before – by running Suricata offline analysis on the original 200 GB file (i. e. $N = \infty$). We can see that more than 80 % of all security threats are detected even when only the initial 5 packets of all flows are considered and more than 90 % when $N > 30$. Furthermore, with forwarding of more packets into the IDS (a further increase of N) the detection rate increases only rather slowly. Therefore, we argue that if the IDS system is able to process only part of the network traffic due to insufficient performance, it should focus primarily on the processing of the first several packets of each flow and the rest should be preferred for discarding.

A different view of the same data is provided in Fig. 4. The number of processed packets is divided by the number of detections for each analyzed value of N. We can see that by lowering the threshold N, considerably fewer and fewer packets must be, on average, analyzed to detect a security threat. Just for comparison, when the whole PCAP file is processed by Suricata (i. e. $N = \infty$) the value of packets per detection ratio rises to over 2000. These findings further prove the conclusion of the previous paragraph, that the IDS can be tuned to higher detection efficiency under heavy loads by the controlled preference of initial packets of flows for analysis.

IV. System Design

As already mentioned in the Introduction, the design of our IDS acceleration concept is mainly motivated by the need to reduce input packet rate to the IDS, while retaining as much relevant information as possible to maintain high detection accuracy. Since we assume that the most relevant information regarding security is present in several packets at the beginning of a network connection, **we design our concept to drop all packets that follow the first N packets of each network flow.** The value of threshold N can be arbitrarily altered depending on momentary network situation. Of course, because of this design choice, the system does not detect attacks which arise after the first N packets of the flow, but as the analysis results from the previous section have shown, the number of such attacks is very small (less than 10 % even for $N = 30$). Furthermore, the value of N should be set to ∞ when IDS is processing input data at sufficient rate and gradually lowered only to prevent blind buffer overflow as input traffic volume starts to overwhelm the IDS.

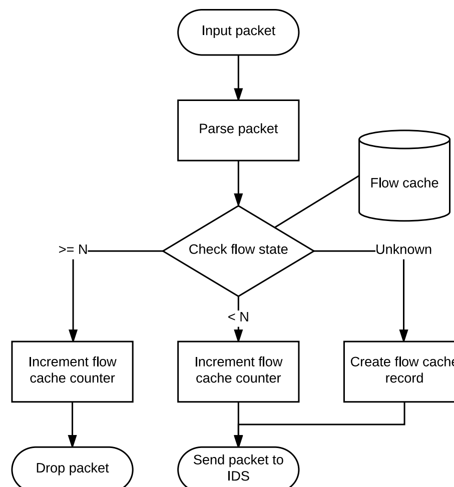


Fig. 5. Flowchart of packet processing in the proposed IDS acceleration.

To perform the described decision, a system implementing the proposed concept must maintain very basic flow statistics. This requires two main additional modules: packet header parser and flow cache. Packet header parser is required to obtain packet header fields that uniquely identify a network flow. We use the standard five-tuple of IP addresses, port numbers, and L4 protocol number to identify flows. Network flow cache is necessary to store and update records of actual flow lengths. Every incoming packet either creates a new flow record or increments a size counter in an existing one.

A flowchart of the proposed IDS input system operation is shown in Fig. 5. Every input packet is firstly parsed and flow identification fields are extracted. Then the associated flow record is searched in the flow cache. A new flow record is created for unknown (not found) flows, packet counter is incremented for known flows. The packet is dropped if the counter for appropriate flow exceeds configured threshold N otherwise, the packet is forwarded to IDS for normal processing. Furthermore, there is an independent housekeeping process (not shown in the flowchart) that removes old entries from the flow cache. It operates with configurable inactive and active timeout periods of flow records.

This scheme of operation is general and independent on the particular features of IDS used. It is therefore applicable to any software IDS for which it is possible to alter the input path. Since most software IDS employ extensible modular design, their input modules or plugins are often a convenient place for the implementation of our acceleration method.

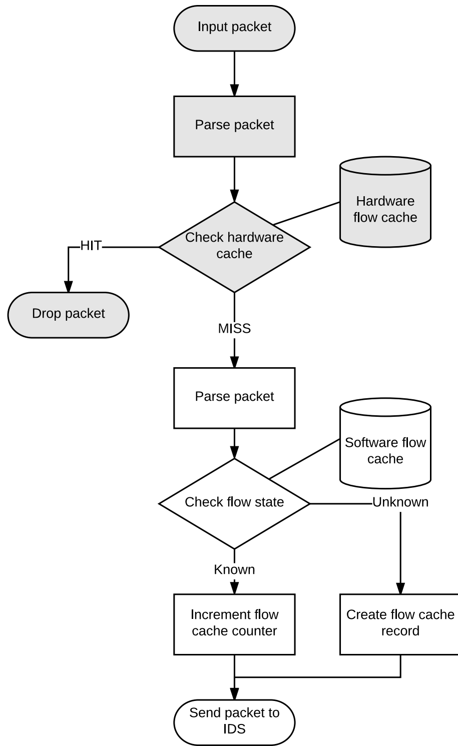


Fig. 6. Packet processing in the proposed IDS acceleration with HW offload.

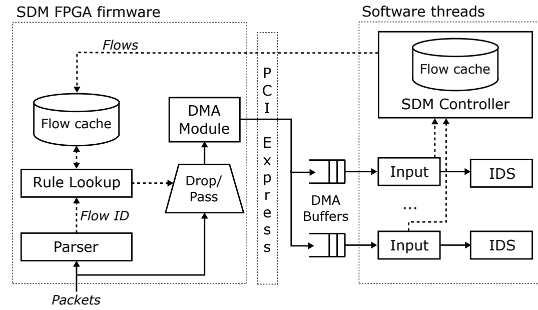


Fig. 7. Our hardware accelerated IDS input concept utilizing subset of SDM.

A. Hardware Accelerated Offload

Utilization of the proposed input system inevitably requires some additional processing from the CPU, adding to its total load. To further explore the performance limits of our approach and achieve even better results, we employ hardware accelerator that drops unwanted packets before they reach the CPU. To enable this functionality, both the packet parser and the network flow cache must be present in the hardware accelerator, so that the decision whether to drop or pass packets can be offloaded and made directly by the accelerator.

The updated scheme of system operation with the utilization of hardware acceleration is shown in Fig. 6. The hardware accelerator (grey blocks) parses packets, searches for appropriate flow records in its flow cache and drops all matching packets. Then the software path (white blocks) only passes packets to IDS and maintains flow records in its software flow cache. The housekeeping process (not shown) replicates (offloads) the heavy flow records that exceed configured threshold N from the software flow cache to the hardware one and also removes outdated flow records from the software and hardware cache according to the configured timeouts.

While it would be certainly possible to *move* the packet parser and the flow cache to the hardware entirely, we rather *replicate* them. The main reason for this decision is to maintain the flexibility of utilization. By maintaining a software flow cache, alternative packet discarding mechanisms can be easily used and fine-tuned. By choosing which flows will be offloaded from the software flow cache to the hardware one, the system can, for example, use different discarding thresholds N for individual traffic types or IP subnets. Also, the packet parser is typically present in IDS anyway, so that it can be shared and presents no additional computation load for CPU. Finally, due to the fact that most of the traffic is discarded in the hardware, the performance penalty of maintaining a software flow cache is significantly reduced.

To implement the proposed hardware accelerated version of our concept, we utilize the Software Defined Monitoring (SDM) system [10]. Although SDM is primarily designed to accelerate flow-based network monitoring, a subset of its functionality can be easily utilized also for our IDS acceleration concept. This utilization is outlined in Fig. 7. SDM uses FPGA accelerator cards with 10, 40 or 100 Gbps Ethernet interfaces connected to a host server via PCI Express bus. The SDM FPGA firmware itself (on the left) already implements capture of input packets from Ethernet links, hardware packet parser, a cuckoo hashing based form of flow cache with packet filtering capabilities, as well as fast DMA transfers with the support for flow-based traffic distribution among CPU cores, enabling effortless multi-threaded IDS operation. The hardware flow cache is realized using on-card QDR memories and has a total capacity of over 250 000 rules (flows). Because network traffic has a heavy-tailed character of flow sizes and we want to offload only the heaviest of the flows, the cache capacity should not be a very limiting factor. However, if it proves to be an obstacle, on-chip URAMs can be used to further enlarge the cache in newer UltraScale+ FPGAs. The SDM software (on the right) includes mainly the SDM Controller, which implements software flow cache and presents a simple C language API for easy integration into existing IDS. As depicted, only input modules/plugins of accelerated IDS are communicating with software flow cache of SDM Controller maintaining its records and requesting offload of selected flows into hardware cache. The offloading process is optimized for latency and throughput, the whole hardware cache can be filled with new records in less than a second. The capacity of the software flow cache is considerably larger compared to the hardware one and it can store tens of millions of flow records.

V. Experimental Results

Evaluation of the proposed acceleration concept presented in this paper uses Suricata IDS [5] running on a commodity

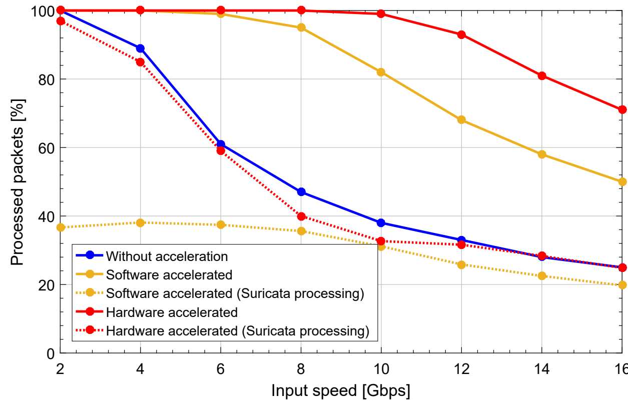


Fig. 8. Percentage of incoming packets processed for different input speeds.

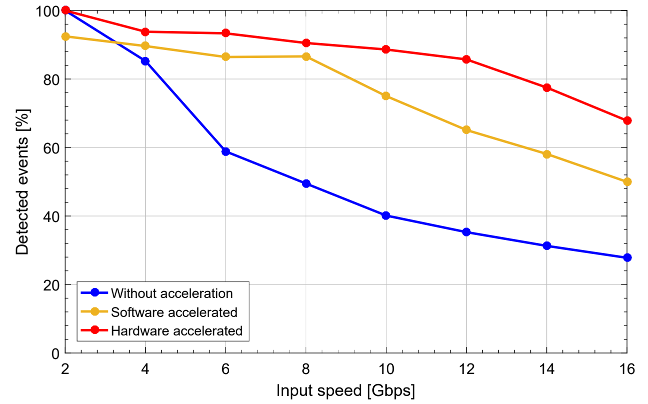


Fig. 9. Percentage of security events detected by Suricata at different speeds.

SuperMicro server with 8 core Intel Xeon E5-2670 CPU operating at 2.6 GHz and with 64 GB of RAM. The same 200 GB PCAP file with real network traffic that we used for initial analysis is also used for these experiments.

Three deployment scenarios are evaluated and compared:

- *Without acceleration*, when Suricata is executed as it is, without any modifications whatsoever. This creates a baseline to evaluate acceleration benefits against.
- *Software accelerated*, where Suricata input software plugins discard packets according to the scheme from Fig. 5, implementing their own software cache.
- *Hardware accelerated* scenario utilizes SDM to discard packets according to the scheme from Fig. 6. Input plugins of Suricata are altered to communicate with SDM Controller and instruct it to offload selected heavy flows (flows that exceed configured N) into the hardware cache.

In both software and hardware accelerated versions, the value of offload threshold N is hand-picked and optimized to obtain the best results. We assume that in a real deployment, N can be automatically adjusted on the fly by the SDM system to adapt to changing network traffic characteristics as already shown in [10]. Basic idea is to lower the value of N (i.e. offloading more) whenever an input buffer overflow (blind discard) is detected and raise it again (i.e. offloading less) when the input traffic rate drops. The threshold adaptation must also consider the current load of the hardware flow cache, which has a limited size. For the best offload ratio, it is advantageous to keep the flow cache nearly full of active records. On the other hand, to maximize the quality of detection we want to forward as many packets to the IDS as it can handle.

A. Complete Ruleset Detection

In the first experiment, we test Suricata with all of the 13 642 rules from the EmergingThreats database [13] (the same as in section III). This configuration enables the most detailed and precise threat detection but it is also extremely demanding on consumed CPU performance per packet.

Fig. 8 shows packet drop rates for each of the tested scenarios. For non-accelerated version (blue), Suricata starts uncontrolled packet drops at input rate between 2 and 4 Gbps. For software accelerated version (orange), the software input plugin actively performs controlled packet discarding reducing the load of the tested IDS (orange dotted line), which significantly helps in reducing uncontrolled packet drops at the input (shown as the decline of the solid orange line from the 100 % edge). The limited CPU performance forces the system to start dropping packets uncontrollably at input rate between 6 and 8 Gbps. With the hardware acceleration, most of the packets never even arrive at the CPU. This is shown as the dotted red line with circles in the graph – CPU (software IDS) needs to process only around 30 % of all packets at high speeds (maximum acceleration rate). At the input speed of around 10 Gbps, the rate of packets passed by the accelerator to the CPU becomes still too high to process, and uncontrollable discarding occurs. Further reducing the rate of packets that are sent to the CPU (increasing the controlled packet discarding) would require lowering of the threshold N to extremely low values. That, however, was not possible, because the hardware flow cache size of SDM implementation is limited to around 250 000 items (flows). Furthermore, reduction of N below a certain point itself can also considerably lower detection quality of IDS (see the left side of Fig. 3).

More important numbers are shown in Fig. 9, which plots the percentage of detected events, related to the 100 % baseline given by offline (non-discarding) analysis of the used PCAP file. This ultimately represents the detection quality of tested IDS for given input rate. Only for the speed of 2 Gbps, when no packets are dropped, all events present in the input data are detected by non-accelerated version (blue). The percentage of all events detected in all scenarios lowers rapidly as the uncontrolled packet drops shown in the previous graph rises. However, the increase in *controlled* discarding, present in both accelerated versions (orange and red), causes only a moderate decline of detection quality.

With the controlled discarding, the system maintains good detection quality for much higher input packet rate. In hardware accelerated scenario (red), the detection rate drops under 80 % only at speeds higher than 12 Gbps, while without acceleration (blue) this drop occurs right after 4 Gbps mark (3× sooner). From a different perspective, the detection quality

of Suricata IDS is enhanced up to 2 or 3× by the proposed hardware acceleration at higher input speeds.

Furthermore, it is worth noting that for 2 Gbps input rate, the software accelerated version still performed some controlled discarding, which reduced detection quality. In hardware accelerated scenario, we employed on the fly accommodation of decision threshold N to higher values for lower speeds based on actual CPU load (note that red dotted line follows blue line in Fig. 8). That is why the hardware accelerated version can reach 100 % detection rate when the system is not overloaded. To further show the benefit of the threshold accommodation, the software accelerated version was left to discard packets unnecessarily even at lower data rates (steady orange dotted line in Fig. 8). This is exactly why the detection rate is lower (only 90 %) even at low input rates. In a real deployment, this configuration can easily be avoided. We also want to point out the fact that in the accelerated versions, only a small portion of packets is actually sent to Suricata for processing (dotted lines in Fig. 8). The detection quality remains still considerably high – around 90 % – due to the increased IDS efficiency, as already predicted by Fig. 4 in section III.

Now we return back to our assumption number 4) from the Introduction and use Fig. 10 to supports its claim. Each measured value from the previous experiment is drawn as one point in the (processed packets × detected events) space. For unaccelerated version (blue), we can see a strong linear correlation (blue dotted line), which supports our initial assumption that uncontrolled discarding is effectively random. This is also our baseline since it represents the default IDS deployment use case. With the hardware accelerated (controlled discarding) version (red), all our measured points are above the baseline, which means that for a given amount of packets processed by the IDS, more events were detected. Note that darker red dots are from measurements at highest speeds, which are negatively influenced by blind discarding. Theoretical data from our analysis in section III are shown as the dotted red line. The reason for the measured data being slightly worse is that there is a small latency in installing packet discarding rules to the SDM hardware. Therefore, the accelerator sometimes lets more than N packets in, which causes the difference between the model and our implementation.

B. Malware Detection

In real network deployments of IDS, it is common to select only a specific subset of available detection rules to focus only on the most critical threats or relevant threats [14], [15]. In the wake of recent massive outbreaks of malware infections, especially ransomware like WannaCry or Petya [16], we focused on malware detection rules here. This way, we select a total of 967 malware oriented rules from the original 13 642. As a result, virtually all reported events are only of ET MALWARE type while other (previously more prevalent) types like ET SCAN, ET DOS or ET POLICY are not detected. With smaller ruleset used in Suricata, a reduction in CPU performance demands per packet is expected to lead to higher achievable speeds.

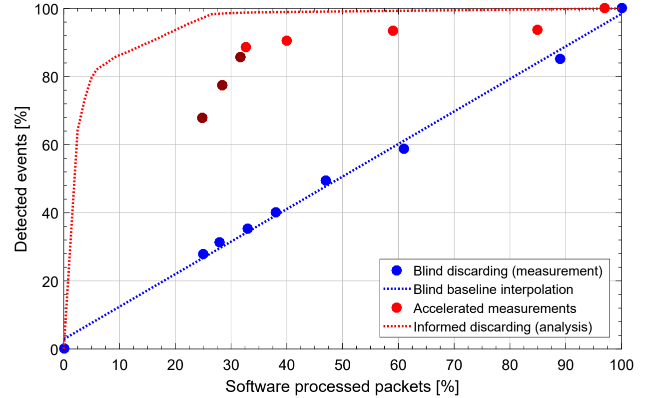


Fig. 10. The relation between processed packets and detected events.

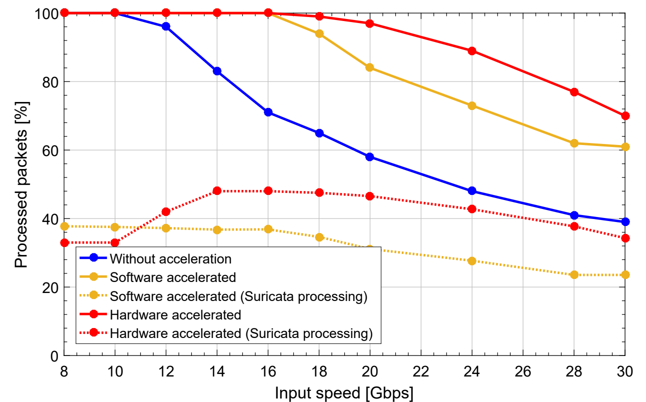


Fig. 11. Percentage of packets processed for different speeds (small ruleset).

Fig. 11 shows packet drop rates for each of the tested scenarios with the smaller ruleset. For non-accelerated version (blue), Suricata starts uncontrolled packet drops at input rate between 10 and 12 Gbps. For accelerated versions the speeds are higher, the software version starts dropping packets uncontrollably at input rate between 16 and 18 Gbps and hardware version starts only at around 20 Gbps. These numbers show that reduced ruleset enables Suricata to reach speeds about 10 Gbps higher as in the previous experiment, while other basic characteristics of these graphs remain the same.

More important numbers are shown in Fig. 12, which again plots the percentage of detected events, related to the 100 % baseline obtained from the offline analysis. For the speeds of up to 10 Gbps, when no packets are dropped, all expected events are detected by non-accelerated version (blue). The percentage of all events detected in all scenarios lowers even more rapidly as before with the rise of uncontrolled packet drops shown in the previous graph. And again, even the high rate of *controlled* discarding, causes only a rather slow drop in detection quality – detection quality drops under 80 % 2× later. At the highest input rates, detection capabilities are enhanced up to 2 or 3× compared to non-accelerated version.

Interesting information can be seen from Fig. 13. Again each measured value from this experiment is drawn as a point in the (processed packets × detected events) space. For

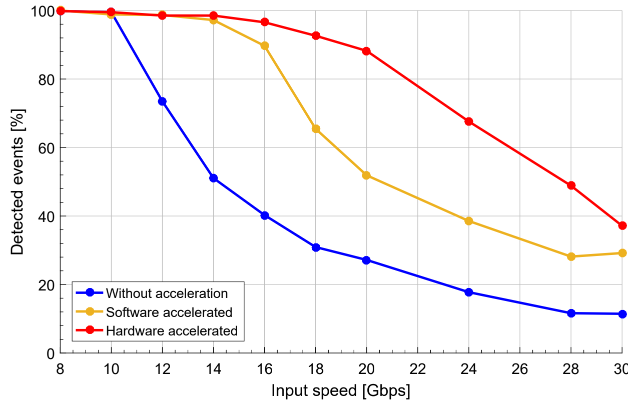


Fig. 12. Percentage of security events detected by Suricata (small ruleset).

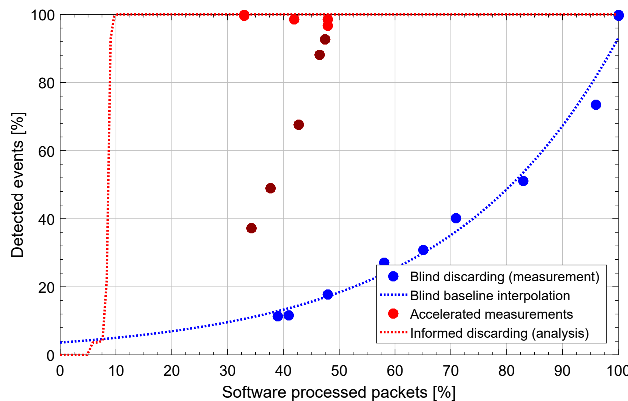


Fig. 13. The relation of processed packets and detected events (small ruleset).

unaccelerated version (blue), a strong exponential baseline correlation (blue dotted line) is now present instead of linear. This further favors the use of acceleration for IDS, as blind discarding has an even worse impact on malware detection rate as on general detections. The exponential correlation here is due to the common need to process multiple subsequent packets of a flow in order to detect a single malware threat. Loss of even one of these packets often leads to failed detection. Theoretical data from an offline PCAP analysis are shown as the dotted red line. Here, the detection rate is even more dependent on the first packets from flows as in the first experiment – all detections occurred between the 4th and 10th packets of all flows. Hardware accelerated (controlled discarding) version (red) again shows that all our measured points are well above the baseline, even the darker red dots representing measurements at highest speeds, which are significantly negatively influenced by blind discarding.

VI. Conclusion

We have designed and tested a new concept of Intrusion Detection System acceleration based on a controlled (informed) reduction of incoming traffic while retaining sufficiently good overall threat detection capabilities. Based on the analysis results showing that the first packets of network flows are the most important for security detections, in our concept, we propose to drop all packets that follow the first N packets of each network flow, where the value of N is optimized on-the-fly based on current IDS load. In other words, if an IDS is overloaded and have to skip processing of some packets, we propose a system for informed selection of which packets to skip (ends of heavy flows) instead of reliance on random buffer overflow mechanism. The proposed concept can be implemented as a pure software system, but can also take advantage of hardware acceleration to achieve even higher IDS speed up. Furthermore, the concept is general enough to be deployable with any software based IDS.

In this paper, we use Suricata IDS for experimental testing on captured data from a real high-speed network. We have tested two basic configurations of Suricata – full ruleset (13 642 rules) and smaller ruleset optimized for malware detections (967 rules). Achieved experimental results conclusively show that our proposed form of informed discarding is considerably better compared to default blind buffer overflow. Utilizing our acceleration concept, we are able to achieve **processing of 2 or 3× higher input link speeds** compared to non-accelerated IDS, while high detection quality is maintained. Or from a different point of view, our acceleration enables the IDS to **detect up to 3× more events** at given high-load compared to deployment without acceleration.

References

[1] M. Roesch et al., “Snort – Network Intrusion Detection & Prevention System,” Aug 2018. [Online]. Available: http://www.snort.org/
[2] ClearFoundation, “l7-filter,” 2018. [Online]. Available: http://l7-filter.clearos.com/
[3] ntop, “nDPI,” Aug 2018. [Online]. Available: http://www.ntop.org/products/ndpi/
[4] V. Paxson et al., “The Bro Network Security Monitor,” Aug 2018. [Online]. Available: http://www.bro.org
[5] M. Jonkman et al., “Suricata,” Aug 2018. [Online]. Available: http://suricata-ids.org
[6] H. Song, T. Sproull, M. Attig, and J. Lockwood, “Snort offloader: a reconfigurable hardware NIDS filter,” in *Field Programmable Logic and Applications (FPL)*. IEEE, 2005.
[7] A. Mitra, W. Najjar, and L. Bhuyan, “Compiling PCRE to FPGA for Accelerating SNORT IDS,” in *Architecture for Networking and Communications Systems (ANCS)*. ACM, 2007.
[8] V. Košař, M. Žádník, and J. Kořenek, “NFA reduction for regular expressions matching using FPGA,” in *Field-Programmable Technology (FPT)*. IEEE, 2013.
[9] N. Weaver, V. Paxson, and J. M. Gonzalez, “The Shunt: An FPGA-based Accelerator for Network Intrusion Prevention,” in *Field Programmable Gate Arrays (FPGA)*. ACM, 2007.
[10] L. Kekely, J. Kučera, V. Puš, J. Kořenek, and A. V. Vasilakos, “Software Defined Monitoring of Application Protocols,” *IEEE Transactions on Computers*, vol. 65, no. 2, 2016, ISSN: 0018-9340.
[11] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, “Enriching Network Security Analysis with Time Travel,” in *Special Interest Group on Data Communications (SIGCOMM)*. ACM, 2008.
[12] V. Stoffer et al., “100G Intrusion Detection,” Aug 2018. [Online]. Available: https://commons.lbl.gov/display/cpp/100G+Intrusion+Detection
[13] Proofpoint Inc., “Emerging Threats Open Ruleset,” Aug 2018. [Online]. Available: https://rules.emergingthreats.net/
[14] M. Jonkman, Aug 2018. [Online]. Available: http://doc.emergingthreats.net/bin/view/Main/NewUserGuide
[15] ——, “What Every IDS User Should Do,” Aug 2018. [Online]. Available: http://doc.emergingthreats.net/bin/view/Main/WhatEveryIDSUserShouldDo
[16] A. Kujawa et al., “Cybercrime tactics and techniques Q2 2017,” 2017. [Online]. Available: https://www.malwarebytes.com/pdf/white-papers/CybercrimeTacticsAndTechniques-Q2-2017.pdf

A.3 Paper III

Detecting Routing Loops in the Data Plane

Jan KUČERA, Ran BEN BASAT, Mário KUKA, Gianni ANTICHI, Minlan YU and Michael MITZENMACHER. "Detecting Routing Loops in the Data Plane". In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT 2020. Barcelona, Spain: ACM, 2020, pp. 466–473. ISBN: 978-1-4503-7948-9.

Detecting Routing Loops in the Data Plane

Jan Kučera
CESNET
jan.kucera@cesnet.cz

Ran Ben Basat
Harvard University and UCL
ran@seas.harvard.edu

Mário Kuka
CESNET
mario.kuka@cesnet.cz

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

Minlan Yu
Harvard University
minlanyu@seas.harvard.edu

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

ABSTRACT

Routing loops can harm network operation. Existing loop detection mechanisms, including mirroring packets, storing state on switches, or encoding the path onto packets, impose significant overheads on either the switches or the network.

We present Unroller, a solution that enables real-time identification of routing loops in the data plane with minimal overheads. Our algorithms encode a varying fixed-size subset of the traversed path on each packet. That way, our overhead is independent of the path length, while we can detect the loop once the packet returns to some encoded switch. We implemented Unroller in P4 and compiled into three different FPGA targets. We then compared it against state-of-the-art solutions on real WAN and data center topologies and show that it requires from 6x to 100x fewer bits added to packets than existing methods.

CCS CONCEPTS

• **Networks** → **Network algorithms**; **Network monitoring**; *Programmable networks*; *In-network processing*;

KEYWORDS

Network algorithms, routing loops, programmable data planes.

ACM Reference Format:
Jan Kučera, Ran Ben Basat, Mário Kuka, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. 2020. Detecting Routing Loops in the Data Plane. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3386367.3431303

1 INTRODUCTION

Real-time detection of traffic loops is essential for the performance of today's networks. Unidentified loops may lead to losses, which in turn increase the tail latency [14]. Also, packet losses due to traffic loops are often interpreted as a signal of congestion, e.g., in TCP, leading to a reduction in throughput [1]. As an example, in a production cluster of 2500 switches, Microsoft reported that traffic trapped in routing loops led to a significant increase in overall traffic [29]. Also, it has been demonstrated that the portion of traffic not caught in the loop but sharing some of the affected links can be severely affected in terms of delay and jitter [14]. Finally, loops are one of the main causes for routing instability which can affect performance on the network as a whole [10, 25, 26]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT '20, December 1–4, 2020, Barcelona, Spain
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431303

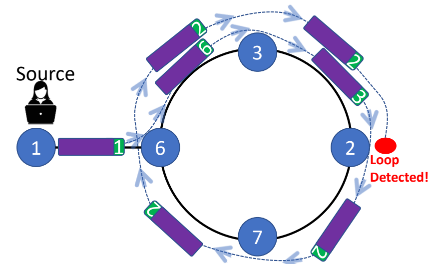


Figure 1: Unroller in action: each packet stores the minimal switch ID seen and resets the stored ID after each phase. When a packet reaches the switch with the stored ID, the loop is reported.

Advanced approaches to loop detection include storing state on switches [19, 24] or mirroring just selected packet header fields [9, 13]. The former leads to significant overhead on switch memory, while the latter leads to significant overhead on the network. Both aspects are important, as the scarce switch SRAM memory can be instead used for ACL rules or customized forwarding [23], while excessive control traffic can have prohibitive data collection overheads [21]. Lately, the recent advances of programmable switches [5] has opened the opportunity to tackle the routing loop detection problem by storing path information directly on packets. For example, the in-network telemetry (INT) [11] allows each switch to put its ID on a packet as it passes by. As a consequence, if a switch sees its own ID on an incoming packet, it can conclude the existence of a routing loop and take appropriate action, e.g., report and reroute the packet. This simple solution suffers from an important drawback: storing the full path information on a packet takes significant header space. For a path of six hops, for example, we need 32 Bytes (8 Byte INT header and 4 Byte switch ID for each hop) [11], which is an overhead of 3.2% for packets with an average size of 1 KBytes. While a reduced overhead can be obtained for specific data center topologies [27], a more generic approach is needed when dealing with any arbitrary big topology.

In this paper, we design Unroller, a solution that enables real-time identification of traffic loops while keeping low overhead on both network and switches. The idea is to store within each packet only *a subset* of the path taken, even a single switch ID, while *guaranteeing*

that a routing loop can be identified in a bounded number of hops. This is possible by dividing the path of a packet in phases, i.e., consecutive series of hops, that increases exponentially, i.e., $1, 2, 4, 8, \ldots$. Every time a switch processes a new packet, it is allowed to store its ID only if it is smaller than the one currently stored into the packet or if we are at the beginning of a new phase. Intuitively, not too long after reaching the loop, the packet will enter a phase that is long enough to reach again the last switch that has updated its ID on the packet (see Figure 1). We evaluate Unroller against state-of-the-art solutions using real WAN and data center topologies. We show that, although storing only partial information into packets might introduce errors, the probability of falsely reporting a loop is negligible in practice. Furthermore, our solution requires from 6x to 100x less bits added to packets than existing methods.

In summary, the main contributions of this paper are:

- We present Unroller, a novel solution that enables real-time identification of routing loops in the data plane by storing information on packets.
- We evaluate Unroller on a number of real WAN and data center topologies. We also implement our solution in P4 and compile into three different FPGA targets.
- We analyze Unroller and rigorously prove performance bounds.
- We open source our code: https://github.com/kucejan/unroller.

2 DESIGN SPACE

Past proposals can be classified into three main categories depending on how they handle the information needed for detecting loops; (1) keep flow state at switches; (2) mirror information at switches; or (3) keep information on packets.

Storing flow information at switches [18, 24] and periodically exporting the information to a collector can put too much pressure on the switch hardware. Indeed, keeping state for a large number of active flows (e.g., up to 100K [22]) is by itself challenging with limited switch space (e.g., 100 MB [20]). Moreover, space is at a premium because operators need the memory for more essential control functions such as ACL rules, customized forwarding [23], and other network functions and applications [16, 20]. The advantage of storing information on switches is the low network overhead; we only need to occasionally export the switches' state for analysis and can avoid using excessive control bandwidth.

Mirroring information at switches upon a packet arrival [13] or after a given timeout [9] creates significant scalability concerns for both trace collection and analysis. The traffic in a large-scale data center network equipped with hundreds of thousands of servers can introduce terabits of traffic [12, 22]. Assuming a CPU core can process tracing traffic at 10 Gbps, on the order of thousands of CPU cores would be required for trace analysis [29], which is prohibitively expensive.

Finally, a third set of solutions propose to keep information on packets [11, 15, 27]. Specifically both INT [11] and Tiny Program Packets [15] suggest a mechanism for each switch to record their ID in the incoming packet. This allows rapid detection of loops in the data plane[1] at the cost of a per packet overhead that grows linearly with the network diameter, i.e, the ID is encoded in 4B and 2B for INT and TPP respectively. With Pathdump [27], the authors instead enable on-line routing loop detection by leveraging the fact that commodity SDN switches can recognize only two VLAN tags in hardware. With this in mind, they consider only scenarios where a third VLAN tag only arises in the presence of a loop, and when an attempt is made to add a third tag, the switch CPU is invoked to manage the loop detection.

[1] If a switch receives a packet with its ID already stored, then most likely the packet has entered in a loop.

Table 1: Comparisons of Unroller and the state-of-the-art solutions for routing loop detection.

Type	Solution	Real Time	Switch Overhead	Network Overhead
On-switch State	FlowRadar [18] Hash IP Traceb. [24]	✗	high	low
Header Mirroring	NetSight [13] Everflow [29] Trajectory Samp. [9]	✗	low	high
Full Path Encoding on Packets	INT [11] TPP [15] PathDump [27]	✓	low	high
Partial Encoding	**Unroller**	✓	**low**	**low**

A last key classification for such algorithms is whether they can detect a loop *in real time*: while a packet is in flight. Real-time detection of loops enables (1) selective reporting: let the packet traverse the loop again to record the identifiers of the participating switches; and (2) active rerouting: forward the packet to a different port in an attempt to avoid packet loss. All existing solutions are either unable to detect loops in real time or have a packet overhead that is linear in the number of hops, as presented in Table 1. Given the design space with the trade-offs current solutions face, in this paper, we answer the following question:

> *Can we design an algorithm that detects routing loops at real time, in the data plane, while keeping low switch and network overheads?*

We show that this is possible with Unroller, a technique to encode only a small subset, e.g., a single identifier, of the switches ID along the path, while *guaranteeing* detection in a bounded number of hops.

3 UNROLLER

One possible approach to detect loops by encoding information onto packets is to store the identifier of all switches that the packet traverses. This is how INT would handle this task. When a switch receives a packet, it checks if it is on the packet's list and, if so, reports a loop. As previously discussed, this generally adds significant bandwidth overhead and should be avoided.

A possible alternative is to store a Bloom filter which encodes the set of visited switches. Intuitively, we can hold a compressed representation of the path and save bandwidth at the cost of false positives. As before, once a switch is reported as positive by the filter, we report a loop. This Bloom filter solution must deal with false positives, but it remains wasteful even without that issue. Intuitively, there is no need to remember *all* switches on the loop, but only *some switch* on the loop. If a packet stores the same switch ID while traversing the entire loop, we can report the loop when we see the repeated switch ID. Let us first assume that the packet's first hop is already part of the loop. In this case, we can record on the packet

Table 2: List of symbols and notations.

Symbol	Definition
B	The number of hops before the loop.
L	The number of switches in the loop.
X	The number of hops before reaching a switch twice (B+L).
b	The phase growth base; the i'th phase lasts for b^i hops.
z	The Unroller bit-overhead on each packet.
c	The number of switch IDs encoded into packets.
H	The number of hash functions used on each switch ID.
Th	The threshold for reporting a loop.

the *minimum* switch ID that it has seen. We are guaranteed to detect the loop after two iterations through the loop; the minimal switch is recorded in the first loop and observed again in the second loop. The problem becomes a bit more complicated when there could be a path of switches the packet traverses before reaching the loop. In this case, the above approach would fail when the minimal identifier appears on the path leading to the loop rather than the loop itself. We suggest the following solution (Table 2 summarizes the notation used in this paper).

Let B be the number of hops *before* the loop and L be the number of switches in the loop. Notice that any algorithm would require the packet to traverse $X \triangleq B + L$ hops before reaching some switch for the second time, which gives a lower bound on the number of hops required for detection. We now show a deterministic algorithm that stores a single switch ID, has no false positives, and finds the loop after at most $4.67X$ hops (without knowing B or L). As before, we keep the minimum identifier we have seen, but now we occasionally *reset* the identifier as though we are restarting, and we gradually increase the resetting intervals.

Our algorithm has a parameter b that determines how aggressively we increase the resetting intervals. The execution takes place in *phases* so that at the end of each phase, we reset the stored identifier; the i'th phase lasts for b^i hops. We prove that after no more than

$$(2L-1) + \max\left\{\frac{2bL-1}{b-1}, bB+1\right\} \leq 4.67X$$

hops (the inequality holds for $b = 4$), the packet reaches a switch that can report the loop. If the switches can perform floating point operations, or if we can compute $\lfloor b^i \rfloor$ for non-integer b using a lookup table, it is possible to optimize the ratio further.

THEOREM 1. *Our algorithm identifies the loop after at most* $(2L-1) + \max\left\{\frac{2bL-1}{b-1}, bB+1\right\}$ *hops* at the worst case.

We split the proof of the theorem into three simple lemmas.

LEMMA 2. *After at most* $\frac{2bL-1}{b-1}$ *hops, we get to a phase that lasts at least* $2L$ *hops.*

PROOF. Let us first denote by p the first phase number that lasts for at least $2L$ hops. Observe that $p = \lceil \log_b 2L \rceil$; the number of hops until we reach this phase is then

$$\sum_{i=0}^{p-1} b^i = \frac{b^p - 1}{b-1} \leq \frac{2bL-1}{b-1}. \qquad \square$$

LEMMA 3. *After at most* $bB + 1$ *hops, the stored ID is from a switch on the loop.*

PROOF. We know that after B hops the packet reaches the loop. We want to show that, once the packet reaches the first switch in the loop, after at most $(b-1)B+1$ additional hops the phase ends and the identifier resets, at which point the stored ID will be from a switch on the loop. Since the previous phase (if one exists) before reaching the first switch in the loop cannot last more than B hops, it follows that the current one must end within $b \cdot B$ hops. A slightly tighter analysis shows that it actually ends within $(b-1)B + 1$ additional hops. Denote the phase number when we reach the first switch on the loop by p. By the end of this phase, the stored ID will be from a switch in the loop. We have that

$$\frac{b^p - 1}{b-1} = \sum_{i=0}^{p-1} b^i < B,$$

and thus $p < \log_b(B(b-1)+1)$. As the current phase is of length b^p, the lemma follows. $\square$

LEMMA 4. *If at the start of a phase the stored identifier is from a switch on the loop and the phase lasts at least* $2L-1$ *hops, then we terminate after at most* $2L-1$ *hops.*

PROOF. Let v be the switch with the smallest ID in the loop. From (1) and (2) it follows that (i) the packet has already reached the loop, (ii) that the ID that is stored of a node in the loop and that (iii) the phase is long enough; after at most $L-1$ hops the packet reaches v and thereafter does not change the stored identifier. After another L hops it reaches v again and the loop is reported. $\square$

3.1 Lower Bound

Our algorithm only guarantees detection after $4.67X$ hops. An interesting question is *what is the minimal number of hops required for loop detection by an algorithm that stores a single identifier*? As we now state, deferring the details to Appendix A, any deterministic algorithm that does not assume knowledge of B requires at least $\approx 3.73X$ hops detection time. This shows that our approach is not far from optimal for deterministic algorithms.

THEOREM 5. *Any deterministic loop detection algorithm that stores a single identifier requires at least* $3.73X \cdot (1 - o(1))$ *hops for detection in the worst case.*

3.2 Average Case Analysis

The above analysis shows that we require at most 4.67 times as many hops to report a loop than the costly algorithm that stores the entire path. This analysis holds at the worst case, but it is also useful to analyze the *average case*. For reasoning about the average case, we require that the switch IDs will be random so that each switch has the same probability of holding the smallest ID. If this is not the case, we can use hashed switch IDs for the algorithm; these may introduce false positives (similar to the Bloom filter algorithm), but the trade-off between overhead to error is much more favorable in our algorithm. Alternatively, we may consider a random *permutation* on the switch identifiers that is known to all switches. We have no false negatives and all loops are still guaranteed to be reported. We show here that in the average case, the loop is detected after at most $3X$ hops, when $b = 3$. There are three cases to consider, depending on the length, denoted q, of the first phase that begins on the loop with length at least L.

If $q = (1+\alpha)L$ for some $0 \leq \alpha \leq 1$, then up to this phase, by our previous analysis, the packet has traversed at most $(q-1)/(b-1)$ hops. In this phase, since the switch with the minimal identifier

is equally likely to be any on the loop, we hit the switch with the minimal identifier twice with probability α, and in this case, the expected number of additional hops is $(1+\alpha/2)L$. If the switch with the minimal identifier is not hit twice in this phase, which occurs with probability $(1-\alpha)$, it will be hit twice in the next phase, after an expected $(1+\alpha)L+(1+(1-\alpha)/2)L$ hops. Overall, the total expected number of hops is at most

$$L\left(\frac{1+\alpha}{b-1}+2-\frac{\alpha^2}{2}+\frac{(1-\alpha)^2}{2}\right)=L\left(\frac{1+\alpha}{b-1}+2.5-\alpha\right).$$

For $b=3$, this expression is at most $3L$.

If $2L<q\leq bL$, then the loop will be found in this phase, after an expected $3L/2$ hops. Up to this point, the packet has traversed at most $bL/(b-1)$ hops, which is also $3L/2$ when $b=3$, giving an overall count of at most $3L$ hops.

If $q>bL$, then the previous phase was at least L hops but did not start on the loop. In this case, we have traversed at most $bB+1$ hops, and B is at least $L/(b-1)$. The loop will be found in this phase, after an expected $3L/2$ hops. In this case, X is at least $B+L$, and the expected number of hops to find the loop is at most $bB+1+1.5L\leq 3X$.

For $b=3$, in all cases, we have shown that after at most $3X$ hops we identify the loop, and this is the best choice for b for the average case analysis. The average case analysis provides a different bound than the lower bound for the worst-case analysis, and uses a different choice of b than our best upper bound for the worst-case analysis.

3.3 Reducing the Per-Packet Overhead

The above algorithms suggest storing a switch identifier on each packet. However, in some cases, the identifiers may be large and pose an undesirable overhead. In such cases we propose to *hash* the switch identifiers into z bits. That is, instead of storing a switch identifier, Unroller will encode its smaller hash onto the packet. This reduces the number of bits added to each packet but introduces false positives as two switches not on a loop, but simply on the path, may have the same hash.

We propose a simple counting technique that exponentially reduces the probability of false positives. We add a small counter that tracks the number of times we have seen a switch whose hash matches the one on the packet. Once the counter reaches a predetermined threshold of Th, we report the loop. If there is a loop, the counter eventually reaches Th; if there is no loop, a false positive now requires Th switches on the path to have the same hash, which is much more unlikely. This solution requires an additional $\lceil\log_2 \mathit{Th}\rceil$ bits per packet[2], but significantly reduces the chance of false reporting. For example, on a path of length 20 hops, with $\mathit{Th}=4$, $z=7$, and $b=4$, the chance of false positives is lower than 10^{-5} while using $(7+2)$ bits of overhead per packet. Therefore, we can run with only a few false positives while reducing the overhead by 72%. We note that using $\mathit{Th}>1$ does not come for free as it increases the number of hops required for detection (namely, by $(\mathit{Th}-1)\cdot L$ hops).

3.4 Trading Bandwidth for Convergence

So far, we have allowed the algorithm to store a single identifier. As we saw, this allows us to derive algorithms that are 3-4.67 times slower than the X hops lower bound (which assumes no bandwidth constraints). A natural question is whether we can get faster detection if we allow storing more than one identifier but not the entire path.

The main drawback of storing just one identifier is that we had to balance the rate in which we increase the reset time (the parameter b) in a way that we do not lose much when $B\gg L$ but also when $L\gg B$.

In Appendix B, we explore how to use multiple identifiers on packets to reduce the expected number of hops required for detection. Specifically, we show that by using H hash functions and storing c identifiers for each (a total of $c\cdot H$ identifiers), we can reduce significantly the number of hops. Intuitively, using multiple hashes allows different switches to have "minimum IDs" with respect to some hash function while each of the c identifier stored for a hash function tracks the minimum only on a $1/c$-fraction of the phase.

Table 3: Parameters encoded in the packets' header being used by our algorithm.

Values encoded in packet headers	
X_{cnt}[3]	The current number of visited hops of the packet along its path.
$SW_{ids}[]$	The array of the current switch IDs seen.
Th_{cnt}	The current value of the threshold counter.

3.5 Discussion

Here, we discuss the importance of phases and the trade-off associated with the identification of the switches involved in the loop.

Importance of switch ID resetting. Let us assume a variant of Unroller where each switch inserts its ID, with a set probability, only if the incoming packet does not already carry the maximum number of IDs. This solution works well when the packet's first hop is already part of the loop. If, however, the packet encounters a number of hops before the loop, this solution might introduce false negatives when only the pre-loop IDs are stored within the packet. By introducing phases in the algorithm, we force the values already stored within the packet to be overwritten at times, thus avoiding this problem.

Identification of switches involved in a loop. There is an obvious trade-off between the detection of the loop and the additional identification of all the switches involved. Directly recording as many IDs as possible into packets aids the discovery of network elements involved in the loop. However, this comes at the cost of additional overhead on packets, which leads, in normal conditions, to undesired effects on network performance [3]. With Unroller, we opted for a lightweight mechanism to detect loops. Once a loop is identified it is possible, for example, to tag the packet to collect the involved switch IDs and send a report for analysis.

4 IMPLEMENTATION

We implemented Unroller using the $P4_{16}$ language [8] and compiled on a software target BMv2 [7], and three FPGA based targets using the P4-To-VHDL compiler [4].

P4 implementation. The core of Unroller is implemented in 60 lines of code. The implementation consists of a single control block applied at the ingress pipeline. The input program parameters are b, z, c, H and Th (Table 2). Additionally, Unroller requires extra

[2] We do not need to encode the value Th but report the hop that sees a hash match when the counter equals Th-1.

[3] In cases where the hop number can be inferred from the TTL (e.g., see [2, 3]), we can avoid storing X_{cnt} and reduce the bit overhead.

Table 4: Architecture HW resources utilization results.

Platform	LUTs	REGs	BRAM	Frequency
Virtex 7	26 234 (7.23 %)	29 944 (4.13 %)	396 kb (1.17 %)	224 MHz
Virtex US+	26 221 (7.23 %)	30 520 (4.21 %)	684 kb (2.02 %)	225 MHz
Stratix 10	21 917 (1.17 %)	45 907 (1.22 %)	301 kb (0.12 %)	189 MHz

information being carried on each packet, as summarized in Table 3. Observe that X_{cnt} requires at most 8 bits [3], $SW_{ids}[]$ takes $c \cdot H \cdot z$ bits, while Th_{cnt} only needs $\log_2 Th$ bits Here, we assume that each switch has an unique identifier, stored in a register alongside all the aforementioned configuration parameters. No match-action tables are needed. All the logic fits in a single `apply` section of the control block. Specifically, for each incoming packet, we (1) read the configuration parameters from the registers and increment X_{cnt}; (2) evaluate the hash functions to randomize the switch ID; (3) check if one of the IDs ($SW_{ids}[]$) stored within the packet have to be updated; (4) drop the packet and inform the controller when a loop is identified. Unroller updates the IDs list present in the packet header only if either there is space for a new value, the current switch has an ID smaller than the one currently stored in the list, or if the packet enters a new phase. This is the case when the X_{cnt} counter stored within the incoming packet is equal to a power of the phase growth base b. Fortunately, for $b = 2$ or $b = 4$, this operation can be performed using standard bitwise checks. Th_{cnt} counter tracks the number of times the packet has seen a switch whose ID matches one of the stored values in the list. Once the counter reaches a predetermined threshold of $Th - 1$, a loop is reported.

Compiling Unroller to programmable switches. We compiled Unroller on the P4 software switch target based on the behavioral model (BMv2). Here, the main constraint is the number and pattern of accesses to on-chip registers [20]. To reduce the number of operations, we used a 256-sized lookup table that records, for each possible X_{cnt}, whether it is the start of a new phase. Alternatively, it is possible to store pre-hashed identifiers into registers, to reduce the number of hash operations. Unroller requires two pipeline stages, uses minimal resources, and does not store any per-flow state in the switch. Furthermore, if the set phase growth base b is not a power of two, a lookup table is necessary for determining the packet's phase. This is because specific operations such as division or power evaluation are not natively supported by hardware.

Compiling Unroller to FPGAs. We used the P4-To-VHDL compiler to port the produced P4 code to different FPGA chips. This was not a one-step process as the original code needed a few adaptations to meet FPGA timing constraints. Specifically, the compiler allows calling actions that manipulate packets only from a match-action table and not directly from a control block. Because of this, we added a dummy match-action table with a single default action unconditionally manipulating the packet. We compiled the Unroller logic into three different FPGA-based targets supporting 100GbE ports: Xilinx Virtex 7 (model XCVH580T), Xilinx UltraScale+ (model XCVU7P) and Intel Stratix 10 (model 1SG280HU). Table 4 shows the chip occupancy and the maximum frequency for all the platforms. Here, we can see that Unroller logic is lightweight, requiring less than 8% of chip resources. Since the synthesized architectures are fully pipelined, i.e., capable of processing a new packet every clock cycle, the frequency can be directly correlated with the maximum achievable throughput: ~220 Mpps for Xilinx devices, and ~190 Mpps for the Intel platform. This is more than 100 Gbps for minimum-sized

Table 5: Unroller vs. state-of-the-art solutions on real topologies.

Topology	# of Nodes	Diameter	PathDump Overhead *(bits)*	Bloom filter Overhead *(bits)*	Unroller Avg Time *(#hops/X)*	Unroller Overhead *(bits)*
Stanford	16	2	×	171	1.74	25
BellSouth	51	7	×	189	1.56	25
GEANT	40	8	×	608	2.13	27
ATT-NA	25	5	×	608	2.15	27
UsCarrier	158	35	×	2466	2.47	28
FatTree4	20	4	64	414	1.73	28

Ethernet packets. Given the targets are dimensioned for 100GbE processing, we can fairly state that Unroller logic does not introduce any throughput degradation.

5 EVALUATION

We evaluated Unroller with a Python simulator that generates paths based on the required number of hops before entering a loop (B) and the number of hops comprising the loop itself (L). Unless otherwise stated, each data point reflects 3M runs. Switch identifiers are randomly generated 32-bit numbers, and the default Unroller configuration parameters are $b = 4$ phase base, $c = 1$, $H = 1$ (one hash function) and $Th = 1$ reporting threshold (see Table 2 for notation description).

Comparing Unroller to state-of-the-art solutions. Here we used several real topologies, with different sizes, spanning from WAN to data centers [17, 28]. We compared the loop detection capabilities of Unroller against state-of-the-art solutions that work in real time: (1) PathDump [27] and (2) an especially crafted approach that adds a Bloom Filter into packets to store switch IDs. The former adds a fixed overhead on each packet, i.e., 64 bits, and does not experience false positives, but can only be applied to a very limited set of topologies [27], e.g., FatTree and VL2. By employing a probabilistic data structure to store switch IDs, the latter can introduce false positives, as Unroller does. To compare both solutions fairly, we randomly picked two nodes in each considered topology and selected a shortest path between them. Out of all possible loops that intersect with that path, we picked one uniformly at random. We then measured, over 3M runs, the minimum overhead (in bits) needed in each packet so that no false positives were reported. Table 5 shows the results. Unroller can detect loops, without experiencing any false positives, using a very small packet overhead. Depending on the topology, our solution requires from 6x to 100x fewer bits than the Bloom Filter counterpart. This comes at the expense of detection speed: while the Bloom Filter can identify a loop as soon as a switch is hit twice by the same packet, Unroller might require one or two extra passes over the loop, as reported in the *Avg Time* column in Table 5. INT would require packets to store an increasing number of switch IDs at each hop, making this approach more expensive (in terms of per-packet bit overhead) than those previously discussed.

Sensitivity analysis. Here, we aim at assessing how the different parameters introduced in this paper (see Table 2) might affect Unroller performance. Unless otherwise stated, the adopted default values are the following: $B = 5$ hops before the loop, $L = 20$ loop hops, $z = 32$ bits per packet, $c = 1$, $H = 1$ (one hash function) and $Th = 1$ reporting threshold. We first evaluate the average detection time, measured as the ratio between the number of hops required for detection and the $X = B + L$ hops lower bound. This time is affected by the loop

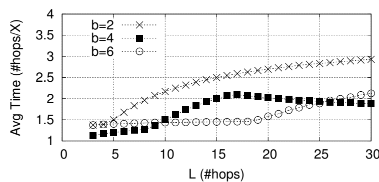


Figure 2: Detection time varying L and b.

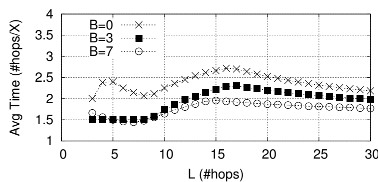


Figure 3: Detection time varying L and B.

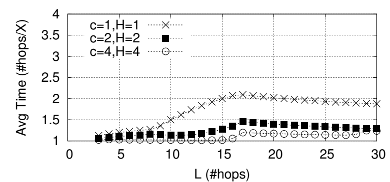
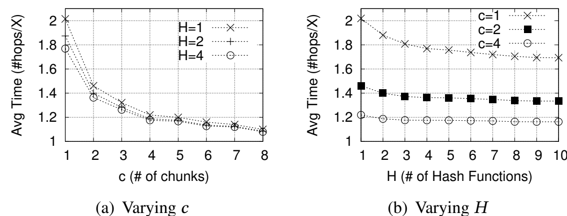
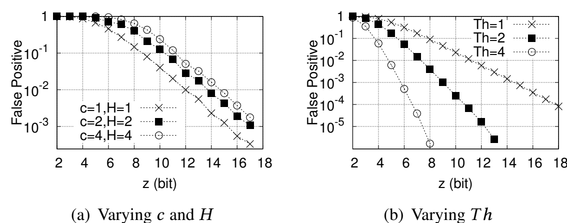


Figure 4: Detection time varying L and c, H.



(a) Varying c (b) Varying H

Figure 5: Detection time for different c, H configurations.



(a) Varying c and H (b) Varying Th

Figure 6: False Positives when compressing switch IDs to z bits.

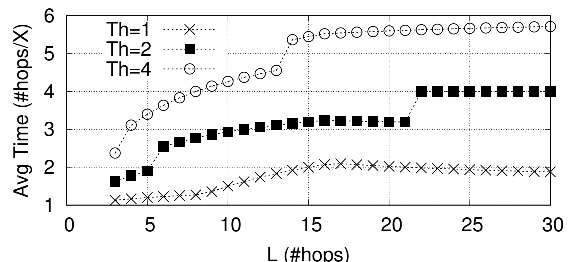


Figure 7: Detection time using counting technique varying Th.

length L when storing only a single full switch ID in each packet. Figure 2 shows the relationship when varying the b parameter. The smaller the value of b, the more aggressively Unroller resets the switch ID stored in the packet, causing an increase of the average detection time. Figure 3 shows the impact of B when b is fixed to 4. Here, the average detection time increases when B decreases. This is the effect of the resetting interval mechanism.

In Figure 4, we fixed $b = 4$ and $B = 5$ and studied the effect on the average detection time of partitioning each phase into c chunks and randomizing the switch ID using H hash functions. Specifically, we stored $c \cdot H$ hashed switch identifiers into each packet. All of the stored IDs are compared to the current switch identifier, and a loop is reported if a match is found. Clearly, the more chunks and hashes used, the better it is for the average detection time. Figures 5(a) and 5(b) show in more details the individual impact of parameters c and H to the detection time. We can see that the improvement is greater when we are increasing the number of chunks c when compared to increasing the number of hashed switch identifiers H. This means that Unroller is more sensitive to the number of chunks rather than the number of hash functions.

Although storing multiple identifiers in the packet ($c > 1$ or $H > 1$) improves the average loop detection time, it also imposes a bigger overhead on each packet. Thus, we analyzed the extension of the algorithm, which reduces the per-packet overhead by compressing the switch identifiers into z-bit values. This practice, however, may introduce false positives. We test using a path length of 20 hops, with $B = 20$ and $L = 0$. As the adopted path does not contain loops, any reported loop by Unroller is a false positive. Figure 6(a) depicts the effect of the compression over the false positives when varying c and H and keeping the hash width z. Partitioning of phases ($c > 0$) or storing more hashed identifiers ($H > 0$), if combined with the compression, increases the false positive rate but leads to faster detection of loops (Figure 4).

Figure 6(b) shows similar results when the threshold technique to reduce the false positive rate is deployed. In this case, the false positives are reduced exponentially with the size of the threshold. However, this comes at the cost of a slight increase in the average detection time, as demonstrated in Figure 7.

6 CONCLUSION

In this paper, we presented Unroller, a lightweight loop detection solution that is readily deployable on emerging technologies such as programmable switches. We evaluated our solution and showed that it could quickly and accurately detect routing loops using a minimal bit-overhead on packets. Further, Unroller does not store state on switches, leaving their scarce memory to other applications. Unlike some of the existing solutions, Unroller can identify loops in real time, by the switches themselves and without a remote analysis node. We envision that such a capability would enable rerouting mechanisms that could prevent packet losses that happen when packets traverse a loop until their TTL zeros out. For example, recently introduced solutions to enable near-optimal compression of backup rules [6] can be adopted in cooperation with Unroller to quickly reroute packets on pre-determined backup ports upon the detection of a loop.

Acknowledgments. We thank our shepherd and anonymous reviewers for their valuable comments. This work was supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 co-funded by the Ministry of Education, Youth and Sports of the Czech Republic and European Regional Development Fund, by the Brno University of Technology under number FIT-S-20-6309, by NSF CCF-1563710, CCF-1563710, and CNS-1834263, the Zuckerman foundation and the UK's EPSRC under the projects EARL (EP/P025374/1) and NEAT (EP/T007206/1).

A LOWER BOUND

Observe that any such algorithm can be determined by the interval it takes before resetting the identifier ($b^0, b^1, \ldots$, in our algorithm). Let x_1 denote the number of hops before the first reset, x_2 denotes the number of hops before the second one, etc. We denote by $\beta \triangleq \max_{n\in\mathbb{N}} \left\{ \frac{x_n}{\sum_{i=1}^{n-1} x_i} \right\}$ the maximal growth rate in the resetting periods. Let n be such that $\frac{x_n}{\sum_{i=1}^{n-1} x_i} = \beta$ and denote $y = \sum_{i=1}^{n-1} x_i$.[4] We also note that by setting a minimal value for n (i.e., $\beta \triangleq \max_{n\in\mathbb{N}, n\geq T} \left\{ \frac{x_n}{\sum_{i=1}^{n-1} x_i} \right\}$ for some T) we get an arbitrarily large number of hops before possible detection (X). We start with some lemmas.

LEMMA 6. *Any deterministic algorithm that stores a single identifier must make at least* $(\beta + 1) \cdot X - O(1)$ *hops before identifying a loop.*

PROOF. Let $B = y + 1, L = 2$, and consider the case where the minimal identifier is at the last hop before the loop. In this case, the algorithm will reset after y hops, then store the minimal identifier, and will not detect loop before the next reset. Therefore, it will only report the loop after $y + x_n + 2L - 1 = y(1 + \beta) + 2L - 1 = (\beta + 1) \cdot X - O(1)$. □

LEMMA 7. *Any deterministic algorithm that stores a single identifier must in the worst case use at least*

$$\min\{4, (3 + 2/\beta)\} \cdot X - O(1)$$

hops before identifying a loop.

PROOF. Throughout the lemma, we assume that $B = 0$ and consider the value of L. We take cases on β.

If $\beta \leq 0.5$ then the algorithm is not guaranteed to find the loop (it will keep on resetting before that, e.g., for $L = x_1$).

When $0.5 < \beta < 1$, we consider $L = \lfloor 2y/3 \rfloor + 1$ and assume that the minimal identifier is the last switch in the loop. Therefore, after y hops the packet has not reached the minimum for the second time and its identifier is reset. Then, before reaching the minimum for the second time, we have a reset at $x_n + y < 2y$ hops. Specifically, this means that the cycle is detected only after $4L - 1 = 4X - O(1)$ hops.

Next, consider $1 \leq \beta < 2$ (i.e., in this case, we have $x_n < 2y$). Let $B = 0, L = y + 1$, and consider the case where the minimal identifier is reset after this y-hops (i.e., it is at the end of the loop). Since x_n, we will not complete two cycles before the next reset; therefore, the loop is detected after no fewer than $4L - 2 = 4X - O(1)$ hops.

Finally, let $\beta \geq 2$. Here, let $L = \lceil \beta y/2 \rceil + 1$ and consider the case where the minimal identifier is the y'th hop in the loop. The algorithm will reset the identifier after seeing the minimum for the first time. It will then complete an entire cycle before seeing it again, and just before reaching it for the third time, it will reset again (as we reach $y + x_n$ hops). Therefore, the algorithm will make at least $y + x_n + 2L - 1 = y(\beta + 1) + 2L - 1 = (3 + 2/\beta)L - O(1)$ hops. □

We now infer the correctness of Theorem 5.

[4] For simplicity, we assume that such exists, otherwise the result will hold up to a term that vanishes as the loop grows longer.

PROOF. Using the two lemmas above, we have that the detection time is lower bounded by

$$\max\{\beta + 1, \min\{4, 3 + 2/\beta\}\} \cdot X - O(1)$$

hops. Taking the minimum over all real β values we get a lower bound of $(2 + \sqrt{3})X - O(1) > (3.73 - o(1))X$. □

B USING MULTIPLE HASHES

Given an integer parameter $c \in \mathbb{N}$, consider partitioning each phase into c *chunks*. Intuitively, we are going to store c times as many identifiers, but each will only be active in a $1/c$ fraction of the phase. Specifically, during phase p, chunk j will get the minimal identifier of hops $\lceil b^p/c \cdot (j-1) \rceil, \ldots, \lceil b^p/c \cdot j \rceil - 1$. The algorithm still compares the current switch identifier to all of the stored IDs and reports a loop if it finds a match.

The analysis only needs to change slightly: Lemma 3 can now show that after about at most $B + (b-1)B/c + 1$ hops we have an identifier in the loop. In turn, this gives that the overall number of hops reduces to at most

$$2L + \max\left\{\frac{2bL - 1}{b - 1}, B + (b-1)B/c + 1\right\}.$$

As an example, if we are allowed to store several identifiers, we can set $c = 2$ and $b = 7$ for a detection after at most $4.33X$ hops at the worst case.

Next, if we allow randomization, we can also consider using $H \in \mathbb{N}$ *hash functions*. Specifically, we assign each switch s with H identifiers $\{h_i(s) \mid 1 \leq i \leq H\}$ using random independent hash functions $h_1, \ldots, h_H$. A packet now contains H IDs $m_1, \ldots, m_H$, one for each minimum obtained by $h_1, \ldots, h_H$. When reaching a switch s, we compute its hashes and check if any of them matches the ones on the packet (i.e., whether there exists $i \in \{1, \ldots, H\}$ for which $h_i(s) = m_i$), and if so report a loop. Otherwise, for all $i \in \{1, \ldots, H\}$ we set $m_i = \min(m_i, h_i(s))$ if we are in the middle of a phase, or $m_i = h_i(s)$ if the last phase has ended and a new phase begun. Intuitively, if the phase is enough to complete two cycles over the loop, we can get *some* switch on the loop with *some* minimal identifier faster than if we had a single identifier. This is because the expectation of the minimum among H uniform variables in $\{0, \ldots, L-1\}$ has an expectation lower than $L/(H+1)$. Similarly, if the phase covers the loop $1 + \alpha$ times for some $\alpha \in [0, 1)$, the chance that we will get a minimal identifier in the first αL hops increases from α to $1 - (1 - \alpha)^H$. It then takes another L hops to complete another cycle and report the loop.

REFERENCES

[1] 1997. Packet Loss Impact on TCP Throughput in ESnet. (1997). http://fasterdata.es.net/network-tuning/tcp-issues-explained/packet-loss/.

[2] R. Ben Basat, G. Einziger, and B. Tayh. 2020. Cooperative Network-wide Flow Selection. In *International Conference on Network Protocols (ICNP)*. IEEE.

[3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[4] Pavel Benáček, Viktor. Puš, Hana Kubátová, and Tomáš Čejka. 2018. P4-To-VHDL: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems* 56 (2018), 22–33.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*.

[6] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute. In *Conference on Emerging Networking Experiments And Technologies (CoNEXT)*. ACM.

[7] P4 Language Consortium. 2018. P4 Switch Behavioral Model. (Jan 2018). https://github.com/p4lang/behavioral-model.

[8] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification, version 1.0.0. (Jan 2018). http://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[9] N. G. Duffield and Matthias Grossglauser. 2001. Trajectory Sampling for Direct Traffic Observation. In *Transactions on Networking, Volume: 9, Issue: 3*. IEEE/ACM.

[10] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2004. The Case for Separating Routing from Routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*. ACM.

[11] The P4.org Applications Working Group. 2018. In-band Network Telemetry (INT) Dataplane Specification. (Jan 2018). https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf.

[12] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[13] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[14] Urs Hengartner, Sue Moon, Richard Mortier, and Christophe Diot. 2002. Detection and Analysis of Routing Loops in Packet Traces. In *Workshop on Internet Measurment (IMW)*. ACM.

[15] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating Systems Principles (SOSP)*. ACM.

[17] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[18] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[19] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[20] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[21] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[22] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[23] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. DC.P4: Programming the Forwarding Plane of a Datacenter Switch. In *Symposium on Software Defined Networking Research (SOSR)*. ACM.

[24] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. 2001. Hash-based IP Traceback. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[25] Ashwin Sridharan, Sue B. Moon, and Christophe Diot. 2003. On the Correlation between Route Dynamics and Routing Loops. In *Internet Measurement Conference (IMC)*. ACM.

[26] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Mao Morely, Scott Shenker, and Ion Stoica. 2005. HLP: A Next Generation Inter-domain Routing Protocol. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[27] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with Pathdump. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association.

[28] James Hongyi Zeng and Peyman Kazemian. 2012. Mini-Stanford Backbone. (2012). https://reproducingnetworkresearch.wordpress.com/2012/07/11/atpg/.

[29] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

A.4 Paper IV

Enabling Event-Triggered Data Plane Monitoring

Jan KUČERA, Diana Andreea POPESCU, Gianni ANTICHI, Han WANG, Andrew W. MOORE and Jan KOŘENEK. “Enabling Event-Triggered Data Plane Monitoring”. In: *Proceedings of the 2020 SIGCOMM Symposium on SDN Research*. SOSR 2020. San Jose, CA, USA: ACM, 2020, pp. 14–26. ISBN: 978-1-4503-7101-8.

Enabling Event-Triggered Data Plane Monitoring

Jan Kučera
CESNET
jan.kucera@cesnet.cz

Diana Andreea Popescu
University of Cambridge
diana.popescu@cl.cam.ac.uk

Han Wang
Barefoot Networks
hanwang@barefootnetworks.com

Andrew Moore
University of Cambridge
andrew.moore@cl.cam.ac.uk

Jan Kořenek
Brno University of Technology
Centre of Excellence IT4Innovations

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

ABSTRACT

We propose a push-based approach to network monitoring that allows the detection, within the dataplane, of traffic aggregates. Notifications from the switch to the controller are sent only if required, avoiding the transmission or processing of unnecessary data. Furthermore, the dataplane iteratively refines the responsible IP prefixes, allowing the controller to receive information with a flexible granularity. We implemented our solution, Elastic Trie, in P4 and for two different FPGA devices. We evaluated it with packet traces from an ISP backbone. Our approach can spot changes in the traffic patterns and detect (with 95% of accuracy) either hierarchical heavy hitters with less than 8KB or superspreaders with less than 300KB of memory, respectively. Additionally, it reduces controller-dataplane communication overheads by up to two orders of magnitude with respect to state-of-the-art solutions.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network measurement**; *Programmable networks*; *In-network processing*.

KEYWORDS

Network measurements, traffic aggregates, Elastic Trie, P4.

ACM Reference Format:
Jan Kučera, Diana Andreea Popescu, Han Wang, Andrew Moore, Jan Kořenek, and Gianni Antichi. 2020. Enabling Event-Triggered Data Plane Monitoring. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3373360.3380830

1 INTRODUCTION

Network management practices can be performed efficiently if high-volume traffic clusters are promptly detected [20, 24, 32, 39, 48]. Indeed, spotting a single source or destination that sends or receive a significant amount of data (heavy hitter) is beneficial for accounting [21, 24], or traffic engineering [6, 25]. In contrast, detecting a source that reaches multiple distinct destinations (superspreader) is needed for worm or scan detection [50, 51]. Finally, finding which flow contributes the most to the traffic pattern changes in a short period of time (change detection) is of paramount importance in the context of anomaly detection [36, 37]. All of the aforementioned events exhibit high-volume traffic clusters in a different way: while in the context of heavy hitters the "cluster" relates to packets or bytes arrival rate, for superspreaders the interest shifts to the flows arrival rate.

In the past, the detection of those events were performed outside the dataplane in software collectors. Switches, to lower overheads and data collection bandwidth at the cost of estimation accuracy [14, 22, 40], employed packet sampling and exported statistics using well known protocols such as NetFlow [16] or sFlow [2]. Lately, the advent of programmable switches [8] has enabled the possibility of extending dataplane functionality with more advanced traffic analysis features. Nonetheless, current devices have constrained resources, requiring clever solutions to deal with both computation and memory limitations. Such restrictions have led the research community to deal with a specific trade-off: while a single use-case can be easily enabled in the dataplane, scaling to more requires the help of a central controller. Indeed, recent proposals that push more computation in the switch focus only on one specific goal: detection of the top-k heavy hitters [48] or microbursts [13]. In contrast, solutions that aim for a more generic approach aggregate traffic information in probabilistic data structures, i.e, sketches [30, 39, 52, 54], which are then entirely exported to a controller for analysis. Despite the use of sketches results in a very flexible and generic approach to network monitoring, the controller still needs to receive the generated information from the dataplane at a fixed time interval, and then estimate the various application-level metrics of interest. Such an architecture has similar drawbacks to that of the OpenFlow (OF) protocol: the ability to apply network policy updates based on the received data depends on the switch-controller's interactions capabilities of collecting statistics at short time scales [20].

In this paper we start from the observation that important network management practices [20, 24, 32, 39, 48], i.e., traffic engineering, security, benefit if heavy hitters, superspreaders and traffic pattern changes are promptly detected. We thus build a solution which is capable of detecting the mentioned network events entirely in the dataplane, by iteratively tracking the responsible IP prefixes and only subsequently informing the controller. We designed a new data structure, *Elastic Trie*, with the constraints of emerging programmable switches in mind, and present its implementation in both P4 and for two different FPGA devices. The idea is to build in the switch a prefix tree that continuously grows or collapses to focus only on the prefixes that account for a "large enough" share of the traffic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR '20, March 3, 2020, San Jose, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380830

Network event	Management task
(Hierarchical) Heavy Hitters	accounting [21, 24], traffic engineering [6, 25]
Changes in traffic patterns	anomaly detection [36, 37]
Superspreaders	worm [51], scan [50], DDoS detection [54]

Table 1: Three network events for many use-cases.

This enables the detection of either (hierarchical) heavy hitters or superspreaders, and at the same time by looking at its growing rate it is possible to identify changes in the traffic patterns. Elastic Trie (ET) shares some high level principles with recent solutions for network monitoring such as Marple [42] and Sonata [28], but it is fundamentally different. Marple focuses mostly on flow performance metrics and not on traffic aggregates. In contrast, Sonata enables operators to get insights on traffic volumes and anomalies, but both requires a central controller to iteratively refine the query to efficiently capture only the traffic that pertains to the operator's query. ET performs such a refinement within the dataplane, thus completely offloading the control path.

The main contributions of the paper are as follows:

- We propose a *push-based* approach to network monitoring, where the dataplane informs the control plane only when a specific network event is detected.
- We present a data structure that enables the detection of changes in network traffic and, at the same time, detects either hierarchical heavy hitters or superspreaders entirely in the dataplane. Our solution iteratively refines the responsible prefixes so that the controller receives a finer or coarser grained information depending on the desired reporting time.
- We implemented our idea in P4 using match-action tables and for two different FPGA devices. We finally demonstrate its performance in terms of throughput and latency, and its detection capabilities by evaluating it through trace-driven simulations.

The remainder of the paper is organized as follows. We first provide a generic definition of high-volume traffic aggregates and related events (§2). We then concentrate on challenges in the aggregate detection motivating a new solution (§3) and discuss its desired properties (§4). We then present ET, our solution (§5), alongside the prototype implementation (§6) and the experimental evaluation (§7). Finally, we cover related works (§8) and conclude the paper (§9).

2 THREE PRIMITIVES, MANY USE-CASES

A broad spectrum of use-cases can be enabled with the detection of (hierarchical) heavy hitters, superspreaders and changes in the traffic patterns. This section provides a generic definition for traffic clusters and discusses how those events are related, while Tab. 1 links them to the specific use-case.

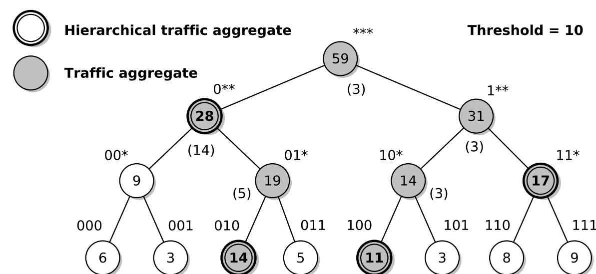


Figure 1: Example of pure (gray nodes) and hierarchical (double circle nodes) traffic aggregates following the generic definition. Each node represents a prefix p with associated amount of traffic.

2.1 High-volume traffic clusters

A high-volume traffic cluster (or aggregate) can be defined as a prefix exceeding a pre-determined threshold in a time window [19]. Assuming the use of the source IP address as a key, the goal of the clusters detection problem is to find the source IP prefixes that contribute with a traffic volume, in terms of bytes, packets or flows, larger than a threshold T during a time interval t. The threshold T can be also specified as a percentage of the total number of inputs. However, in real live monitoring, it is not possible to know the number of inputs in advance, thus the threshold T has to be estimated, e.g., based on the number of inputs during the previous time interval t.

Fig. 1 depicts an example of traffic aggregate prefixes following the previous definition. Each node of the tree represents a prefix p in a reduced 3-bit model domain of IP addresses and its associated amount of traffic. Terminal nodes express only the traffic volume produced by full IP addresses. Non-terminal nodes summarize the traffic of a prefix p. The contribution of each prefix is represented as a number in each node. Considering the use of a threshold $T = 10$, nodes `010`, `100`, `11*` and all their ancestors are identified as high-volume aggregates. For example, each child of the `11*` node contributes independently less than the threshold T, but in total both children contribute enough to exceed the threshold and report the `11*` prefix as an aggregate.

A hierarchical aggregate is a special case of traffic aggregate [19]. It is a prefix p, which exceeds a threshold T after excluding the contribution of all its high-volume descendants[1]. In Fig. 1, only prefixes `010`, `100`, `0**` and `11*` are hierarchical aggregates. The amount of traffic of each aggregate prefix without the impact of its hierarchical descendants is shown in brackets. In this example, the `11*` node is a hierarchical aggregate, as none of its children contributes enough to exceed the threshold T, but the amount of traffic from both children exceeds the threshold. In contrast, the `1**` prefix is not hierarchical because a significant part of its contribution originates from its descendants `100` and `11*`, which are already hierarchical aggregates and must be excluded. It is worth noting that, while the detection of hierarchical aggregates requires the knowledge of pure high-volume traffic clusters, the

[1]The descendant prefixes need to satisfy the definition of high-volume traffic aggregate.

opposite is not true. Reporting the hierarchical aggregates to a controller guarantees minimum overhead, while providing all the necessary information. Taking Fig. 1 as an example, a switch capable of detecting traffic aggregates would export the following prefixes: `0**`, `1**`, `01*`, `10*`, `11*`, `010` and `100`. In contrast, a switch reporting just hierarchical aggregates would provide `0**`, `11*`, `010` and `100`. In both cases the amount of useful information is the same[2], but with hierarchical aggregates we export less data.

2.2 Traffic clusters events

Given the previous definition of high-volume traffic clusters, we show how heavy hitter, change detection and superspreader network events fit into it.

a) Heavy hitter. A heavy hitter (HH) [19] is defined to be a host that sends or receives at least a given number of packets (or bytes) over a short period of time. It is a traffic cluster in terms of packets (or bytes) per second.

b) Change detection. Change detection is the practice of finding which flows contribute the most to the traffic pattern changes over two consecutive time intervals [12]. The method detects traffic anomalies by deriving a model of normal behavior based on the past traffic history and looking for significant changes in short-term behavior that are inconsistent with the model [35]. It is a traffic cluster in terms of packets (or bytes) per second.

c) Superspreader. A superspreader (SS) is defined to be a host that contacts at least a given number of distinct destinations over a short time period. It is a traffic cluster in terms of unique flows per second. In addition, if the same spread detection is applied to the destination, this analysis allows Denial of Service (DDoS) victim detection [54].

3 MOTIVATING A NEW SOLUTION

State-of-the-art solutions that allow the detection of high-volume traffic clusters, periodically export aggregated flow counters to a controller which ultimately is in charge of estimating the metrics of interest [30, 39, 52, 54]. Nevertheless, such an architectural choice requires a careful controller-dataplane coordination.

The reporting time dilemma. *When shall I export my data structure to a central controller?* We ran a first experiment to estimate the importance of setting the correct reporting time. In the context of heavy flow detection, let us assume a flow is indexed only through the packet source IP and the switch has enough memory to keep track of every flow. Let us also assume that the switch exports periodically the counters, and the controller, in charge of the detection, considers heavy the flows that exceed 1% of the total traffic. Finally, let us consider a reporting time of 20 seconds, as suggested by state-of-the-art solutions [39, 48]. We analyzed four one-hour packet traces from CAIDA [9, 10] and we split them into 720 chunks of 20 seconds each. We then computed the heavy flows based on the previous definition. Finally, we decreased the reporting time and we calculated which of the heavy flows could have been detected earlier. Fig. 2 reports the CDF of reported heavy flows varying the reporting time. Interestingly, on average, more than 60% of the heavy flows could have been detected within one second. Note that

[2]Some of the reported high-volume traffic aggregates are just prefixes of more specific hierarchical traffic aggregates.

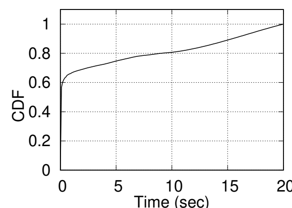


Figure 2: CDF for heavy flow detection time.

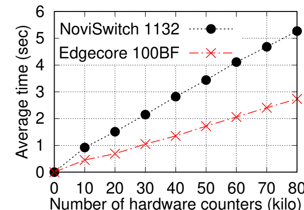


Figure 3: Time to retrieve hardware counters.

the results are based on offline analysis of packet traces only and thus do not contain false positives. This test suggests that, in theory, the reporting time should be set as low as possible. However, in practice, it is important to take into account the switch-controller capabilities of collecting statistics at short time scales, which we quantify in the next experiment.

The cost of statistics collection. *Is it just about exporting the data structure at very short timescales?* We ran an experiment to measure the amount of time it takes to retrieve an increasing number of hardware counters from a switch. We used two different hardware based systems. An OpenFlow-enabled switch, NoviSwitch 1132 [43], which has been designed for use in high bandwidth and flow-intensive network deployments, and a P4-enabled device, the Wedge 100BF-32X, a white box from Edgecore. We connected the switches to a server running a controller and we built an application that allows to request an increasing number of flow counters. The switches were idle when the counters were pulled. Fig. 3 shows the results we obtained. Although the use of probabilistic data structures, i.e., sketches, can help in reducing the number of counters to be exported, past research has shown that from around 60K to 150K counters are still required to provide useful information to a controller [39, 52]. In this context, the NoviSwitch needs at least 5 seconds, and the Edgecore 2.5 seconds. The lesson learned is that retrieving a large amount of data from hardware is time consuming and requires time scales of seconds. This finding has two major implications: (1) given that the statistics retrieval process is performed periodically, the operation needs to be dimensioned with respect to the switch capabilities: the reporting time cannot be lower than the time needed to collect the statistics; (2) in the worst case scenario, a controller can apply appropriate network updates only seconds after a specific event has happened. Therefore, performing traffic analysis in the controller might introduce delays that depending on the specific use-case are not acceptable, e.g., (D)DoS detection. On the other hand, pushing notifications to a controller as soon as an event is detected in the dataplane would allow a reaction in a more timely fashion.

The limited memory access. *Would a push-based sketch work then?* Programmable switches based on match-action architectures, i.e., RMT [8], process packets in a pipeline and for stateful processing (aggregation of flow counters) use a small amount of SRAM that persists across consecutive packets. To guarantee high throughput, the complexity of pipeline stages is limited: this impacts the number of memory accesses. Only one or a few addresses in the memory block can be read or written from a dataplane algorithm, but due

to per-stage timing constraints not the entire memory region [4, 8]. Hence, it is not possible to package the entire counter-based or sketch-based data structure in a single In-Band Network Telemetry (INT) style packet [27, 31]. Such a limitation opens up a specific question:

> *Is it possible to design a data structure, well-suited for a push-based design, that would access only a small memory block and expose a single entry upon the detection of a network event?*

4 DESIRED PROPERTIES

Fig. 4 surveys the design space for the detection of high volume traffic clusters and places our solution, *Elastic Trie*, in the context by following the thick red lines through the design tree. This section describes the insights that inform our major design decisions.

Push-based friendly. Given today's constraints of programmable switches, only a very small memory block can be read at once and sent to a controller with a single digest packet [4, 8]. Thus, to support push-based notifications, the design of the data structure is of paramount importance: it needs to quickly locate the memory address where the prefix to be announced to a controller is stored. Let us take a sketch data structure as an example: in order to find the most populated bucket and send its related information in a digest packet, a program should first scan all the possible entries. This is clearly not optimal. Indeed, current sketch-based architectures work with a poll-based mode [30, 39, 52, 54], where the controller retrieves the whole data structure from the dataplane. In Elastic Trie (ET), we seek for a solution where with limited memory accesses, the dataplane program can find the IP responsible for the traffic cluster and send the related information with a digest packet to a controller.

Optimization for network management. Dividing the time in fixed intervals simplifies the network events detection. At the end of each time window, it is possible to identify the flows that consume more than a fraction T of the link capacity, i.e., heavy hitter, or determine the host that contacts more than a number of unique destinations, i.e., superspreader. For this reason, current solutions for network monitoring typically operate by exporting counters or specific data structures, e.g., sketches, to the controller at fixed time scales [38, 39, 53]. However, this approach tightly bounds the reactive capabilities of the network with the dataplane statistics reporting time, as it needs to be (at least) comparable to traffic variations [3, 6]. If this last condition is met, solutions like dynamic routing of heavy flows [6, 20, 45] or dynamic flow scheduling [47] can be implemented. However, state-of-the-art solutions adopt a fairly large reporting time (typically 20 seconds [39, 48]) to overcome the limitations shown in §3, thus limiting network reaction capabilities. Instead, we propose to start tracking a coarse-grained approximation of the prefix responsible for our supported network events and iteratively refine it over the time. Then, depending on user settings, the controller receives a finer or coarser prefix information with bounded time delay.

Historical network trend awareness. Change detection is the process of identifying flows that contribute the most to traffic

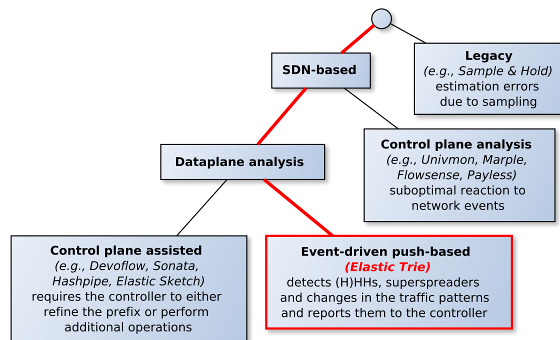


Figure 4: Design space in traffic aggregates detection.

change over two consecutive time intervals [12]. Previous solutions [28, 39] rely on the controller to compute the differences from multiple intervals. With ET instead, we seek a solution capable to directly compute such an operation within the dataplane at the expense of minimal memory consumption. This allows to minimize as much as possible the amount of data to be exported.

5 ELASTIC TRIE

ET enables the detection of network events associated with high-volume traffic clusters from within the dataplane without the need to be coordinated by a controller. It works in a packet-driven manner and can be implemented using match-action based architectures such as RMT.

In this section, we describe how ET works taking HHH detection as an example and show that it can also be used to spot superspreaders and network traffic changes. We then discuss the user interface exposed from an ET-enabled switch to a network operator and show how ET can be used in the context of network-wide monitoring.

5.1 Data Structure & Algorithm

Let us use hierarchical heavy hitter (HHH) detection as an example to explain how ET works. In this scenario, we need to take into account packets (or bytes) to identify an aggregate. When we designed ET, we decided to use a tree-based data structure for the following reasons: (1) IP addresses are naturally organized according to prefixes into a hierarchy and (2) if aggregates are indexed in a tree, then by using standard longest-prefix matching techniques, it is possible to quickly find the small memory block where the prefix is stored, and then create the digest packet.

Each node of the prefix tree (trie) consists of three elements: the counter associated to the left child, the one associated to the right child and a timestamp. The counters represent the amount of traffic, i.e., packets, bytes or flows, for each of the node's direct subprefix, while the sum of the counters represents the amount of traffic sent by the prefix itself. The timestamp specifies the time when the node was created or the last time when the counters were reset. The starting condition is associated to a trie composed by a single node, corresponding with the zero-length prefix $*$. The idea behind the proposed solution is to have a trie that grows or collapses to focus on the prefixes that account for traffic aggregates.

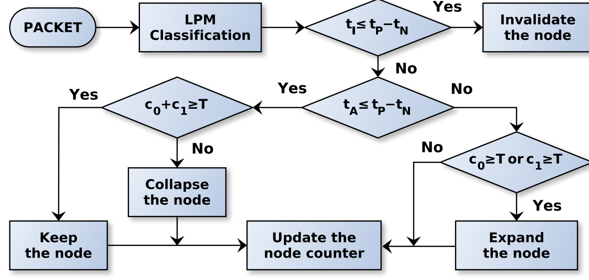


Figure 5: Flowchart showcasing input packet processing of the Elastic Trie detection algorithm.

To achieve this, we defined two timers: *active timeout* t_A and *inactive timeout* t_I, where $t_A < t_I$. The active timeout t_A is the upper bound interval after which the prefix is evaluated and possibly reported as (H)HH to the controller. The inactive timeout t_I defines instead the interval after which the node is considered inactive and its counters outdated. This configuration has the advantage that ET is not limited to a fixed time window and nodes are expanded and invalidated asynchronously: when using sketches, the whole data structure has to be zeroed by the controller at the end of each time window, this is not the case for ET. Fig. 5 depicts the key steps of ET algorithm. For every incoming packet, the LPM (corresponding node) is looked up. Let us denote by c_0 and c_1 the left and the right child counters of the node. The node timestamp (t_N) is compared against the packet arrival timestamp (t_P). Based on the comparison, node counter values and timeouts t_A, t_I, there are five possible cases to be considered:

Invalidating the node. If the inactive timeout t_I has expired ($t_I \leq t_P - t_N$), then the node has been inactive for a long time. The values of the counters are outdated and not relevant anymore for the detection. This happens when the source stops sending packets for a while. Because the detection process is built on a packet-driven basis, such situation cannot be evaluated in a different way. The inactive timeout mechanism handles the situation when the packets belonging to a source prefix start to flow again and the old values must be invalidated. Fig 6a illustrates this case. Regardless of the counter values, the tree node is simply removed and the counter values discarded.

Expanding the node. This is the case when both the active and inactive timeouts have not expired ($t_P - t_N < t_A < t_I$), but one of the counters (let us assume, for example, c_0) exceeds the threshold T that is used to discriminate heavy prefixes ($c_0 \geq T$). In this case, the subprefix associated with c_0 is (optionally) reported to the controller as HH but not as HHH yet. Fig. 6b depicts this case: the data structure automatically starts the refinement of the prefix (`10*`) by creating a new child (`100`) corresponding to c_0. According to the definition of hierarchical cluster, the original c_0 must be set to zero to remove the contribution of the newly created descendant. Since, we do not have any records for the newly created child yet, its node will have the timestamp set to the current packet timestamp and both its counters set to zero.

Keeping the node. This is the case when the inactive timeout t_I has not expired, but the active timeout t_A has ($t_A \leq t_P - t_N < t_I$), and the sum of counters exceeds the threshold T ($c_0 + c_1 \geq T$), but none of the counters contributes enough to reach the threshold individually ($c_0 < T$; $c_1 < T$). The case is shown in Fig. 6c. The prefix `11*` is a HHH, because it exceeds the threshold T and none of its children exceed to threshold individually. The node is reported to the controller, its timestamp updated with the packet timestamp value and the counters are reset.

Collapsing the node. If the inactive timeout t_I has not expired, the active timeout t_A has expired ($t_A \leq t_P - t_N < t_I$) and the sum of counters does not exceed the threshold T ($c_0 + c_1 < T$), the node is collapsed (Fig. 6d). The node (`10*`) is removed from the structure, and it is replaced by the nearest parent. The counters of the parent node (`1**`) are zeroed and the timestamp set to the current packet timestamp. This is in contrast with the node invalidation case, where the nearest parent is not reinserted or renewed. Note that collapsing a node can happen only when the node has either none or one child because a node with two children does not match LPM.

Updating the node counter. This is performed when both the active and inactive timeouts have not expired ($t_P - t_N < t_A < t_I$) and none of the counters exceed the threshold T ($c_0 < T$; $c_1 < T$). In this scenario, the counter corresponding to the packet subprefix is updated and the trie structure does not change. Note the counter is also updated after other actions when the node is kept, expanded or collapsed. In these cases the newly created node or the nearest parent counters are updated instead of the current node counter.

5.2 Elastic Trie with the other events

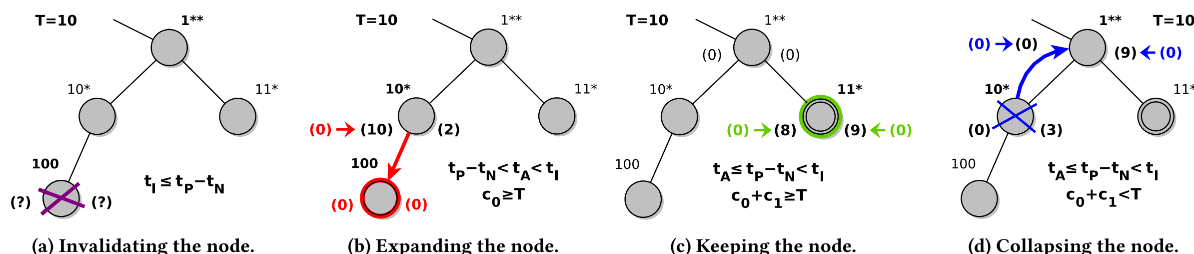
In this section we show how ET supports also the detection of superspreaders (DDoS victims) and network traffic changes.

Superspreaders detection. As introduced in §2, SS is a host that contacts at least a given number of distinct targets. Thus, to enable such a detection, it is important to keep track of the number of destinations contacted by each source prefix. To address this challenge we used a standard Bloom filter [7], a memory-efficient probabilistic data structure commonly used to test for set membership. Specifically, we deployed the filter to test if a packet belongs to a new unique flow or not. The key to index the filter consists of the source IP prefix looked up during LPM classification phase and destination IP address of the packet.

The control logic that adjust the hierarchical structure is the same, but a test-and-set operation on the filter is performed for each incoming packet and the node counters are updated only if a new unique flow is detected.

Change detection. Changes in the traffic patterns can be detected by modeling the normal traffic behavior based on the past history and looking for significant changes that are inconsistent with the model itself. By tracking the number of nodes expanded or collapsed over an active timeout interval t_A, it is possible to spot sudden changes.

We added an integer counter which is incremented and decremented when any node of the tree is expanded or collapsed, respectively. When the traffic is steady, the number of nodes expanded and collapsed should be similar. Thus the counter value should vary around zero. When it is not the case, significant changes in the short-term traffic behavior have happened and are promptly reported to a controller.



(a) Invalidating the node. (b) Expanding the node. (c) Keeping the node. (d) Collapsing the node.

Figure 6: The core cases of Elastic Trie refinement, assuming the threshold $T = 10$. Each node represents a prefix and builds the data structure. Node counters are shown in brackets on the sides.

Limitations. A single ET instance cannot track HHH and SS at the same time. To detect both events, it is necessary to deploy two individual structures side-by-side. While the HHH detection needs to use counters to track the number of packets, the SS detection tracks unique flows. On the other hand, the change detection can be freely combined with both packet or flow aggregates and built independently on top of HHH or SS detection trie.

5.3 User Interface

ET can detect the network events discussed in this paper without the need of any specific query from the operator. When an event is detected, a digest message is sent from the switch to the controller. It is still possible though to alter the behavior of the dataplane by changing T and t: recall that a traffic cluster is defined as the IP prefixes that contribute with a traffic volume, in terms of either bytes, packets or flows, larger than a threshold T during a time interval t.

Level of aggregation. The first parameter is the threshold (T). It directly impacts the number of reported prefixes and the memory requirements of the data structure, as shown in the evaluation section (§7): the higher the threshold, the less prefixes will be reported – they aggregate more traffic.

Tree building speed. The second parameter is the time interval (t). This allows the operator to effectively control the speed at which the tree is built and events reported. As shown in the evaluation section (§7), low values negatively affect the memory requirements of the data structure but allow the operator to react in a more timely fashion. On the other hand, high values allow ignoring transient events. As described in detail in the algorithm section (§5.1), we, introduce two time intervals parameters: *active timeout* t_A and *inactive timeout* t_I to distinguish between node reporting and invalidation time. In §5.4 we further define variable active timeout mechanism which beneficially allows setting different intervals for different tree levels.

5.4 Discussion

In this section we discuss an optimization that accelerates the trie building mechanism, thus speeding up the detection process, and show how ET could be used to perform network-wide monitoring operations.

Variable active timeout. The starting condition for the structure is associated to a trie composed by the zero-length prefix $*$. Depending on the packet flow, the trie is then built to focus on the heavy prefixes. Although the refinement process, as explained in §5.1, does not depend on the selected active timeout, the process of deciding if a specific prefix is a traffic aggregate and the potential reporting to the controller does. This means that in the worst case scenario a full IP address is reported after $32 \times t_A$ seconds: the upper bound for building the tree from the root to the lowest level. To mitigate this, we propose a variable active timeout mechanism which sets different intervals and corresponding thresholds for different prefix lengths, i.e., smaller timeout and threshold for shorter prefixes and vice versa.

Network-wide monitoring. The digests received by different ET-enabled switches can be used by a central controller to perform network-wide traffic analysis. Besides the obvious network-wide heavy hitter detection [29] use case, which is inherently possible by setting the appropriate threshold T in the switches and aggregating the received notifications in the control plane, others are also feasible. Specifically, the superspreader notifications received by different switches can be used by a central controller to perform the degree histogram estimation [46], commonly adopted practice to detect botnets involved in coordinated scans [26]. Furthermore, the controller can leverage the (H)HH primitive with an appropriate threshold T to detect global network icebergs [57]. Those are particularly difficult to detect within a single system, as the responsible packets might come from a large number of hosts and thus traverse different paths. We leave the evaluation of our system in a network-wide context to future work.

6 IMPLEMENTATION

This section discusses the implementation of ET on programmable hardware, using FPGA and the $P4_{16}$ language [18].

6.1 P4 prototype

Fig. 7 depicts a high-level view of the architecture and illustrates the operations performed for each incoming packet. The structure is organized around three main building blocks: (A) the LPM classification stage, (B) the main memory and bloom filter used to gather traffic statistics alongside related timestamps and (C) the control logic to dynamically adjust the hierarchical data structure and to report results of the detection to an external controller.

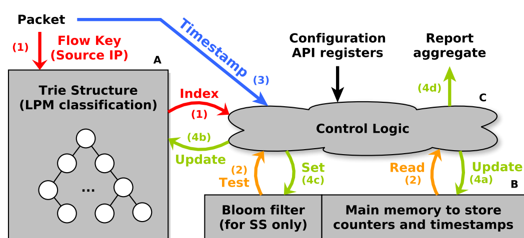


Figure 7: Elastic Trie dataplane architecture.

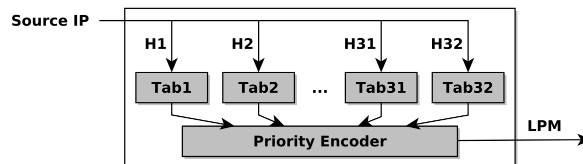


Figure 8: The LPM classification stage in P4.

Each incoming packet is first parsed to extract the desired flow key, i.e., source IP[3]. Then, the hierarchical tree structure is accessed to find the LPM (step 1). The result of this stage is an index that is used to access the main memory, where the data structure of the associated node is stored (step 2). Optionally for SS also an item presence is tested in the Bloom filter. The read values are compared with the packet timestamp (step 3) and the user settings, i.e., T, t. The appropriate operation is computed (step 4) following the specifications described in the previous section. Specifically, the comparison can trigger an update of the main memory (step 4a), an update of the LPM classification scheme (step 4b), setting an item in the Bloom filter (step 4c) or a push notification to the controller (step 4d). In the following, we provide a more detailed description of the mapping between the three main building blocks and $P4_{16}$ match-action constructs.

LPM classification stage. Although P4 offers built-in match tables supporting LPM, we could not utilize them for implementation of the trie structure, since the latest P4 specification does not support modifications of these tables directly from the dataplane, even though some targets like FPGAs may support it. As this feature is essential for our architecture, we opted for a custom LPM implementation. We used a hash table for each prefix length (Fig. 8), thus requiring 32 hash tables to support each IPv4 prefix[4]. Each hash table is implemented as a register array. Upon packet arrival, all the hash tables are read in parallel, by hashing the associated prefix of the flow key. We use `hash` extern API with `CRC32` as an algorithm to generate hash values to access the registers. Hash tables referring to short prefix values usually require less memory, as they need to store information for a smaller number of results. Thus, depending on the amount of memory preallocated to each hash table, we use a direct access based only on the prefix value itself (the `IDENTITY` algorithm in P4 API) for some of the shortest prefix tables. Each hash table lookup result can then be represented as a single bit value, `1` (found), `0` (not found) respectively. We then concatenate these bits to form a bitvector, which serves as input key for a static ternary match table implementing a priority encoder.

Main memory access mechanic stage. The hash value of the resulting LPM is used as address to access a register array that stores the required node structure information for that specific prefix, i.e., two packet counters and a timestamp. We use 32-bit wide packet counters and 48-bit wide timestamp as it is available in the packet metadata structure in P4. To detect hash collisions in our implementation of LPM classification stage in P4, we further extended the node data structure with a up to 32-bit wide flow key field (IPv4 prefix). Note that we do not need to store the prefix length because we use a separate hash table for each length. Thus, the size of each node structure is 144 bits (112 bits for the node and 32 bits for the IP address). In the case of a hash collision, the nearest shorter prefix node is used. The Bloom filter can be also easily implemented in P4 as a combination of a register-based bit array and a set of hash functions.

Control logic. This stage compares the packet timestamp with the node timestamp and applies the logic described in §5.1. The node collapse or expansion is performed by updating the appropriate hash table storing the specific prefix that needs to be adjusted, while the push-based mechanic is implemented by generating a packet digest (`digest` extern object in P4 API) containing the IP prefix detected as traffic aggregate alongside its node information such as the sum of the counters and the timestamp.

6.2 Potential limitations and challenges

Although we successfully implemented ET in P4 and compiled the code in the behavioral model [17], this does not guarantee ET being efficiently implementable in any high-speed P4 target available today. As briefly mentioned in §3, the main constraint of programmable switches is the number and the pattern of accesses to on-chip registers. Unfortunately, at the time of working on the paper, we did not have access to one to provide a full implementation on an actual hardware target. This section thus discusses the adopted pattern of accesses to on-chip registers, the potential limitations of the ET data structure and challenges in porting our code to commercially available P4-enabled switches.

The core part of the ET algorithm uses a register array to store the nodes of the tree. In the worst-case scenario, ET requires at most only two memory reads and writes (reading and updating a single node and optionally reading and updating its parent or child node). This could be a problem if the target does not support multiple accesses into the register array. However, the array can be eventually split into two parts, to store nodes in even and odd levels of the tree separately. This trick can guarantee a single read/write pattern per packet.

Another specific drawback of the ET algorithm is the need for a custom LPM implementation, which requires 32 hash tables to support each prefix length. Hash tables are implemented as a single register array requiring 32 reads and at most one write (inserting or removing a node in the tree), in contrast to the tree nodes, each

[3] While Elastic Trie is oblivious to the specific packet field used as flow key, the source IP address is commonly used for SS and (H)HH detection.

[4] Using less hash tables and supporting only a subset of prefixes comes at the cost of node complexity. Indeed, each node needs to store a counter for each associated subprefix. This means that if we use only hash tables for just the prefixes \8, \16, \24 and \32, we need to construct nodes with 256 counters each.

Chip	LUTs		Regs	Frequency
	Logic	Memory		
Virtex 7	11 088	2 880	14 104	172.4 MHz
Virtex US+	9 135	2 641	14 103	307.9 MHz

Table 2: HW resources used and frequency achieved by Elastic Trie FPGA implementation.

hash tables stores only a single bit information. Using the same trick the array can be optionally split to have a register for each table. If a P4 target enables only a limited number of independent reads in a single stage, hash table registers can be eventually separated into more consecutive stages.

Finally, the custom LPM implementation forces spreading of the ET algorithm across multiple pipeline stages and thus introduces undesirable read/write dependencies, e.g., the memory index acquired from the LPM hash tables is used to access the node information which can result back in updating the LPM hash tables in case of a node collapse or expansion. In fact, the LPM hash table registers need to be accessed from the first stage to acquire the index, but also from the second stage to write the updates. However, this could potentially be a challenge for some of today's P4 targets if a set of registers is strictly bound to a single stage, effectively forcing the compiler to put all the registers in a single stage.

6.3 FPGA prototype

To demonstrate the general feasibility of implementing ET data structure on existing programmable targets and to quantify its requirements in terms of hardware resources, we at least created a pure VHDL implementation of ET for two different FPGAs, i.e., Xilinx Virtex 7 (model XC7VH580T) and Virtex UltraScale+ (model XCVU7P). We merged the LPM stage and main memory records into one memory block and utilized distributed memory to implement it as 32 parallel hash tables. Tab. 2 shows the chip occupancy and frequency of our design for both platforms. The latency introduced by the design is 7 clock cycles. Considering the achieved frequency, the latency is 40.6 ns for Virtex 7 and 22.75 ns for UltraScale+. The architecture is capable to process a new packet every 4 clock cycles (the first 3 out of 7 stages are pipelined). This results in an overall throughput of 43.10 Mpps for Virtex 7, and almost twice as much, 76.97 Mpps, for the UltraScale+ platform.

7 EVALUATION

Following a common practice adopted in evaluating P4 enabled solutions [39, 48, 52], we implemented a C++ simulation model to assess our approach against real traffic traces from an ISP backbone network. Additionally, the two FPGA prototypes (§6.3) provide insights about expected performance on real hardware and by compiling the P4 code in the behavioral model [17], we verified its correctness. In this section, we first describe our setup and we evaluate the trade-offs of ET. Then, we discuss its detection accuracy against the supported network events (Tab. 3 summarizes them all). We also evaluate ET varying memory occupancy and data structure configuration and compare it with state-of-the-art solutions. Finally, we analyze the controller-dataplane communication overheads and our detection speed.

Traces. We used four different one-hour packet traces from CAIDA [9, 10] recorded from 10 Gbps backbone links in San Jose and Chicago in 2009 and 2016 (both directions A and B). Each of the trace contains between 1.6 and 2.4 billion packets with mean transmission rates in range 440k-640k pkts/s (2.3-3.9 Gb/s) and flow rates up to 61k flows/s. The traces are distributed in one minute chunks and each chunk contains on average 30M packets with around 840K unique IP addresses. For further and detailed statistics for the individual CAIDA traces we refer to [11]. Unless otherwise stated, all the following results in the section are indicated as an average evaluated over the continuously replayed chunks of all four CAIDA traces. Unfortunately, we could not use the newer datacenter traces from the Facebook Network Analytics Data Sharing program [1], as they are sampled. Other publicly available datacenter traces from 2009 [5] have been anonymized without prefixes being preserved, which also makes them inappropriate for the type of tests needed in this paper.

Setup. We first set the primary measurement reporting time (active timeout t_A in our case) to 20 seconds and the inactive timeout t_I to 5 minutes. Then, when assessing the variable active timeout behavior (discussed in §5.4), we set it differently for each trie level. Specifically, we used an exponential function that specifies the value of the timeout for each of the trie level: the closer the node to the root, the lower the timeout. This allows to have much smaller timeouts for shorter prefixes, thus enabling a quicker tree build. In fact, the refinement process does not directly depend on the selected active timeout. It can be better understood as an upper bound for the delay between two reports, especially when there are no changes in detected aggregates. During the refinement process new aggregates are always reported immediately when the threshold is exceeded, thus the real reporting time is effectively much smaller and proportional to the rate of the threshold being reached. The threshold T, used to discriminate the prefixes that are "large enough", has been set to be 1%, 5% and 10% of the maximum amount of traffic (packets or flows).

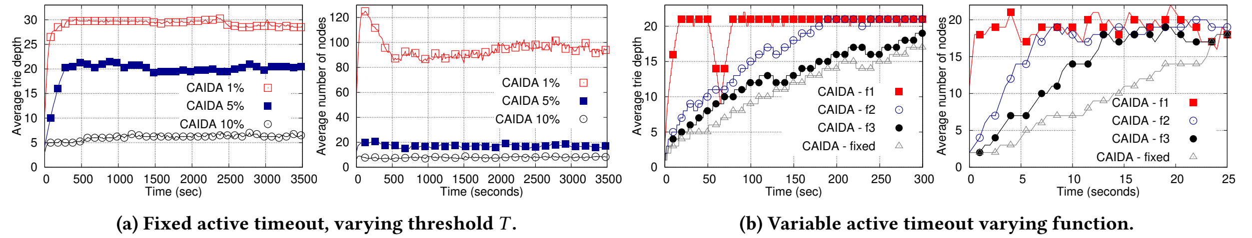
Metrics. We evaluated the number of required nodes and the trie depth varying the configuration parameters. Then, to estimate its network event detection capabilities, we used the F_1 *score* metric [52]. Assuming T_P (*true positives*), F_P (*false positives*) and F_N (*false negatives*), $F_1 = 2T_P/(2T_P + F_P + F_N)$. Unless otherwise stated, the F_1 score is always indicated as the average over the chunks of the traces.

Prefix comparison. We define two ways assessing ET detection capabilities: (1) can ET report the exact prefix? (2) can ET report at least a 2 bit shorter (coarser grained) version of the prefix? As by construction, ET starts reporting an approximation of the responsible prefix and iteratively refine it over time, we believe this is a good metric to better grasp the trade-off between fast detection and accuracy.

Memory allocation. ET has been architected to be implemented in hardware where allocation or de-allocation of memory at runtime is not possible. Thus, we statically pre-allocate a specific amount of memory for our data structure. An invalidation or collapsing of nodes is mostly used to track the traffic changes, not to manage the

Network event	Event definition	Implementation using Elastic Trie	Management tasks
(Hierarchical) Heavy Hitters	Identify hosts/prefixes which contribute with a traffic volume more than a defined threshold T during a time interval.	Node counters to count volume of specific prefixes. Expand and keep node events to identify exceeding the threshold T.	accounting, traffic engineering
Superspreaders	A superspreader is a host that contacts at least a given number of distinct destinations over a short time period (spread detection applied to source hosts).	Bloom filter to identify distinct destinations. Node counters to count distinct destinations. Expand and keep node events to identify source prefixes exceeding a predefined threshold.	scans, spread of malware detection
DDoS victims	A DDoS victim is a host that is contacted at least by a given number of distinct sources (spread detection applied to destination hosts).	Bloom filter to identify distinct sources. Node counters to count distinct sources. Expand and keep node events to identify destination prefixes exceeding a predefined threshold.	DDoS detection
Changes in traffic patterns	Identify hosts/prefixes which contribute the most to the traffic changes over the last time interval.	Tracking a difference of number of expanded and collapsed nodes to detect this event. Expand, collapse node events to identify specific prefixes/hosts involved.	anomaly detection, DoS detection

Table 3: Implementation of different network events detections using Elastic Trie algorithm.



(a) Fixed active timeout, varying threshold T. **(b) Variable active timeout varying function.**

Figure 9: Trie depth and number of nodes varying threshold and timeout behavior.

memory. The lack of memory, thus, results only in worse detection accuracy due to more hash collisions.

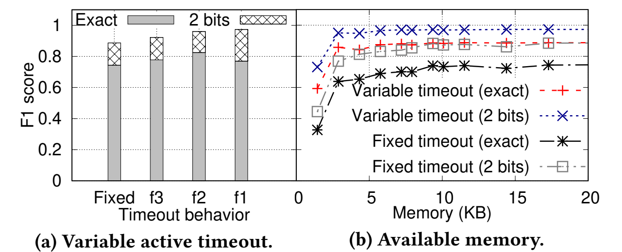
7.1 Data structure properties

Fig. 9a shows ET's average depth and average number of nodes over time for CAIDA traces varying the threshold. The threshold T was set to 1%, 5% and 10% of the amount of traffic in terms of packets. The depth and number of nodes are as expected proportional to the selected threshold: the lower the threshold, the larger the depth and the number of nodes, since more prefixes are detected as heavy. It is also possible to see the learning phase of the trie at the beginning of the trace, when the trie has to build up from the less specific prefix. After this phase, the trie reaches a steady state that reflects the current traffic behavior. Fig. 9b offers, for the threshold of 5%, a more detailed view on the learning phase and compare the impact of variable active timeout with different exponential functions. Using a variable active timeout mechanism, we can speed up the learning phase by 93%, going from 300 seconds needed for fixed timeout to 20 seconds needed using the most aggressive function f_1. In contrast, aggressive functions are much more sensitive to traffic patterns, resulting in potential fluctuations of the trie.

7.2 HHH detection

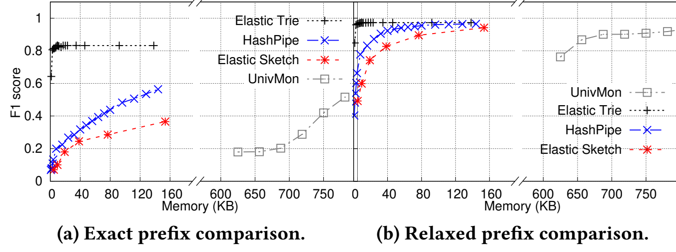
In this section we first present the HHH detection capabilities without resource constraints, then our implementation driven results. The former does not take into account the impact of implementation details such as amount of available memory or potential hash collisions. This allows us to get an understanding of the behavior of our solution in the best case scenario. The latter takes into account limitations in memory availability, as well as hash collisions that might happen during the classification stage. This allows us to get an understanding of the trade-offs between memory and detection results.

Results without resource constraints. Fig. 10a shows the HHH detection capabilities. We used a threshold of 5% and generated the results using both exact and relaxed prefix comparison. Since the basic behavior of ET is to build a trie that focuses on the prefixes that account for a large share of the traffic, sometimes it might happen that due to a transient event the system does not have enough time to finalize the building process and to fully identify the responsible traffic aggregate. However, reporting a partial and approximate result can still be very useful for the network operator. Indeed, the figure shows that the accuracy with exact prefix detection is significantly lower than its only 2 bit coarser grained prefix version. The effect of variable active timeout can also be seen in Fig. 10a. When using a more aggressive variable timeout the F_1 score increases, due to a smaller false negative rate. In contrast, the precision decreases causing a higher false positives rate. This is a direct effect of smaller active timeouts that lead the system to detect more prefixes including the previously mentioned short-term



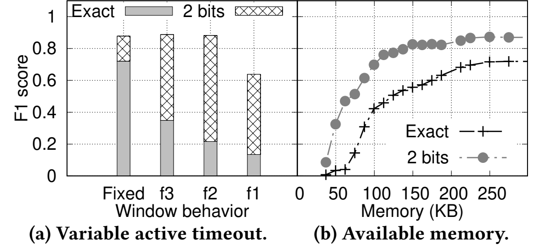
(a) Variable active timeout. **(b) Available memory.**

Figure 10: HHH detection capabilities.



(a) Exact prefix comparison. (b) Relaxed prefix comparison.

Figure 11: Comparison between Elastic Trie, HashPipe, Elastic Sketch and UnivMon of (H)HH detection capabilities.



(a) Variable active timeout. (b) Available memory.

Figure 12: Superspreader detection capabilities.

aggregates. In this scenario, the F_1 score is always between 0.9 and 1.0. Using different functions for variable active timeout, it is then possible to fine tune the trade-off between recall and precision to maximize the F_1 score metric.

Implementation driven results. We assess the impact of the amount of available memory over the F_1 score. We find that our solution can successfully detect, with approximately 0.75 F_1 score, the exact HHH prefix using a fixed active timeout and less than 20KB (Fig. 10b). If a coarser grained prefix is accepted, which is less precise by only two bits, then the F_1 score jumps to 0.9. Again, this is the consequence of the nature of the data structure: it might happen that it does not have enough time to build properly. When using a variable timeout (Fig. 10b), the results improve sensibly. In this case, it is possible to detect the exact HHH prefix with 0.85 F_1 score with less than 8KB. Moreover, if a 2 bit coarser grained HHH prefix is accepted, the F_1 score jumps to 0.98. Increasing the available memory does not significantly improve the detection capabilities of the system, because it is bounded by the ability of the trie to react and build up according to the input traffic patterns.

In Fig. 11, we compare the HHH detection capabilities of ET against state-of-the-art solutions: UnivMon [39], Elastic Sketch [52] and HashPipe [48]. All of them are able to detect HH only as full length prefixes (addresses). Moreover, UnivMon and HashPipe use an alternative definition for HH detection, named the "top-k problem". Instead of reporting prefixes that are larger than a given threshold, they report the top-k sources, no matter the amount of traffic they are actually sending. To perform a fair comparison, and align their results with the one produced by our system (which follows the classic HHH definition), we aggregated their output addresses into prefixes and considered only the ones that carry traffic above the fixed threshold T.

Fig. 11a shows the results using exact prefix comparison. To reach a F_1 score around 0.5-0.6, HashPipe needs a lower amount of memory (~144KB) than Elastic Sketch (~320KB), which is still much lower than the amount needed by UnivMon (~800KB). In contrast, ET significantly outperforms other solutions. This is also confirmed by the results obtained when a coarser grained prefix is permitted (Fig. 11b). Nevertheless, the memory requirements of the four solutions represent a fair comparison metric. HashPipe, Elastic Sketch and ET have similar memory requirements, but HashPipe can only detect Heavy Hitters, while our solution enables, at the same time, also traffic pattern changes detection. UnivMon is not restricted to a single network event, but requires 90% more memory to work. Finally, Elastic Sketch in its heavy flows mechanism ignores mice flows. In contrast, ET can, thanks to the adopted trie-based data structure, aggregate these flows into shorter prefixes, which results in more accurate HHH detection.

7.3 Superspreader detection

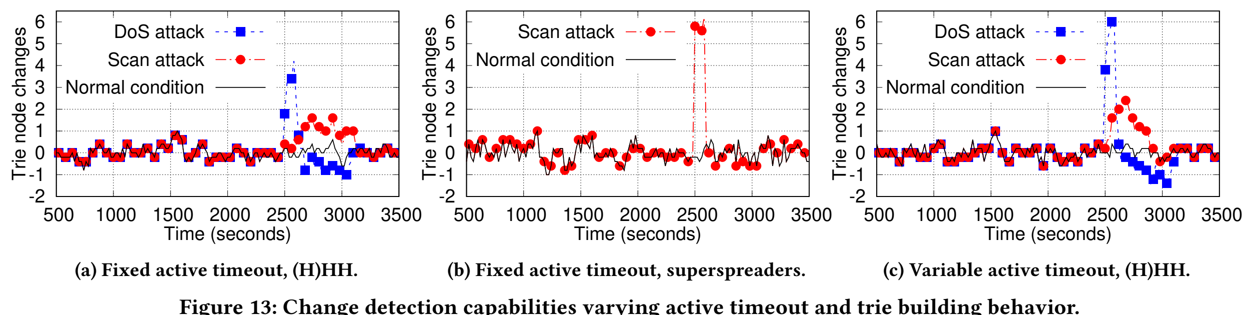
As in the HHH case, we first introduce the results without taking into account available memory or hash collisions. Then, we show the trade-offs between available memory and superspreader detection capabilities.

Results without resource constraints. Fig. 12a shows the superspreader detection capabilities without the impact of available memory or hash collisions using CAIDA traces and varying the active timeout behavior. In contrast to the same evaluation carried for the HHH detection, the results show that the system performs better using a fixed timeout. In this case, it is clear that the trie cannot build in time, as F_1 score grows sensibly when we use a 2 bit coarser grained superspreader prefix. Overall, for fixed active timeout the detection capabilities are still good, as F_1 is around 0.9.

Implementation driven results. In Fig. 12b we show the impact of available memory over the detection capabilities, taking into account our P4 implementation. For this test, we used a fixed active timeout, 25KB of preallocated memory for prefix trie structure, and we varied the amount of memory available for the Bloom filter. We find that superspreaders can be successfully detected with an F_1 score of approximately between 0.72 and 0.87 with less than 250KB of allocated memory. Among the considered solutions already available in the literature, i.e., UnivMon, Elastic Sketch and HashPipe, only the former theoretically supports superspreader detection. However, its current implementation does not allow to reproduce such a test. We were thus unable to compare it against our system.

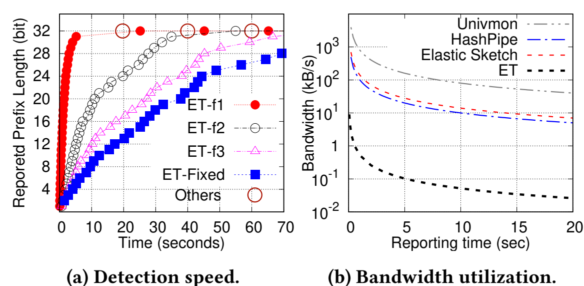
7.4 Change detection

To demonstrate the traffic change detection capabilities of ET, we artificially injected network traffic simulating a sudden heavy flow and a scanning into one of the CAIDA traces. The attack has been emulated after 2500 seconds since the beginning of the trace. A sudden HH and scanning are two types of attacks that can potentially change traffic patterns. At the same time, they are also pretty different: while a HH is typically a source that sends a huge amount of traffic to a designated destination, the scan is a source contacting many random destinations. Fig. 13a shows the time on the x-axis and a moving average of trie changes (difference between number



(a) Fixed active timeout, (H)HH. (b) Fixed active timeout, superspreaders. (c) Variable active timeout, (H)HH.

Figure 13: Change detection capabilities varying active timeout and trie building behavior.



(a) Detection speed. (b) Bandwidth utilization.

Figure 14: Comparison of detection speed and bandwidth utilization between ET and other approaches.

of expanded and collapsed nodes) on the y-axis. Note that tree is built based on HHH detection using a fixed active timeout of $t_A = 20$ seconds. In the figure we can distinctly see differences during normal conditions and the state under DoS attack or scan. In Fig. 13b, we repeated the same test but from a different perspective. Now the trie is built on top of the SS detection. In this case, the HH is not detected at all, because it represents a communication with only one distinct destination. On the other hand, the scan, as a typical case of superspreader, is much more significant now. Finally in Fig. 13c we run again the same test and build the trie for HHH detection using the variable active timeout mechanism. Due to the accelerated trie construction, there are many more changes in the trie over a short time period. This allows to highlight small changes in the traffic patterns, as shown when comparing the scan behavior for Fig. 13a and Fig. 13c.

7.5 Controller-dataplane communication

In this section, we asses how fast our data structure can provide useful results to an external controller alongside the involved overheads. We used the same setup as in the previous change detection case, and injected a sudden HH into a CAIDA trace. However, this time we focused on the events reported to the controller, especially on the delay to detect the first coarse-grained approximation of the prefix, and later the final prefix responsible for the attack. Fig. 14a shows the prefix length reported on y-axis with respect to the time from the beginning of the HH on x-axis. It can be seen that immediately after the start of the HH the coarse grained prefix is reported and then continuously refined over time. The figure also shows the effect of a variable active timeout. When using the most aggressive variable timeout function, the final responsible prefix is detected in less than 5 seconds, and then reported regularly every 20 seconds. The active timeout parameter can thus be understood as an upper bound for the delay between two reports. If there is no change detected when the timeout expires, the current prefix will be reported again. In the figure, we also include the reports generated periodically by state-of-the-art solutions such as Univmon, Elastic Sketch and HashPipe. Although, an external controller can be instructed to retrieve those data structures at shorter timescales, it is important to dimension the statistic retrieval process coherently with the actual time needed to get the hardware information (as shown in Fig. 3). Specifically, in the case of Univmon, Elastic Sketch and HashPipe, consecutive requests cannot be lower than the time needed to retrieve the data from the hardware. This effectively creates a lower bound in the detection speed for those systems. The same does not apply for ET, as it does not need the external controller to retrieve the full data structure to detect a certain event.

Finally, Fig. 14b compares the amount of data exchanged between a single switch and an external controller when either ET, Univmon, Elastic Sketch and HashPipe are in place. Varying the reporting time window (x-axis), we calculated the required bandwidth assuming 800KB size for the Univmon data structure, 140KB for the Elastic Sketch data structure and 100KB for the HashPipe data structure (default values according to respective papers). For ET we set 12B as the size of a single prefix report, which is enough to report the prefix, the length and also the associated sum of counters and timestamp. We then calculated the average number of HHH reports per second from CAIDA traces running ET with a threshold of 5 %. The figure shows that in comparison to the other solutions, ET can save a significant amount of control plane traffic (more than two orders of magnitude).

8 RELATED WORK

In the past many SDN-based monitoring solutions which rely on OF-based statistics retrieval from switches have been proposed [15, 49, 53]. They suffer from important limitations: (1) coupling between forwarding and monitoring rules, (2) the controller needs to know in advance which flows have to be monitored in the dataplane and (3) as the dataplane exposes just simple counters, the controller needs to do all the processing to determine the network state. In contrast, our solution reports to the controller the network events of interest as soon as they happen, without the need of central coordination. To speed up the identification process, iterative

refinement of the traffic of interest is done directly in the dataplane, to avoid expensive control plane interactions, contrary to solutions in [32, 41, 56]. Although algorithms that use iterative refinement of flows to determine heavy hitters [55] and anomalies [34] have been proposed in the past, they were not targeted for match-action type architectures. Dynamic trie-based data structures have been used for many years [19, 23, 33]. However, in contrast to them, our solution is time-based and hardware-friendly. The main difference is the way that nodes are expanded and invalidated based on stored timestamps, so the event reports are not limited to periodic time windows.

More recently, a number of monitoring frameworks leveraging P4 programmability have been proposed [30, 38, 39, 44, 48, 52]. FlowRadar [38] keeps track of all the flows in the network, their associated counters, and exports this information periodically to a remote collector, which ultimately uses them for various monitoring applications targeted to datacenters. In contrast, our aim is to offload as much as possible the controller, by directly exporting processed traffic information. UnivMon [39], ElasticSketch [52] and SketchLearn [30] use sketch-based data structures in the dataplane to record network traffic statistics and export them at fixed time intervals to the control plane which is in charge to perform a number of measurement tasks. Although some of these solutions apply optimizations to compress as much as possible the data structure, as demonstrated in §3, the interaction between control and dataplane can be very expensive and imply delays that are not acceptable. HashPipe [48] determines the top-k heavy hitters in the dataplane, while Popescu et a. [44] presents a solution for hierarchical heavy hitters detection. They both cover one single measurement task, while our solution is more generic. Finally, Sonata [28] proposes a query interface for network telemetry, uses sketches in the dataplane, and zooms-out the network traffic of interest by refining the network query, starting from the finest level. The refinement is done by the controller, while in our case, directly in the dataplane.

9 CONCLUSION

We proposed a push-based approach to network monitoring, where the dataplane informs the control plane only when specific conditions are met. To achieve this, we presented a new data structure, Elastic Trie, that enables the detection of traffic pattern changes and either (hierarchical) heavy hitters or superspreaders within the dataplane. Our solution has been designed with the constraints of emerging programmable switches in mind, as it works in a packet-driven manner, and can be implemented using common match-action based architectures such as RMT.

Elastic Trie uses a hash table based prefix tree that grows or collapses to focus only on the prefixes that account for a "large enough" share of the traffic. This enables the detection of either (hierarchical) heavy hitters or superspreaders, and at the same time by looking at its growing rate it is possible to identify changes in the traffic patterns. We prototype our solution in P4 and for two different FPGA targets. From our FPGA implementations we provide information about expected performance on real hardware and from our C++ model we show that Elastic Trie achieves high accuracy in detecting the targeted events with the memory constraints imposed by today's switches.

ACKNOWLEDGMENTS

We thank our shepherd Mina Tahmasbi Arashloo and Ran Ben Basat, Theophilus Benson and anonymous reviewers for their valuable comments that helped us to improve the paper. This work was supported by the National Programme of Sustainability (NPU II), project IT4Innovations excellence in science – LQ1602, by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 co-funded by the Ministry of Education, Youth and Sports of the Czech Republic and European Regional Development Fund and by the UK's Engineering and Physical Sciences Research Council (EPSRC) under the EARL project (EP/P025374/1).

REFERENCES

[1] 2018. Data Sharing on traffic pattern inside Facebook's datacenter network. https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/.
[2] 2018. sFlow. http://www.sflow.org/about/index.php.
[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX.
[4] Ran B. Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *International Conference on Network Protocols (ICNP)*. IEEE.
[5] Theophilus Benson. 2010. Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.
[6] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *COnference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.
[7] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. In *Communications of the ACM (CACM), Volume: 13, Issue: 7*. ACM.
[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[9] CAIDA. 2018. Anonymized Internet Traces 2009 – 17th September 2009. http://www.caida.org/data/passive/passive_2009_dataset.xml.
[10] CAIDA. 2018. Anonymized Internet Traces 2016 – 17th March 2016. http://www.caida.org/data/passive/passive_2016_dataset.xml.
[11] CAIDA. 2020. Statistical information for Anonymized Internet Traces. http://www.caida.org/data/passive/passive_trace_statistics.xml.
[12] Christian Callegari, Stefano Giordano, Michele Pagano, and Teresa Pepe. 2012. Detecting Anomalies in Backbone Network Traffic: A Performance Comparison Among Several Change Detection Methods. *International Journal of Sensor Networks, Volume: 11, Issue: 4*.
[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Workshop on Self-Driving Networks (SelfDN)*. ACM.
[14] Baek-Young Choi, Jaesung Park, and Zhi-li Zhang. 2003. Adaptive random sampling for traffic load measurement. In *International Conference on Communications (ICC)*. IEEE.
[15] Shihabur R. Chowdhury, Md. Faizul Bari, Reaz Ahmed, and Raouf Boutaba. 2014. PayLess: A low cost network monitoring framework for Software Defined Networks. In *Network Operations and Management Symposium (NOMS)*. IFIP/IEEE.
[16] B. Claise. 2018. Cisco Systems NetFlow Services Export Version 9. https://tools.ietf.org/html/rfc3954.
[17] P4 Language Consortium. 2018. P4 Switch Behavioral Model. https://github.com/p4lang/behavioral-model.
[18] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification, version 1.0.0. http://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.
[19] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. 2008. Finding Hierarchical Heavy Hitters in Streaming Data. In *Transactions on Knowledge Discovery from Data, Volume: 1, Issue: 4*. ACM.
[20] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-performance Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[21] Nick Duffield, Carsten Lund, and Mikkel Thorup. 2001. Charging from Sampled Network Usage. In *Internet Measurement Workshop (IMW)*. ACM.
[22] Cristian Estan, Ken Keys, David Moore, and George Varghese. 2004. Building a Better NetFlow. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[23] Cristian Estan, Stefan Savage, and George Varghese. 2003. Automatically inferring patterns of resource consumption in network traffic. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[24] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[25] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. 2001. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Transactions on Networking, Volume: 9, Issue: 3*. IEEE/ACM.

[26] Yan Gao, Yao Zhao, Shobha V. Schweller, Yan Chen, Sawn Song, and Ming-Yang Kao. 2007. Detecting stealthy attacks using online histograms. In *International Workshop on Quality of Service (IWQoS)*. IEEE.

[27] The P4.org Applications Working Group. 2018. In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf.

[28] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[29] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Symposium on SDN Research (SOSR)*. ACM.

[30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[31] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[32] Lavanya Jose, Minlan Yu, and Jennifer Rexford. 2011. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*. USENIX.

[33] Nils Kammenhuber and Lukas Kencl. 2005. Efficient statistics gathering from tree-search methods in packet processing systems. In *International Conference on Communications (ICC)*. IEEE.

[34] Faisal Khan, Nicholas Hosein, Chen-Nee Chuah, and Soheil Ghiasi. 2011. Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis. In *Architectures for Networking and Communications Systems (ANCS)*. IEEE Computer Society.

[35] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based Change Detection: Methods, Evaluation, and Applications. In *Internet Measurement Conference (IMC)*. ACM.

[36] Anukool Lakhina, Mark Crovella, and Christophe Diot. 2004. Diagnosing Network-wide Traffic Anomalies. In *Computer Communication Review, Volume: 34, Issue: 4*. ACM.

[37] Anukool Lakhina, Mark Crovella, and Christophe Diot. 2005. Mining Anomalies Using Traffic Feature Distributions. In *Computer Communication Review, Volume: 35, Issue: 4*. ACM.

[38] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[39] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[40] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. 2006. Is Sampled Data Sufficient for Anomaly Detection?. In *Internet Measurement Conference (IMC)*. ACM.

[41] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[42] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[43] Noviflow. 2018. Noviswitch 1132 product guide. https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2.0.pdf.

[44] Diana Andreea Popescu, Gianni Antichi, and Andrew W. Moore. 2017. Enabling Fast Hierarchical Heavy Hitter Detection Using Programmable Data Planes. In *Symposium on SDN Research (SOSR)*. ACM.

[45] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[46] Vyas Sekar, Michael K. Reiter, and Hui Zhang. 2010. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *Internet Measurement Conference (IMC)*. ACM.

[47] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[48] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Symposium on SDN Research (SOSR)*. ACM.

[49] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. 2010. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *Passive and Active Measurement (PAM)*. Springer-Verlag.

[50] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[51] Yinglian Xie, Vyas Sekar, David A. Maltz, Michael K. Reiter, and Hui Zhang. 2005. Worm Origin Identification Using Random Moonwalks. In *Security and Privacy (SP)*. IEEE Computer Society.

[52] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[53] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. 2013. FlowSense: Monitoring Network Utilization with Zero Measurement Cost. In *Passive and Active Measurement (PAM)*. Springer-Verlag.

[54] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Networked Systems Design and Implementation (NSDI)*. USENIX.

[55] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. 2007. ProgME: Towards Programmable Network Measurement. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[56] Ying Zhang. 2013. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM.

[57] Qi (George) Zhao, Mitsunori Ogihara, Haixun Wang, and Jun (Jim) Xu. 2006. Finding Global Icebergs over Distributed Data Sets. In *Principles of Database Systems (PODS)*. ACM.

A.5 Paper V

Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques

Patrik GOLDSCHMIDT and Jan KUČERA. "Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques". In: *Proceedings of 2021 IFIP/IEEE International Symposium on Integrated Network Management*. IM 2021. Bordeaux, France: IFIP, 2021, pp. 772–777. ISBN: 978-3-903176-32-4.

Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques

Patrik Goldschmidt
CESNET a.l.e., Prague, Czech Republic
goldschmidt@cesnet.cz

Jan Kučera
CESNET a.l.e., Prague, Czech Republic
jan.kucera@cesnet.cz

***Abstract*—TCP SYN Flood is one of the most widespread DoS attack types performed on computer networks nowadays. As a possible countermeasure, we implemented and deployed modified versions of three network-based mitigation techniques for TCP SYN authentication. All of them utilize the TCP three-way handshake mechanism to establish a security association with a client before forwarding its SYN data. These algorithms are especially effective against regular attacks with spoofed IP addresses. However, our modifications allow deflecting even more sophisticated SYN floods able to bypass most of the conventional approaches. This comes at the cost of the delayed first connection attempt, but all subsequent SYN segments experience no significant additional latency (<0.2ms). This paper provides a detailed description and analysis of the approaches, as well as implementation details with enhanced security tweaks. The discussed implementations are built on top of the hardware-accelerated FPGA-based DDoS protection solution developed by CESNET and are about to be deployed in its backbone network and Internet exchange point at NIX.CZ.**

***Index Terms*—TCP SYN Flood, DDoS mitigation, TCP SYN Authentication, RST Cookies, SYN Drop, TCP Handshaker**

I. Introduction

Transmission control protocol (TCP) is an integral part of the Internet protocol suite. As its importance is fundamental for the operation of the Internet, it is often misused to cause various cybersecurity threats. Data from the past several years show a strong trend towards TCP abuse to perform Distributed Denial of Service (DDoS) attacks. The report from Q2 2020 by Kaspersky Lab states that the most frequent way of a DDoS was TCP SYN flooding, utilized by 94.7% of all the attacks [1]. According to Cisco, the number of DDoS attacks will double to 14.5 million p.a. by 2022 [2].

The purpose of this paper is to describe details of this weakness, present existing countermeasures, and most importantly, convey our experience with the implementation and evaluation of three network-based SYN Flood mitigation strategies. Our implementations have been designed as a part of the DDoS protection for high-speed networks developed by CESNET [3].

In this work, we extend the device's mitigation capabilities by developing network-based heuristic approaches to mitigate SYN Flood attacks. The discussed methods are not as widespread as end-host TCP SYN Cookies but are way more effective in certain situations, such as when dealing with an enormous amount of spoofed IP addresses, and enable flexible utilization according to the current attack surface. Proposed algorithms are used only as a reactive defense against ongoing cyber-attacks, hence not affecting the traffic when no threats are detected. With the use of high-capacity data structures posing as IP whitelist and blacklist, our robust solution also supports SYN limiting, creating an especially strong mitigation mechanism even against more sophisticated attacks.

This research was supported by the Security Research Programme of the Czech Republic 2015 – 2022“ (BV III/1 – VS) granted by the Ministry of the Interior of the Czech Republic under No. VI20192022137 Adaptive protection against DDoS attacks.

978-3-903176-32-4 © 2021 IFIP

II. TCP Security Considerations

The creation of a reliable data channel required by the TCP is achieved by a three-way handshake. The process starts with an initiating host (client). The client sends a segment with the SYN flag set and generates a pseudo-random value of x used as the Sequence number (SEQ_x). The receiving host (server) then generates its SEQ_y, acknowledges the client's data by setting its *ACK* to the received segment $SEQ_x + 1$, and enables SYN and ACK flags. The client also acknowledges the received segment, and the channel setup is completed.

A. Known Vulnerabilities and Attacks

Attacks on the TCP are classified as either flood-based or injection-based. Flood attacks aim to exhaust the target's resources by flooding with bogus packets, making it inaccessible for regular clients, hence creating a denial of service. On the other hand, injection attacks are based on eavesdropping on the communication and injecting crafted segments into the TCP session. Injected data may contain malicious code, compromise the user's privacy [4], or reset the session [5].

The functionality of the most popular attack – TCP SYN Flood depends on the 3-way-handshake mechanism, during which the server waits for the arrival of the final ACK to mark the connection as established. The rationale behind a successful DoS assumes that the victim allocates a new state for every received SYN segment and that there is a limit of such states that can be stored. These are defined in RFC 793 [6] as Transmission Control Block (TCB) data structures. TCBs are used to store necessary state information for each TCP connection, and so they require a new memory allocated for each received SYN [7].

Operating system kernels typically try to protect host memory from exhaustion by implementing a limit of contemporary

TCB structures called backlog. When its limit is reached, either incoming SYN segments are ignored, or uncompleted connections in the backlog are replaced. The primary goal of the SYN flooding is thus to exhaust the target's backlog with half-open connections. For this purpose, spoofed IP addresses that do not generate replies to SYN-ACKs, are often used.

Other attacks include various types of floods, e.g., SYN-ACK, RST, and FIN flood. More sophisticated techniques, such as Fake session, include ACK and FIN segments alongside many SYNs to simulate a real client's traffic. Another technique is Session attack, which utilizes a botnet to create many valid TCP connections at once and stretch them as long as possible, making the victim server inaccessible.

B. SYN Flood Mitigation Techniques

Modern operating systems provide relatively large backlogs, being less vulnerable to regular SYN flooding attacks. However, backlogs can not cover distributed variants of the attack, so specialized methods are still required. Linux kernels historically provided robust security by implementing two end-host countermeasures – *SYN cookies* and *SYN caching* [8].

SYN cache method utilizes hashing to store a lightweight fingerprint for every incoming TCP connection. This way, the operating system does not need to allocate the whole TCB, but only a fragment of the original memory is required. Therefore, it is able to queue more requests and so is harder to exhaust [8].

In contrast to SYN cache, the SYN cookies method does not need to store any state information at all. Essential data defining the connection alongside a timestamp and a secret are hashed into a 32-bit value representing the *SEQ* number of the SYN-ACK segment. Upon ACK response receipt, the server can reconstruct original SYN parameters and successfully establish a connection. However, the method denies SYN-ACK retransmission and also restricts TCP options usage [9].

A little-used but interesting approach is *TCP Random drop*. Its principle is to replace a random half-open connection when the backlog is full and another SYN is received. Replacement is done by sending a RST segment, discarding the TCB structure, and allocating a new one for the incoming connection. Dropped legitimate clients are expected to try to reestablish a connection. Its rationale is that by making the queue large enough, a server under attack can still offer a high probability of successful connection establishment, but legitimate sessions may still be occasionally denied [10].

Although often effective, the presented end-host mitigation techniques are not suitable in all scenarios. Some of them could be implemented as a part of the intermediary device software, but their usage would require a mapping between different *SEQ* and *ACK* numbers, making their operation rather inefficient. Therefore, other specialized approaches also exist.

Various TCP extensions and modifications with anti-DoS capabilities like TCP Cookie Transactions [11] and TCP Fast Open [12] are also available. However, none of them is globally used, mainly due to the lack of support from vendors.

Other network-based countermeasure techniques include *traffic filtering* [13] and its improved variant *reverse-path forwarding* [14]. Their fundamental idea is to deny all incoming traffic that does not match its source network prefix. This allows discarding all traffic from spoofed IP addresses, but its deployment would be necessary on the majority of Internet service providers, which cannot be generally relied on [15].

SYN Flood attacks were historically mitigated by firewalls, proxies, or IDS/IPS systems, which usually used SYN-ACK spoofing or ACK spoofing techniques [15] [16]. These practices are mostly present to this day but often reside in the cloud as a part of virtualized IDS/IPS systems instead of traditional per-network defense [17]. The methods mentioned above are, however, not always optimal. SYN-ACK spoofing does not solve the mentioned problem with degraded performance due to the required SYN/ACK values mapping. ACK spoofing can protect the server's backlog but distributed SYN floods may still cause network congestion or high processor utilization of the security intermediary device or the server itself. Therefore, another three methods providing both good performance and decent SYN-flood protection are introduced in Section III.

Both end-host and network-based techniques are frequently employed, and they generally do not interfere when used in combination [15]. Newer trends in DDoS mitigation also utilize artificial intelligence and machine learning principles, such as [18], which are generally able to protect against a wider range of attacks but suffer from a poorer performance when compared to their previously mentioned counterparts.

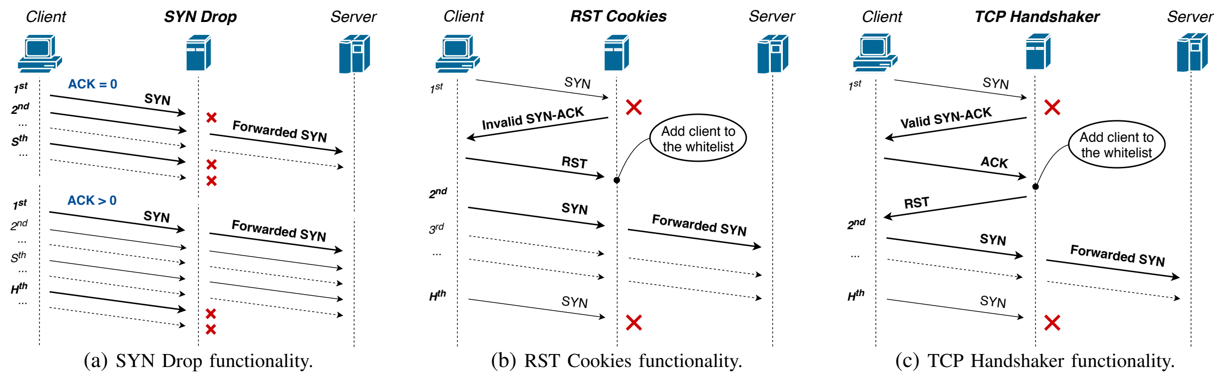
III. Network-based Mitigation Methods

The SYN Cookies end-host mitigation principle proved to provide adequate protection against SYN Flooding attacks, but its usage may be undesirable in certain situations. Since servers are typically busy handling clients' requests, it may be preferable to filter the traffic on the network level, thus not wasting their resources by processing potentially malicious traffic. For this purpose, three TCP SYN authentication algorithms are presented in the following subsections.

A. SYN Drop

SYN Drop mitigation method is based on a simple principle to limit the maximum number of sent SYNs from a single IP address. For this reason, the module needs to keep an internal state for all clients, monitor their connections, and count the number of SYNs and ACKs received from them. The number of allowed SYNs is given by *soft* (S) and *hard* (H) thresholds. When no ACK from the corresponding IP address is received in the particular time window, the soft threshold, allowing only a couple of packets, is active. If at least one ACK is received, SYNs are limited by the hard threshold, which may allow up to hundreds of connections in a few seconds. Additionally, the first SYN in the soft threshold's case is always dropped to prevent SYN port scanning (Fig. 1a).

The described mechanism effectively denies dummy heavy-hitters by policing the maximum number of SYNs a single host can send. Nevertheless, attackers utilizing massive IP address spoofing may produce enough traffic, managing to take down



(a) SYN Drop functionality.

(b) RST Cookies functionality.

(c) TCP Handshaker functionality.

Fig. 1: Network-based TCP SYN Flood mitigation techniques using TCP SYN authentication.

the server with only the soft threshold active. Alternatively, ACK messages injection or Fake session attacks would be able to fool the mechanism and make it ineffective. Therefore, we also present other SYN Flood mitigation methods in subsections III-B and III-C.

B. RST Cookies

The first mention of the RST Cookies approach can be traced back to 1996 [10]. Unfortunately, the method was never officially published, and the original proposal was only in the form of e-mail communication. It was also never popularized due to incompatibility with Windows 95 [19] and potential performance issues on slow networks. Mentioned e-mails were probably deleted, and so only a few resources exist to this day. For the purpose of our custom implementation, the method needed to be "reinvented", estimating the behavior of the clients according to the specification and actually testing various operating systems for the expected compatibility.

RST Cookies is a heuristic mitigation technique utilizing the 3-way handshake mechanism while relying on the client's behavior as defined in RFC 793. Its fundamental idea is to establish a security association with clients before allowing their connection requests. This is achieved by crafting an intentionally invalid SYN-ACK response to the first SYN received from a client. RFC 793 [6] defines the behavior as follows:

> *If the connection is in any non-synchronized state, and the incoming segment acknowledges something not yet sent (carries an unacceptable ACK), a reset is sent.*

To distinguish whether the RST segment is associated with the invalid SYN-ACK reception, RFC 793, section 3.4 [6] also defines requirements on the sent RST:

> *If the incoming segment has an ACK field, the reset takes its sequence number from that ACK field.*

According to these preconditions, we can distinguish a legitimate client from an attacker without storing any state information locally. Instead, the client's authentication is performed solely on the value stored in the *ACK* field of the SYN-ACK reply. The first SYN from a new (not whitelisted) client is dropped, and an invalid SYN-ACK reply is used to verify its validity (Fig. 1b). The regular client will send a valid RST reply, whereas the attacker without the real TCP stack will not. When the valid RST is received, a security association is established by whitelisting the client's IP address. SYN traffic originating from whitelisted IP addresses is forwarded to its desired destination without further tampering (Fig. 1b).

The algorithm hence blocks all received SYN segments from unknown hosts until a security association with them is established. On account of this behavior, the protected server does not initially know about the intentions to establish a session and thus no state information are allocated until the client is considered legitimate. This mechanism effectively denies all attacks from spoofed IP addresses, which can not generate a valid reply. Random RST segments can not fool the security mechanism because the specific *SEQ* value is expected. For the attacker without a valid TCP stack, it is rather problematic to inject a RST segment with the desired *SEQ*. Therefore, the only viable way is to use legitimate (non-spoofed) clients. As further discussed in Section IV, we enhanced the original algorithm by implementing a hard threshold to limit the maximum number of SYNs from already-validated clients (Fig. 1b), thus merging it with the functionality of SYN Drop.

C. TCP Handshaker

SYN Authentication with *TCP Handshaker* is practically a version of the end-host SYN Cookies method ported to the network-based environment. As outlined in Section II-B, the trustworthiness of the initiating host is verified by setting a specific *SEQ* in the SYN-ACK response and then verifying the value from the ACK segment the client returns.

The TCP Handshaker method (Fig. 1c) works according to this principle, where a specific *SEQ* value is set in the SYN-ACK response and expects the client to confirm its validity by responding with a required value of $SEQ+1$ in its ACK segment. If the values are matched, its IP address is added to the whitelist, and its SYN data are not intervened anymore. However, after the client sends an ACK finalizing the handshake, it thinks that the session is established because the 3-way handshake was successfully finished from its perspective. The problem at this point is that the server knows nothing about the session since the client's SYN was intercepted by the algorithm, and thus never reached the server. To synchronize nodes in this state, the TCP Handshaker needs to send a RST

```
1:  entry ← IP data from association table
2:  if (entry == NIL):
3:      send (invalid) SYN-ACK; drop SYN and exit;
4:  else if (t_s − entry.t_a > t_m):
5:      delete IP from association table;
6:      send (invalid) SYN-ACK; drop SYN and exit;
7:  if (SYN limiting enabled):
8:      if (t_s − entry.window_start ≥ 1s):
9:          entry.syn_cnt ← 0;
10:         entry.window_start ← t_s;
11:     else if (entry.syn_cnt ≥ SYN limit):
12:         if (Blacklist enabled):
13:             add IP to blacklist;
14:         delete IP from association table;
15:         drop SYN and exit;
16:     entry.syn_cnt ← entry.syn_cnt + 1;
17: allow SYN and exit;
```

Fig. 2: SYN Processing of RST Cookies / TCP Handshaker.

for the client's session after its ACK is processed. When the RST is received, the client closes its session and may start another one automatically based on its implementation (Fig. 1c). This behavior is further discussed in Section V.

IV. Design and Implementation Remarks

Since the SYN Drop technique is rather simple, this section will mostly focus on the problematics related to RST Cookies and TCP Handshaker. The following subsections discuss SYN processing and the client authentication process in more detail.

A. SYN Processing

All of the presented methods require to process all ingress SYN segments. In addition, RST Cookies and TCP Handshaker have to process all RSTs or ACKs, respectively. When a SYN is received, the algorithm must determine whether it originates from a new client or a client that is already verified. For this purpose, we use a hash table with the source IP address as its key. The contents of its entries depend on the mitigation method, but various timestamps and counters have to be used. For example, each entry for RST Cookies or TCP Handshaker contains a timestamp specifying when the association has been created (t_a), allowing entries to age. Therefore, upon a SYN segment arrival (t_s), these algorithms have to check whether the IP address is contained in the whitelist and its entry timestamp does not exceed the maximum specified age time (t_m), thus validating the condition: $t_s - t_a > t_m$. If the condition is met, the SYN is dropped, and a valid or invalid SYN-ACK is assembled and sent as a response. Otherwise, the processed SYN is forwarded to its desired destination.

A regular version of SYN processing is depicted in Fig. 2 (lines 1-6). We enhanced the algorithm functionality by adding a counter (*syn_cnt*) and timestamp (*window_start*) to the hash table alongside the existing association timestamp. These are used to implement the hard SYN threshold functionality for already associated clients (lines 7-16). The modified algorithm with the hard limit will successfully block sending large amounts of SYNs by any sophisticated attackers, who would manage to guess the whitelisted IPs or somehow pass through the security association phase. When combined with a blacklist, an ability to detect these smart attackers and deny their traffic entirely is available as well (lines 12-13).

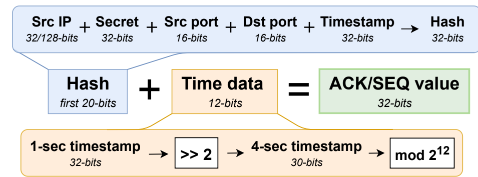


Fig. 3: SYN-ACK generation – the hashing method.

B. Validation Packets Processing

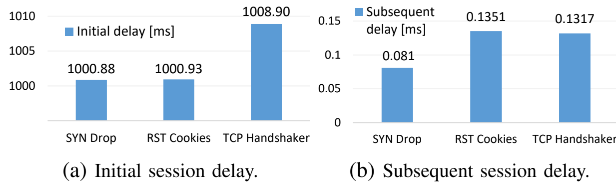
RST segments for RST Cookies and ACKs for TCP Handshaker are used for client validation and thus need to be treated differently in these particular methods. These algorithms have to decide whether the RST/ACK is a part of its authentication mechanism or belongs to the regular TCP traffic. This is done by looking at the *SEQ* (for RSTs) or *ACK* (for ACKs) of the obtained data. If this value is equal to the expected value defined by the algorithm, it is probably a response to its previously sent SYN-ACK. In this case, the processed segment is dropped, and the client's IP address whitelisted. Otherwise, the algorithm has to forward the segment to its destination.

C. ACK/SEQ Values Generation and Validation

The key concept of both RST Cookies and TCP Handshaker is to generate either valid or invalid SYN-ACKs and use their *ACK* or *SEQ* values for future client authentication. In the RST Cookies' case, an invalid SYN-ACK is crafted by setting its *ACK* value differently from the $SEQ + 1$ of the SYN it is referring to. TCP Handshaker requires a valid SYN-ACK, but we are still free to set its *SEQ* as desired. Both of the algorithms hence need to generate a value that cannot be easily guessed by an attacker and which they can reconstruct when required. The simplest solution to this is a constant value placed in each SYN-ACK response and then checked for the match in the RST/ACK. This approach would be functional, but a smart attacker could monitor the traffic and inject the required type of packet with the given constant to trick the security mechanisms. To tackle this issue, we propose a system of dynamic TCP number generation and validation.

Our design of the dynamic TCP number generator and validator uses two policies with two security levels. The first policy generates random numbers periodically and assigns the values to SYN-ACK segments according to the generation time. When a client's response is being processed, the algorithm iterates over the structure of these lastly generated values and searches for a match between the generated and the value read from the segment. The number of iterated elements depends on the generation period and the validity of generated values. When configured sensibly, this method is faster and allows considerably better throughput than its counterpart.

The second approach is somewhat inspired by SYN Cookies. As illustrated in Fig. 3, a unique hash is computed for every connection. The source IP, a 32-bit secret, source and destination ports, as well as a 32-bit timestamp, are hashed into a 32-bit string. Its 12 least significant bits are replaced with a modulo of the timestamp shifted to 4-second precision. This technique provides a reasonable trade-off between security



(a) Initial session delay. (b) Subsequent session delay.

Fig. 4: SYN session delays (including 3-way handshake).

and performance because the attacker would have to guess 2^{20} possibilities from the hash alongside four different timestamps. Four-second precision protects against a replay attack for $2^{2^{12}}$s $\sim$ 194 days (required for the timestamp repetition). Received value verification is then done by reconstructing the timestamp by deriving its value before the modulo operation was applied. This process involves shifting back to the 1-second precision and computing the hash function for every possible second in the given time window. If the reconstructed timestamp is within the timeout range and the first 20-bits of the computed hash match the first 20-bits of the *SEQ* (*ACK*) in the analyzed RST (ACK), the client is considered legitimate. This method undoubtedly provides stronger security since unique values are generated per connection instead of the single value for all connections in the given time window.

Both policies can further be combined with two security modes – crypto- and non-cryptographic. The cryptographic mode uses cryptographically-secure hashing and number generation, aiming to deny generated values guessing and estimation completely. On the other hand, the non-cryptographic variant utilizes regular hashing and pseudo-random numbers, providing high performance at the cost of lowered security.

V. Results and Closing Remarks

Firstly, the compatibility of the presented methods with modern operating systems has been confirmed. This was done with a browser and various console applications, which were supposed to establish a connection to a server protected by our algorithm implementations. As expected, all three methods always caused the first session establishment to fail, and a client had to react accordingly. Our results show that systems without any IPS activated, namely Windows XP – Windows 10, Linux kernels 3, FreeBSD 11, Apple iOS 12, macOS 10.14, and higher are all respecting the TCP standard, and thus being fully compatible. All the methods behave transparently to the protected devices, and all OSes tried to automatically re-establish the session, making the methods transparent for user applications as well. However, we discovered that newer Fedora-based distributions utilize *nftables* stateful connection tracking to drop invalid packets. Such configuration effectively prevents the OS from receiving invalid SYN-ACK responses, making it RST Cookies-incompatible.

The client's behavior is fully dependent on the used method because it defines *how* the session fails. The following paragraphs will examine these behaviors, whereas Fig. 4 summarizes the initial and subsequent session establishment delays for an average of 10k *netcat* data transfers on CentOS 7.8.

SYN Drop causes the session to fail by dropping the first received SYN. The time of its retransmission depends on the client's TCP stack, influenced either by an operating system or an application. The most frequent value we came across was 1000ms (Fig. 4a), but other values may also be present [20].

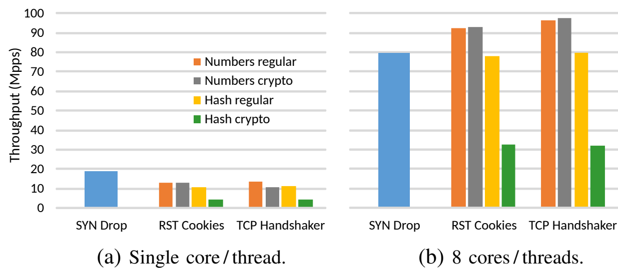
RST Cookies brings a session into an erroneous state by an invalid ACK. In this case, the client is supposed to reply with a RST segment and try to reestablish the session on the same port. This process is also application and host-dependent. The typical period we encountered was also 1000ms (Fig. 4a) on Windows OSes and most *nix (including Android and iOS) applications and popular browsers (Chrome, Edge).

TCP Handshaker closes the first client's session with a RST. In this case, the session needs to be reestablished on a different source port, which OSes and simple programs typically do not perform. However, more robust applications tend to open a new port automatically after a certain period. Since this process requires OS kernel intervention, the initial session delay is slightly higher as in the previous methods (Fig. 4a).

While simple programs like netcat rely on an OS's TCP stack and a single port, more sophisticated applications usually initiate multiple TCP connections at once for a single user request. Two to four sessions are opened initially, followed by another after 200-300ms (e.g., Chrome). Although the retransmission period of 1s is rather high, additional attempts after 200ms are typically sufficient to set up a TCP channel since the client's IP address is already whitelisted. Such applications are thus able to successfully establish the connection without waiting for a retransmission timer. Fig. 4b shows that subsequent whitelisted connections experience no significant delay (<0.2ms). We evaluated all the values for the worst-case scenario using cryptographic hashing. Therefore, simpler security mechanisms like non-cryptographic random number generation tend to reduce these delays slightly. Though delays up to 1s may seem high, it is important to realize that these methods are activated only when an ongoing attack is detected. Therefore, no delays occur during a regular operation, and a slight initial delay is highly preferable to service unavailability while an attack is in progress.

Memory requirements and packet throughput shall be considered as well. All the presented methods require a whitelist data structure to monitor SYN-sending IP addresses and store state information for decision-making. RST Cookies and TCP Handshaker require 20B per whitelisted client if SYN limiting is enabled. For example, peaks on the CESNET's network from March to August 2020 reached up to 540k TCP flows per second. For this purpose, only 640 MiB of memory would be needed for 33.55M client entries, hence containing all of them for a period of one minute. Although the SYN Drop method requires only 13B per client, it needs to store state information for all (also spoofed) clients.

Packet throughput is mostly influenced by the number of processed hash functions. Our implementations contain at least one hashing per TCP segment to access the whitelist. When the hashing security mechanism is used, two hashes per SYN and up to five per RST (RST Cookies) or ACK (TCP Handshaker) segments are needed to validate them. Fig. 5a shows the



(a) Single core / thread. (b) 8 cores / threads.

Fig. 5: Mitigation methods performance comparison.

throughput comparison of these mitigation methods during the simulated SYN Flood attack. The attack was composed of SYN packets originating from randomized IP addresses and ports. As expected, the SYN Drop method performed the best, achieving a throughput of 29.69Mpps. Usage of more sophisticated mitigation furthermore reduced the throughput to less than 15Mpps according to the used security mode.

Modern network interface cards support a mechanism called RSS (Receive Side Scaling), enabling to distribute packets into multiple receive queues, which can be processed by separate CPUs. Therefore, we commonly run numerous instances of these algorithms on multiple CPU cores. This way, a throughput of more than 97Mpps (∼71Gbps) on 8 independent queues can be achieved (Fig. 5b). In this case, the packet rate of more robust methods is higher than of the SYN Drop. This is due to the nature of mixed IPv4/IPv6 segments with different lengths, which have to be forwarded as they are. For this reason, the TX interface buffers are used inefficiently, and so the overall performance decreases. On the other hand, our custom-generated SYN-ACKs are padded, so the performance for these high-speed packet rates may be significantly higher.

Alongside legitimate and attack data, RST Cookies and TCP Handshaker have to cope with responses to the traffic they generate as well. Its volume could become rather significant as the clients not contained on whitelists try to establish new connections. For this reason, we also provide throughput comparisons with a changing vector of clients' RST (ACK) messages in contrast to the received number of SYNs. Note that SYN analysis requires one memory access for whitelist search and an alternative SYN-ACK generation if the client is not verified. In contrast, RST/ACK analysis requires to validate their *ACK* value, either by memory access for the random numbers variant or up to four extra hash computations. As expected, only hashing security policies are affected since memory access is negligible when compared to hashing.

For example, consider a RST:SYN ratio of 0.1 for non-cryptographic RST Cookies hashing and suppose that all the RSTs are destined to the mitigation mechanism, the throughput for 1 thread is 10.4Mpps. If the ratio increases to 1.0 (all SYN senders need to verify themselves with a certain RST), the throughput falls to 9.6Mpps. Even more significant decline can be observed for the cryptographic hash policy, which falls from 4.0Mpps to 2.3Mpps per thread. Since TCP Handshaker uses the same mechanism for ACK generation and validation, its results are quite the same as the cases described here.

VI. Conclusions

This paper has focused on the analysis of the TCP SYN Flood attack and discussed three network-based mitigation methods as its possible countermeasure. Presented experimental results are based on a custom implementation and evaluation with commonly used operating systems and applications.

The simplest method, SYN Drop, offers sufficient protection against pure SYN Floods from regular or spoofed IP addresses. Advanced algorithms – RST Cookies and TCP Handshaker, allow detection and blocking of more sophisticated attacks, able to bypass conventional techniques by simulating the traffic of a real client. The mitigation can function on up to 97Mpps, but prolongs the establishment of the first session.

Further optimizations for better performance and memory requirements can still be conducted. Used memory can be minimized by probabilistic data structures such as Bloom filters. Appropriate hash functions could also improve the overall throughput significantly. Even bigger performance demands may lead to offloading a part of SYN Flood mitigation algorithms into the FPGA device programmable dataplane.

Our future work will focus on observation and analysis of attack vectors of real-world situations, given our experience with DDoS protection solution deployment in operational environments at CESNET's backbone and NIX.CZ.

References

[1] Kupreev *et al.*, "DDoS Attacks in Q2 2020," Kaspersky Lab, Tech. Rep., Aug 2020, https://securelist.com/ddos-attacks-in-q2-2020/98077.

[2] Cisco Systems, "Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper," Tech. Rep., Jan 2017.

[3] CESNET a.l.e., "DDoS Protector," Sep 2020, https://www.liberouter.org/technologies/ddos-protector/.

[4] B. Harris and R. Hunt, "TCP/IP security threats and attack methods," *Computer Communications*, vol. 22, no. 10, 1999.

[5] P. A. Watson, "Slipping in the Window: TCP Reset Attacks," Jan 2004.

[6] J. Postel, "Transmission Control Protocol," RFC 793, IETF, Sept 1981.

[7] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," RFC 4987, IETF, Aug 2007.

[8] J. Lemon, "Resisting SYN Flood DoS Attacks with a SYN Cache," in *Conference on the BSD operating system (BSDCon)*, 2002.

[9] D. J. Bernstein and E. Schenk, "SYN Cookies proposal," Sept 1996, http://cr.yp.to/syncookies/archive.

[10] L. Ricciulli, "TCP SYN Flooding Defense," in *Communication Networks and Distributed Systems Modeling and Simulation (CNDS)*, 1999.

[11] W. Simpson, "TCP Cookie Transactions," RFC 6013, IETF, Jan 2011.

[12] Y. Cheng, J. Chu *et al.*, "TCP Fast Open," RFC 7413, IETF, Dec 2014.

[13] P. Fergusson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks," RFC 2827, IETF, May 2000.

[14] Baker, F. and Savola, P., "Ingress Filtering for Multihomed Networks," RFC 3704, IETF, Mar 2004.

[15] W. M. Eddy, "Defenses Against TCP SYN Flooding Attacks," *The Internet Protocol Journal*, vol. 9, no. 4, Dec 2006.

[16] C. L. Schuba, I. V. Krsul *et al.*, "Analysis of a Denial of Service Attack on TCP," in *Symposium on Security and Privacy (SP)*, 1997.

[17] O. Osanaiye *et al.*, "Distributed denial of service (DDoS) resilience in cloud," *Journal of Network and Computer Applications*, vol. 67, 2016.

[18] C. Li *et al.*, "Detection and defense of DDoS attack-based on deep learning," *International Journal of Communication Systems*, 2018.

[19] Linux Kernel Mailing List Archive, "T/TCP: SYN and RST Cookies," Apr 1998, https://lists.gt.net/linux/kernel/12829.

[20] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer," RFC 6298, IETF, June 2011.

A.6 Paper VI

Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning

Patrik GOLDSCHMIDT and Jan KUČERA. "Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning". In: *Proceedings of 2024 IEEE/IFIP Network Operations and Management Symposium.* NOMS 2024. Seoul, South Korea: IEEE, 2024.

Windower: Feature Extraction for Real-Time DDoS Detection Using Machine Learning

Patrik Goldschmidt
Brno University of Technology, Brno, Czech Republic
Kempelen Institute of Intelligent Technologies, Bratislava, Slovakia
igoldschmidt@fit.vut.cz

Jan Kučera
Brno University of Technology, Brno, Czech Republic
CESNET a.l.e., Prague, Czech Republic
jan.kucera@cesnet.cz

***Abstract*—Distributed Denial of Service (DDoS) attacks are an ever-increasing type of security incident on modern computer networks. To tackle the issue, we propose Windower, a feature-extraction method for real-time network-based intrusion (particularly DDoS) detection. Our stream data mining module employs a sliding window principle to compute statistical information directly from network packets. Furthermore, we summarize several such windows and compute inter-window statistics to increase detection reliability. Summarized statistics are then fed into an ML-based attack discriminator. If an attack is recognized, we drop the consequent attacking source's traffic using simple ACL rules. The experimental results evaluated on several datasets indicate the ability to reliably detect an ongoing attack within the first six seconds of its start and mitigate 99% of flood and 92% of slow attacks while maintaining false positives below 1%. In contrast to state-of-the-art, our approach provides greater flexibility by achieving high detection performance and low resources as flow-based systems while offering prompt attack detection known from packet-based solutions. Windower thus brings an appealing trade-off between attack detection performance, detection delay, and computing resources suitable for real-world deployments.**

***Index Terms*—network intrusion detection, NIDS, DDoS mitigation, real-time, stream data mining, machine learning**

I. Introduction

Distributed Denial of Service (DDoS) is one of the most prevalent cyber-attacks on today's Internet. It is described as a deliberate attempt to render a target system unavailable, threatening the key cybersecurity goal – availability. The attack is typically performed via a large number of geographically distributed computers known as a botnet. Such computers simultaneously generate a huge amount of requests to overwhelm a target system or its underlying network architecture.

DDoS usage has grown massively since the first attack [1] reported in 1999. Cisco predicts 15.4 million DDoS attacks in 2023 [2]. In Q1/2023, Cloudflare observed an increase in hyper-volumetric attacks, peaking above 71 Mrps [3]. Volumetric attacks like DNS amplification, SYN flood, GRE flood, and other TCP/UDP floods are the most popular [3], [4].

In general, defense against cyberattacks spans three high-level objectives – prevention, detection, and reaction [5]. It is well-known that perfect prevention cannot be achieved, whereas the reaction implicitly assumes that the attack has already happened [6]. In this light, attack detection is crucial in successfully reacting toward harm minimization or attack deflection (mitigation).

Attack detection is performed by Intrusion Detection Systems (IDSs), which monitor the activity of a protected system, intending to identify its potential unauthorized use, misuse, or abuse [7]. Identified malicious activities are collected centrally and presented to a security officer. Afterward, either automated or manual reaction can take place.

Most notable aspects of IDSs [8] include: I) Information source, specifying whether network traffic (Network IDS – NIDS) or end-host data are analyzed; II) Detection approach, defining a detection principle – known signatures or deviations from a norm; III) Detection time, putting constraints on the detection delay; and IV) Response type, as whether the system actively participates in attack mitigation after its detection.

We present Windower, a feature-extraction method for real-time network-based intrusion (particularly DDoS) detection at the network perimeter. It processes packet headers and utilizes stream data mining and windowing techniques to extract relevant traffic statistics. We designed Windower to detect flooding attack and periodicity patterns, but it also proved efficient against low-rate DoS. To demonstrate its capabilities, we employed KitNet, an autoencoder ensemble from the Kitsune [9] NIDS, as an anomaly-based attack discriminator and simulated both detection and mitigation scenarios.

As most current machine learning (ML)-based NIDS research implicitly assumes off-line scenarios [10], [11], we specifically aimed to propose a practical and reliable method for real-time attack detection and consequent mitigation. It also allows stream learning, recommended for real-time cybersecurity solutions [12]. The paper's main contributions are:

- We propose a packet-based sliding window method[1] to compute traffic statistics from packet headers using stream data mining, suitable for real-time intrusion detection and mitigation.
- We introduce a specific feature set particularly for volumetric DDoS but also prove its ability to detect low-rate DoS attacks. Moreover, features are detector-independent, allowing both misuse- and anomaly-based approaches.
- We demonstrate Windower's performance in terms of detection rate and latency on several public datasets (CAIDA, CTU-13, SUEE-2017, UNSW-NB15) and compare it to the state-of-the-art packet-based NIDS.

[1] Code publicly available at: https://github.com/xGoldy/Windower

This paper firstly looks at problems of flow-based NIDS for real-time detection in Sec II. We further discuss related work in Sec III and method design in Sec. IV. Experimental results are presented in Sec. V, followed by rigorous discussion in Sec. VI. Finally, Sec. VII concludes the paper.

II. Flow-Based DDoS Detection Pitfalls

The current mainstream way for ML-based DDoS detection predominantly utilizes network flows. This fact is mainly attributed to the wide availability of flow-based datasets, e.g., CIC-IDS2017 [13], CIC-DDoS2019 [14], and ISCX2012 [15], currently considered the most popular for research [10]. Such approaches typically achieve detection rates above 99% [10]. However, in addition to losing information about packet payloads, flow-based detection suffers from two more defects:

1) Network flows provide no communication context.
2) Detection delays occur due to flow creation mechanism.

A network flow captures only a single data exchange between a client and a server, providing no information context about the client's previous communication. Therefore, some attacks using port randomization (e.g., tools HOIC or XOIC incrementing source ports) or port scans create a separate flow for each attack packet. The capabilities of flow-based methods without flow correlation are thus significantly hampered.

Flow-based methods suffer from delays due to creation and export intervals. Typically, a flow terminates after observing a TCP FIN flag or when a timeout occurs. However, attacks like SYN Flood do not carry FIN flags, so flows must be terminated upon timeout. Moreover, observed flows are exported in bulks, so dozens of seconds might pass until the flow data arrives at a flow collector. Afterward, a NIDS still needs to access the collector to retrieve the entries, adding additional delay.

Although the network flow collection is widely deployed and well-matured, it was created with the aim of monitoring and not directly for cyberthreat detection. Therefore, flow-based attack detection might be unsuitable if a near-instant reaction to the attack is required, as it can be significantly delayed, and additional post-processing might be required.

As a possible solution, we propose a packet stream method to compute statistics via a sliding window aggregated based on the source IP. In such a way, it provides the client's communication context by design, thus avoiding the necessity of flow correlation. Since no export must occur, the detection delay is minimized to a few seconds after the attack begins.

III. Related Work: Real-Time ML DDoS Detection

Real-time attack mitigation is the main design goal of practical anti-DDoS solutions [16]. Delays of dozens of seconds or even minutes might thus be unacceptable. In general, there are three main approaches to speed up the detection process: 1) *Window aggregates*, 2) *Per-packet processing*, 3) *Subflows*.

A. Window Aggregates

As the whole data stream cannot be processed at once, windowing splits a (potentially infinite) data stream into logical subsets (windows). Applying operations separately to each window allows for partial stream analysis. Thus, it enables to detect and react upon events (e.g., attacks) in a timely manner.

Time-based windows are based on timestamps, whereas *count-based* windows rely on the order of processed elements (packets/flows) as they arrive from the network.

Vijazasarathy [17] detects DDoS using TCP flags and durations of count-based windows with the Naive Bayes classifier. Similarly, Mousavi and St-Hilaire [18] proposed a DDoS detection for SDN based on the entropy of destination IP addresses within five 50-packet windows and thresholding. Based on Jensen-Shannon Divergence, Bhandari [19] differentiates DDoS attacks and flash events within short 1-second windows.

Aggregates can also be created upon flows. For instance, GEE [20] aggregates flows per source IP inside 3-minute windows. Although such methods achieve relatively high accuracy, timely attack detection is degraded due to NetFlow properties (Sec. II) along with too long window lengths.

B. Per-Packet Processing

Per-packet NIDSs evaluate each packet within the attack discriminator separately. Such systems utilize statistical techniques like n-grams [21] or signatures (e.g., Snort [22]) to search specific patterns within packets. Despite the signatures' efficiency, the research has moved towards ML, especially Recurrent Neural Networks (RNNs), due to their ability to maintain context. DeepDefense [23] uses a bi-directional LSTM-RNN with the last 100 packets, achieving $\sim 98\%$ accuracy and a 1-2% error rate. LSTM-BA [24] improved its performance by combining an RNN with a Naive Bayes classifier.

Per-packet methods are sometimes combined with windowing for additional context. Kitsune [9] uses per-packet feature extraction and KitNET, an ensemble of autoencoders – unsupervised artificial neural networks, for anomaly detection. It keeps the communication context for channels (conversations between two hosts) using five damped time windows, computing 115 features per packet. Chronos [25] employs the same feature extraction as Kitsune, but their handling and model architecture differ. Finally, Doshi et al. [26] combine stateful window along with stateless packet-header features.

Although the discussed methods achieve promising results and Kitsune sparked great interest within the community, they still require evaluating every packet in the detection model. Despite sufficient for smaller-scale local networks, the per-packet approach might be infeasible for larger networks with only a few dozen nanoseconds to process a packet available.

To enable DDoS detection and mitigation in high-speed networks, we argue that one must classify at higher-level abstraction, i.e., windows or network flow aggregates. As outlined in Sec. II, flow-based methods might add additional delay. Therefore, window-based methods seem to be a promising research direction for real-time detection purposes.

C. Subflows

A subflow is formed by the same 5-tuple (IPs, ports, protocol) as a regular flow but is restricted by the number of packets and/or its duration. Therefore, only the first n

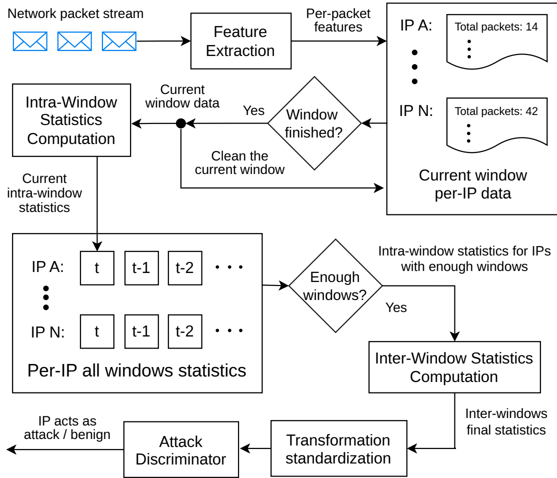


Fig. 1: Architecture of the proposed DDoS detection system.

packets of the flow are examined, or it is terminated after a certain time. These approaches are plausible since they can utilize existing NetFlow architecture while still speeding up the detection. Nevertheless, the client's communication context is still missing, requiring additional post-processing (e.g., as in the NeMo DDoS software [27]) for better performance.

AE-D3F NIDS [28] uses an autoencoder to detect DDoS via features like packet/byte counts, packet sizes, and TTLs based on 10-second bi-directional subflows. Similarly, Lucid [29] aggregates packets within a subflow and performs detection using a Convolutional Neural Network (CNN). Liang and Znati [30] classify the whole flow based on its first N packets via an LSTM model. These designs are less computationally demanding than per-packet approaches but still might be insufficient for real-time operation. Another drawback is that these methods ([29], [30]) expect an input of a certain length, so dummy packets are needed for less than N-packet flows.

IV. System Design

Our main design objective was to provide a lightweight method aiming for the best trade-off between attack mitigation metrics and detection timeliness. We thus had to consider: I) providing attack-distinguishing features in a timely manner and II) evaluating such features quickly enough to keep up with the network. Despite achieving similarly high detection rates [31], flow-based data are not always sufficient for timely detection, whereas processing every packet might be infeasible in large-scale networks. Therefore, we opted to employ sliding-window statistics aggregated per source IP.

Since we aggregate the data based on the source IP address, we expect the same-source traffic to be routed via the same path so it can be captured. Actually, the detection method only examines incoming (potentially malicious) traffic and does not care about the responses, thus monitoring the packet-wise traffic in an uni-flow manner. Relying only on ingress traffic effectively decreases the detection latency, resource requirements, and relaxes the constraints on traffic routing.

Windower is depicted in Fig. 1. Firstly, we extract header values from each packet (Tab. I). They are used to compute the current window statistics, which we aggregate by their source IP. After a specified amount of windows is reached, window-wise statistics are summarized, and standard deviations are computed between windows (Sec. IV-A). Such summaries (Tab. II) are fed into a model to determine whether the host resembles anomalous behavior (Sec. IV-C). After detection, attack mitigation is launched.

A. Feature Extraction and Statistics Computation

As illustrated by Fig. 1, our method is designed to process a raw network data stream. However, instead of processing every packet within the attack discriminator, we extract relevant packet features and compute communication statistics for each source IP address using summarized sliding windows for attack detection. This process is divided into several steps:

1) Packet Feature Extraction: We utilize only packet headers (Tab. I) instead of payloads, making the method application-level independent and unaffected by encryption. Features can characterize various DDoS attacks with temporal and packet sizing patterns different from benign traffic.

2) Sliding Window Per-Source Aggregation: We aggregate packet features by their source IP address inside a sliding window of t seconds. Aggregation per source IP creates a natural communication context, providing rigorous patterns of a single host across multiple network flows. No additional correlation between network flows is hence required.

Aggregation within the current window is done by counts, sums, averages, standard deviations, minimums, maximums, unique counts, and an entropy estimation for every unique IP in the window (Tab. II). As the mechanism might need to process dozens of gigabits per second, it would be infeasible to store extracted features from all packets within each window.

Counts, sums, minimums, and maximums can be computed on the go by a simple update. However, computations of regular average and variance assume that all the samples are available at the time of the computation. For this reason, we utilize data stream (running) algorithms to compute such statistics without storing samples. The streaming mean is computed using Eq. 1, and we use Welford's algorithm [32] for variance. It maintains an auxiliary value s_i updated for every element (Eq. 2) along with running mean $\overline{x}_i$ (Eq. 1). As the window finishes, the variance estimate is computed using Eq. 3, and the standard deviation then simply as $\sigma = \sqrt{s^2}$.

$$\overline{x}_n = \overline{x}_{n-1} + \frac{x_n - \overline{x}_{n-1}}{n} \quad (1)$$

$$s_n = s_{n-1} + (x_n - \overline{x}_n) \cdot (x_n - \overline{x}_{n-1}) \quad (2) \qquad s^2 = \frac{s_n}{n-1} \quad (3)$$

Computing the unique source port count would normally be trivial via the cardinality of a set. Nevertheless, sets require

TABLE I: List of all extracted packet features.

# **Packet Features** (*I=Integer, S=String, B=Boolean*)		
1. Timestamp (I)	5. Destination IP (S)	7. Headers length (I)
2. Source IP (S)	6. Destination port (I)	8. Payload length (I)
3. Source port (I)	4. L4 protocol (I)	9. IP fragmented (B)

TABLE II: List of all intra- and inter-window statistics.

#	Intra-Window Statistics
1. *src_ip*	IP address for the corresponding statistics
2. *window_count*	Number of summarized windows
3. *window_span*	The first and the last window ID difference
4. *pkts_total*	Total number of packets
5. *bytes_total*	Sum of bytes of all packets
6. *pkt_rate*	Estimate of a packets per second (pps) value
7. *byte_rate*	Estimate of a bytes per second (bps) value
8. *pkt_arrivals_avg*	Average time between packet arrivals
9. *pkt_arrivals_std*	Standard deviation between packet arrivals
10. *pkt_size_min*	Overall minimum packet size
11. *pkt_size_max*	Overall maximum packet size
12. *pkt_size_avg*	Average of packet sizes
13. *pkt_size_std*	Standard deviation of packet sizes
14. *proto_tcp_share*	TCP traffic share
15. *proto_udp_share*	UDP traffic share
16. *proto_icmp_share*	ICMP traffic share
17. *port_src_unique*	Number of unique source ports
18. *port_src_entropy*	Source port entropy
19. *conn_pkts_avg*	Average number of socket-to-socket transfers
20. *pkts_frag_share*	Fragmented packets share
21. *hdrpkt_ratio_avg*	Average of header to packet size ratio

#	Inter-Window Statistics (*Std=Standard Deviation*)
22. *pkts_total_std*	Std of a total number of packets
23. *bytes_total_std*	Std of a total number of bytes
24. *pkt_size_avg_std*	Std of packet size averages
25. *pkt_size_std_std*	Std of packet size stds
26. *pkt_arrivals_avg_std*	Std of average time between packet arrivals
27. *port_src_unique_std*	Std of number of unique source ports
28. *port_src_entropy_std*	Std of source port entropy values
29. *conn_pkts_avg_std*	Std of packet count per connection averages
30. *pkts_frag_share_std*	Std of fragmented packets share
31. *hdrpkt_ratio_avg_std*	Std of header to whole packet ratios
32. *main_proto_ratio_std*	Std of ratio of the dominant L4 protocol
33. *intrawin_activity_ratio*	IP estimate within the windows
34. *interwin_activity_ratio*	IP estimate during the period

saving every unique element in the memory. Instead, we rely on HyperLogLog (HLL) [33], a probabilistic structure for reducing space demands. It cannot give a definite answer but provides an approximation within some maximum error range [34]. Using HLL, we achieve a standard error of 4.6% with less than 1 kB of memory per single window entry.

Lastly, we compute the client's source port entropy (the amount of randomness). Although several techniques for calculating streaming entropy exist [35], [36], we opted for regular Shannon Entropy [37] computed from a sample of n=40 observed source ports obtained by Reservoir sampling [38].

3) Intra-Window Statistics: After the current window finishes, a new one is started. However, some information is not used as collected, but additional (intra-window) statistics are computed (Tab. II). Windows are considered valid when they contain at least p packets. Such a restriction aims to reduce statistical noise introduced by small sample sizes. This phase also computes running variances and other statistics, like the average number of packets per connection.

4) Window Summarization and Inter-Window Statistics: Using only one window might not properly capture the variability of communication over time. For instance, flash events resemble identical characteristics to a DDoS [39]. When a flash event is captured by a single window, its features might look similar to an attack. To make Windower more robust, we utilize multiple windows to describe communication patterns more reliably. The functionality is achieved by 1) summarizing multiple windows and 2) computing additional inter-window statistics (Tab. II). Similarly to the minimum packet limit, we set the minimum window count w for noise reduction.

After collecting w windows, the features are summarized via arithmetic means over available per-IP window statistics. We compute TCP, UDP, and ICMP traffic shares, as well as the header-to-payload length ratio. For specific attacks, e.g., SYN flood, the attacker sends packets without payload. These features can thus help to reveal noticeable attack patterns.

Inter-window statistics are computed as standard deviations of corresponding statistics across windows. Their rationale is that regular benign traffic would likely have a substantial variance over machine-generated attacks. Attack tools usually limit an attack to a specific bitrate, achieving high uniformity, thus leading to low variance across multiple time windows. The premise might be incorrect for pulsing DDoS [40] or benign streaming services. Nevertheless, we compensate for it with other features (packet size analysis and packet/byte rates).

We also estimate the client's intra- and inter-window activity during the analyzed period. Intra-window activity is based on a communication gap at the start and end of the window (Eq. 4). Inter-window activity is given as a ratio of the summarized window count to all possible windows within the range (Eq. 5). Due to the p packets window validity requirement, a client might not produce enough data to fill in the window, whereas a flooding attack should have both values close to 1.

$$a_{\text{intra}} = \frac{t_{\text{last-pkt}} - t_{\text{first-pkt}}}{t_{\text{win-len}}} \quad (4) \qquad a_{\text{inter}} = \frac{\#_{\text{windows-summarized}}}{\text{ID}_{\text{last-win}} - \text{ID}_{\text{first-win}}} \quad (5)$$

5) Statistics Preprocessing: Statistics computed in the previous phase require preprocessing before being fed into the attack discriminator. Although the source IP is important for alerting and mitigation, we drop it before classification to prevent evaluation bias [41]. We further drop features *window_count* and *window_span*, which can be useful during postprocessing to adjust the decision confidence but are not helpful during the classification itself. Therefore, 31 features to process within the detection model are left. Since all are numerical, no encoding is required. Nevertheless, we rescale them with z-score normalization to maximize the performance.

B. Hyperparameters Summary

In summary, we can tune the following parameters influencing the method's performance:

- *Window length* (t): Length of a sliding window in seconds.
- *Packet count* (p): We consider a window valid only if a certain source host sent at least p packets within it.
- *Window count* (w): The feature vector is finalized once at least w windows are collected in the last v seconds.
- *Window validity* (v): Windows older than v seconds are outdated and not considered for statistics computation.

As a consequence of our design, an attack can only be detected in $t \cdot w$ seconds, supposing that each window has at least p packets and no window got invalidated due to exceeding

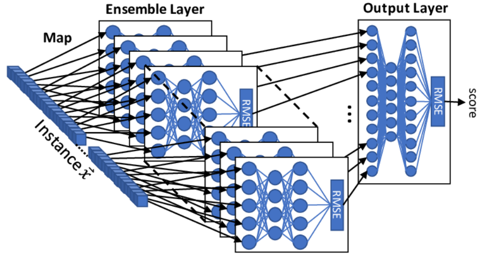


Fig. 2: KitNet [9] architecture, the ensemble of autoencoders.

maximum validity time v. Longer windows are considered more reliable (decreasing false positives), while shorter allow faster detection. Packet count p assures reliable intra-window, while window count w ensures reliable inter-window statistics.

C. Attack Detection

For a demonstration of the attack-detection capabilities, we employed KitNet [9]. However, the extracted features can be utilized by both misuse- and anomaly-based detectors.

KitNet (Fig. 2) is an ensemble of k three-layer autoencoders (AEs). It is a part of Kitsune [9], a state-of-the-art anomaly-based NIDS for online attack detection. Each ensemble AE measures the abnormality of its respective subspace $\vec{v}_i$, given by input $\vec{x}$ mapping f into $k-1$ subspaces, producing a Root Mean Squared Error (RMSE) value. RMSEs are then led to the autoencoder at the output layer, giving a final RMSE value.

During training (with benign only), the model first learns the feature mapping function f using agglomerative hierarchical clustering. In our case, $\vec{x}$ is a vector of 31 elements. Individual autoencoders then learn distributions of their respective subspaces $\vec{v}_i$, and the output layer AE learns the relationship between subspace abnormalities and naturally occurring noise.

The evaluation phase uses learned mapping f and individual autoencoder distributions to produce the final RMSE score. We compare it with the predefined threshold τ, $\mathrm{RMSE} > \tau$ representing an anomaly. During mitigation, we correlate anomalies with the acting source IP to drop its packets. Further KitNet details can be found in the original Kitsune paper [9].

V. Evaluation

We evaluate Windower using four packet-based datasets. This section describes our setup and analyzes the performance compared with Kitsune [9]. We also analyze the performance by varying windowing configurations and discuss the detection speed, KitNET's complexity, and the runtime performance.

A. Utilized Datasets

The selection of a suitable dataset was non-trivial as the NIDS domain suffers from a long-term shortage of standardized quality datasets [6], [42], [43]. Although various datasets were released recently [44], none cover a wide variety of DDoS attacks and provide packet-based data suitable for our purposes. Datasets are very limited in the attack scope or the number of attacking hosts. For instance, CIC-DDoS2019 [14] provides various DDoS attacks, but all its attack traffic comes from a single IP address. Since we perform aggregation based on source IPs, we could not use it because it produced too few aggregated window entries, insufficient for reliable evaluation.

As no dataset is perfect [45], we used multiple ones to evaluate our work in different scenarios and thus strengthen the achieved results and stated claims. When discussing the CAIDA dataset, we refer to a custom mix of the CAIDA DDoS Attack 2007 dataset [46] and the CAIDA Anonymized Internet Traces [47]. Since the DDoS dataset contains only attack traffic, we mixed it with benign traffic from CAIDA Traces. CTU-13 [48] is a dataset of real botnet traffic. We extracted only DDoS from the scenario 45, excluding any C&C traffic. Similarly, we extracted only packets corresponding to the DoS traffic class along with a subset of normal traffic from UNSW-NB15 [49]. Former datasets consist primarily of flood attacks. In order to test the performance against slow DoS attacks as well, we used the SUEE8 variant of the SUEE-2017 [50].

For our purposes, we extracted only the forward direction (targeting the protected network) without responses and created a train set (only benign) and a test set (both benign and malicious traffic) for each dataset, assuming distinct attacking and legitimate IP addresses. We thus label the datasets via attackers' IPs. The number of packets, along with their class affiliation, is given in Tab. IV. Full details about the dataset compilation process are provided on our GitHub (Sec. I).

B. Experiments Setup

We implemented Windower in Python to process the discussed datasets. The evaluation utilized a full feature set – 31 features after preprocessing (Sec. IV-A). A fair comparison with Kitsune was achieved using the same attack detector – KitNET, with equal parameters, i.e., m=10 (the maximum number of inputs for each AE) and the same feature mapping procedure. Performance comparisons of our system with Kitsune are based on its original Python implementation [9]. During training, we always use 10% of the training set for feature mapping. Unless stated otherwise, we set the Windower's parameters as follows: t=1, p=10, w=6, and v=120, i.e., aggregate features across six 1-second long windows.

We executed the experiments on a single CPU core (Intel Xeon E5-2630v3) and 64 GB of RAM. Given modern NICs support of distributing packets into multiple receive queues processed by separate cores, the implementation can be easily parallelized and accelerated in the future.

C. Per-Packet Mitigation Analysis

We evaluate detection capabilities using a simulated mitigation process based on RMSE scores. Fig. 3 shows the RMSE scores assigned to each packet in the CAIDA dataset for both Kitsune and Windower. We use different colors to distinguish scores of benign (green) and attack (red) packets. Each data point in the figure represents a single packet classification.

For Kitsune (Fig. 3a), the output layer produces the RMSE score for each packet as an implicit result of the per-packet processing approach. We can compare this score with a predefined threshold τ to decide whether to drop or pass the packet.

TABLE III: Comparison of mitigation performance using TPR for FPR={0.002, 0.005, 0.01, 0.02, 0.05}.

Dataset	FPR=0.002		FPR=0.005		FPR=0.01		FPR=0.02		FPR=0.05	
	Kitsune	Windower	Kitsune	Windower	Kitsune	Windower	Kitsune	Windower	Kitsune	Windower
CAIDA	0.00000	0.98964	0.00000	0.98964	0.00000	0.98964	0.00000	0.98964	0.00000	0.98964
CTU-13	0.00000	0.99859	0.00000	0.99859	0.00000	0.99859	0.00000	0.99859	0.00000	0.99859
UNSW-NB15	0.54952	0.57073	0.60907	0.66585	0.73372	0.73230	0.81191	0.86095	0.85211	0.97638
SUEE-2017	0.07270	0.22996	0.23747	0.36122	0.33350	0.91729	0.48011	0.91729	0.92678	0.91729

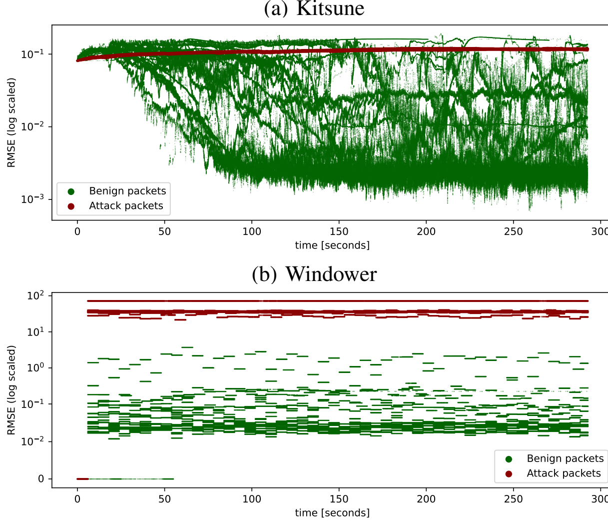


Fig. 3: Visualization of the per-packet Root Mean Square Error (RMSE) values evaluated using the CAIDA dataset.

Unlike Kitsune, Windower (Fig. 3b) always produces an RMSE score for a specific source IP address and its corresponding feature vector based on an aggregated set of time window statistics. To plot Fig. 3b and to fairly compare both solutions, we thus assign RMSE scores to packets during postprocessing according to the latest source IP address classification. In practice, we would directly compare the score of each source IP address classification with τ to immediately decide whether to drop or allow all its packets. For this reason, in Fig. 3b, the RMSE values for individual packets create 6-second segments (lines) as we are dealing with the time-based source IP aggregated windows.

The approach using source IP window-based classification will always introduce a small number of false negatives by design. In the first 50 seconds of CAIDA, in Fig. 3b, one can see several packets assigned zero RMSE score. These packets will be allowed due to not having enough collected data (at least six 1-second time windows) to classify their source IP address, so their RMSE value is undefined. Unless stated otherwise, we always include such false negatives caused by detection delays in the performance evaluation metrics.

As evident from Fig. 3, Windower outperforms Kitsune on the CAIDA dataset. For Windower, we can easily find a threshold, e.g., τ=10, that ideally separates legitimate (green) and attack (red) traffic using the RMSE scores. In contrast to Kitsune, we can only find such τ by introducing a non-negligible number of false positives or negatives.

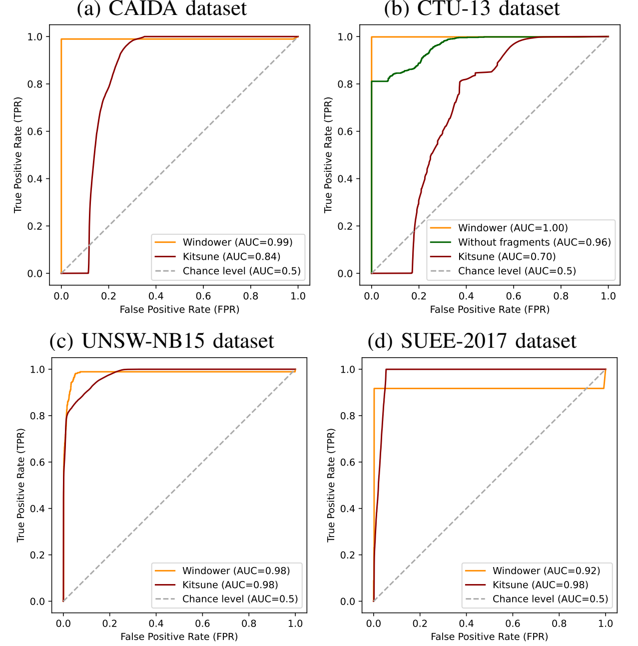


Fig. 4: Mitigation performance depicted using ROC curves.

D. Mitigation Performance

We investigate the method's performance regardless of a specific τ using an ROC (Receiver Operating Characteristics) curve. It plots true (TPR) versus false positive rates (FPR) with regard to increasing threshold. The numerical representation of the classifier's performance is given by the Area Under the ROC Curve (AUC), i.e., AUC=1.0 being a perfect score.

Fig. 4 compares Windower (orange) against Kitsune (red) in terms of mitigation performance using ROC curves. In cases of CAIDA and CTU-13, Windower significantly outperforms Kitsune. For both datasets, Kitsune introduces a substantial amount of false positives (above 30%) to detect a similar number of malicious packets as Windower. From the AUC score perspective, Windower achieves AUC=0.99 for CAIDA and AUC=1.00 for CTU-13, respectively, while Kitsune reaches only AUC=0.84 for CAIDA and AUC=0.70 for CTU-13.

In contrast, the mitigation performance is more or less comparable for UNSW-NB15 and SUEE-2017. Regarding the AUC score, both methods achieved AUC=0.98 for UNSW-NB15, and Kitsune (AUC=0.98) even outperformed Windower (AUC=0.92) for SUEE-2017. However, Windower still introduces fewer false positives with significantly fewer computational requirements (Sec. V-F). We give detailed TPR/FPR results in Tab. III. In the DDoS mitigation case, we are more

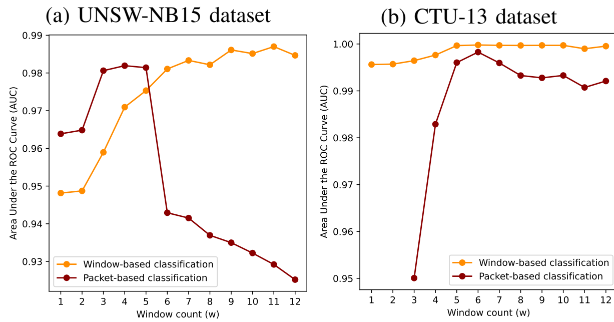


Fig. 5: Windows count (w) analysis using AUC.

concerned with false positives than false negatives. We thus claim that Windower still achieves decent performance for slow attacks with meaningful benefits over Kitsune.

Analysis of CTU-13 revealed that a significant portion of attack traffic is fragmented. For this reason, we analyzed Windower's performance with and without two fragmentation-related features. As seen in Fig. 4b, the performance without fragmentation (green) achieves an AUC of 0.96. With fragmentation, even AUC=1.00 is achieved. Nevertheless, both variants significantly outperform Kitsune (AUC=0.70).

E. Window Count Impact: Detection Delay vs. Accuracy

Fig. 5 investigates the impact of the number of collected windows on the Windower's performance. It shows the AUC score (y-axis) for the UNSW-NB15 and CTU-13 datasets with the varying parameter w – the minimum number of collected windows. We present the score from two different viewpoints:

1) *Window-based classification* (orange) shows the performance without the impact of false negative packet classification (discussed in Sec. V-C) caused by the delay in collecting the minimum number of windows. It reveals the pure success rate of source IP address classifications (benign vs. attack source). The more windows we use (the more statistics we have collected), the higher the classification accuracy.

2) *Packet-based classification* perspective (red) considers the impact of the delay in collecting the specified number of windows. For instance, Windower performs best if w=5 for UNSW-NB15 and w=6 for CTU-13, respectively. Indeed, the more windows we use (the more time we need for collection), the higher the false negative rate effect. The parameter w generally enables fine-tuning the trade-off between detection accuracy and its delay in real-time mitigation.

F. Runtime Performance

Tab. IV shows the runtime performance and compares Windower with Kitsune for all datasets. As apparent, Windower outperforms Kitsune in both training and evaluation runtimes. For illustration, Kitsune needs 83.1 min. for training, while the Windower needs only 14.1 min. (∼6x faster) using CAIDA. In some cases, the speed-up is even more significant (an order of magnitude) for the CTU-13 and SUEE-2017 datasets.

Although Kitsune relies on the ensemble of small autoencoders to increase its speed, it must still evaluate each incoming packet. In addition, the computational complexity of the ensemble itself grows with the number of features. Kitsune's feature extractor derives more than a hundred features for each packet to maintain its context aggregated based on source IP address, as well as channel and socket perspectives.

In contrast, Windower reduces the feature count (only 31) by using statistical features computed across multiple windows. A smaller feature set results in fewer ensemble autoencoders to train and evaluate. Tab. IV also presents the number of autoencoders for individual datasets (*#AEs*). As demonstrated, Windower requires up to 2-3x less AEs.

More importantly, we substantially lower the number of ensemble evaluations (*AE evals*). For Kitsune, it coincides with the packet count as it works on a per-packet basis, e.g., ∼3M AE evaluations using CAIDA. On the other hand, Windower classifies window-based feature aggregates and blocks packets based on source IP addresses. Using the same dataset, it needs no more than 3k evaluations (three orders of magnitude less). For this reason, Windower is fundamentally faster and still offers similar, or even better, mitigation performance.

VI. Discussion

In this section, we discuss possibilities of advanced mitigation, elaborate on real-time operation, and outline probable adversarial behavior and attacks against the system itself.

A. Intelligent Mitigation

In the previous section, we demonstrated the mitigation capabilities using an Access Control List (ACL) mechanism. If an anomalous IP is determined, it is denylisted, and all its consequent traffic is explicitly dropped. We further suggest performing ACL filtering before the Windower processing itself. This can save resources by discarding data from potentially malicious clients using hardware, hence not analyzing the traffic of denylisted hosts. After a certain period, the denylist entry expires, and the host's traffic is allowed again.

More advanced mitigation can be achieved by signature or rule derivation. For instance, if Windower detects a host acting anomalously, it could sample its future traffic. Afterward, algorithms based on string matching [51], state machines [52], string patterns or semantic conditions [53], or AI [54] can be applied to fill up the database of mitigation rules.

Assuming that the same malicious tool generated the attack, there should be similarities in its traffic. If an attacker had not utilized any obfuscation (discussed in Sec. VI-C), the created rules should generalize and block the traffic even from clients not analyzed by the attack detection mechanism.

B. Real-Time Performance

Windower aims to provide reliable real-time DDoS attack detection – a trade-off between detection performance and delay. Per-packet methods (hypothetically) offer the best timeliness, as we know whether to drop or pass a packet instantly. However, deep ML models are too complex to process every packet and meet throughput demands. We thus investigated how to remove the model evaluation from the data path and

TABLE IV: Comparison of Windower's and Kitsune's runtime performance.

Dataset	# packets			Kitsune runtime performance				Windower runtime performance			
	trainset	*evalset (attack/all)*		*train* $[min]$	*eval* $[min]$	*# AEs*	*AE evals*	*train* $[min]$	*eval* $[min]$	*# AEs*	*AE evals*
CAIDA	2 003 716	944 764	3 067 177	83.1	76.4	31	3 067 177	14.1	23.6	9	2 789
CTU-13	21 498 729	190 859	18 807 679	2 464.9	6 538.8	16	18 807 679	197.4	178.6	8	2 675
UNSW-NB15	3 179 605	140 360	10 126 614	80.1	150.7	15	10 126 614	26.8	90.5	13	15 182
SUEE-2017	104 313	15 694	291 384	22.6	148.9	22	291 384	1.2	3.2	14	6 812

limit the number of its evaluations to allow for high packet processing speeds and keep up with the incoming traffic.

Our current Windower's implementation as a single Python process achieves a throughput of $\sim$2 kpps. However, our design enables almost linear parallelization via source IP hashing along with separating windowing and attack detection. In such a scheme, the data plane handles feature extraction and intra-window statistics computation. In contrast, the control plane finalizes the streaming and inter-window statistics and runs the attack discriminator upon finishing the window. The model evaluation is thus removed from the data path, speeding up the overall throughput, while the detection delay is controlled via windowing parameters – the window length and the minimum number of summarized windows.

We highlight that Windower's deployment cost is also significantly lower by not evaluating every packet in the attack discriminator. The proposal thus brings a suitable practical trade-off for precise attack detection with regard to computing resources by redesigning only the data preprocessing.

C. Adversarial Considerations

While Windower achieves promising results and has several strengths, it is still not entirely foolproof to various adversarial techniques – most notably, source IP randomization. Below, we examine variants depending on the adversary's knowledge.

Black-box adversaries lack the system knowledge and thus cannot tailor attacks to overcome the detection mechanism. This is a common assumption for DDoS, which relies on an overwhelming traffic volume rather than attack sophistication. Adversaries often use well-known flooding or slow DoS [55]. Despite being designed against flooding attacks, our method shows potential in detecting slow DoS attacks as well.

Our experiments demonstrate the effectiveness against flooding, sharing common characteristics like increased request rates, specific packet timing, or sizing. Since we do not analyze upper protocol headers, this also applies to multi-vector attacks. Adaptive adversaries may use obfuscation like varying packet sizes, inter-packet timings, or payload randomization. Although it could degrade detection capabilities, we argue that normal behavior could hardly be simulated so that some attack patterns would remain visible. Additionally, the need for obfuscation techniques increases the attacker's cost.

A significant threat to our method is source IP randomization. While the method can efficiently handle common source address spoofing (unless the IP stays the same), it struggles when each packet's source IP is randomized, leading to increased memory consumption and statistics computation issues. We address the memory concerns by limiting stored historical windows and by dropping inactive ones with little traffic. However, the computation issue cannot be handled at the method's level. No statistics can be computed if a per-source IP window contains only one packet. It needs to be tackled outside, e.g., by reverse path forwarding [56], [57] or a specific NAT setup to mitigate IP spoofing. Another way of solving the issue is by detecting an attack via correlation [58] or by dropping spoofed packets via derived rules (Sec. VI-A).

Grey-box adversaries possess some system knowledge. Although not typically considered for NIDS [59], we theorize that such adversaries might predict the usage of anomaly-based NIDS and windowing (common for real-time systems). They may then employ adversarial learning, i.e., gradually retrain the system by low amounts of attack traffic to decrease attack sensitivity [60], supposing used incremental learning to tackle concept drift [61]. The windowing can be challenged by pulsing DDoS, reducing efficiency for most systems with only a single window. Nevertheless, our mechanism utilizes multiple ones. Therefore, despite the pulsing effectively lowering the averages, the inter-window variance would stay higher and rather consistent, providing clues about anomalous activity.

White-box (full knowledge) scenarios are less likely as details of NIDSs are typically well protected [62], and much more devastating attacks such as data theft, malware, or ransomware would become more attractive options instead [6].

VII. Conclusions

This work has presented Windower, a feature extraction mechanism based on sliding window and stream data mining. The windowing mechanism first collects per-source IP statistics within a short time frame. They are further summarized with several previous windows to determine more reliable hosts' communication patterns and temporal variability.

Windower achieved promising results, mitigating 99% of malicious DDoS flooding traffic on two public datasets with only 0.2% of false positives. With 1% of false positives, it also successfully filtered 92% of low-rate DoS traffic. Our solution outperforms Kitsune [9] while improving the runtime performance by more than an order of magnitude, making our method more suitable for real-world scenarios.

Windower currently treats statistics from different IPs independently. Our future plans involve exploring correlations between statistics from various hosts, aiming to reduce false positives and enhance flash crowd events resistance. Additionally, enriching the feature set with application-level features might help to characterize modern L7 (D)DoS attacks better.

Acknowledgements

We thank the project e-INFRA CZ (LM2023054) granted by the Ministry of Education, Youth and Sports of the Czech Republic and Brno University of Technology (FIT-S-23-8141).

REFERENCES

[1] P. J. Criscuolo, "Distributed denial of service: Trin00, tribe flood network, tribe flood network 2000, and stacheldraht ciac-2319," *Department of Energy Computer Incident Advisory Capability (CIAC), UCRL-ID-136939, Rev*, vol. 1, 2000.

[2] Cisco Systems Inc., "Cisco Annual Internet Report (2018–2023) White Paper," Cisco Systems Inc., San Jose, CA 95134 USA, Tech. Rep., 2018, updated March 2020. Accessed 2023-09-03. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[3] O. Yoachimik and J. Pacheco, "DDoS threat report for 2023 Q1," Cloudflare, Inc., Tech. Rep., Apr. 2023, acessed 2023-09-03. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-2023-q1/

[4] O. Kupreev, A. Gutnikov, and Y. Shmelev, "DDoS attacks in Q3 2022," Kaspersky, Tech. Rep., Nov. 2022, accessed 2023-09-03. [Online]. Available: https://securelist.com/ddos-report-q3-2022/107860/

[5] H. Yang, H. Luo, F. Ye, S. Lu, and L. Zhang, "Security in mobile ad hoc networks: challenges and solutions," *IEEE Wireless Communications*, vol. 11, no. 1, pp. 38–47, 2004.

[6] G. Apruzzese, P. Laskov, and J. Schneider, "SoK: Pragmatic Assessment of Machine Learning for Network Intrusion Detection," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2023, pp. 592–614.

[7] B. Mukherjee, L. Heberlein, and K. Levitt, "Network intrusion detection," *IEEE Network*, vol. 8, no. 3, pp. 26–41, Jun. 1994.

[8] A. Thakkar and R. Lohiya, "A survey on intrusion detection system: feature selection, model, performance measures, application perspective, challenges, and future research directions," *Artificial Intelligence Review*, vol. 55, no. 1, pp. 453–563, Jan 2022.

[9] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[10] M. Mittal, K. Kumar, and S. Behal, "Deep learning approaches for detecting DDoS attacks: a systematic review," *Soft Computing*, vol. 27, no. 18, pp. 13 039–13 075, Jan. 2022.

[11] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, no. 1, p. 16, Jun 2018.

[12] F. Ceschin, M. Botacin, A. Bifet, B. Pfahringer, L. S. Oliveira, H. M. Gomes, and A. Grégio, "Machine learning (in) security: A stream of problems," *Digital Threats*, sep 2023.

[13] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," in *4th International Conference on Information Systems Security and Privacy*, Jan. 2018, pp. 108–116.

[14] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing Realistic Distributed Denial of Service (DDoS) Attack Dataset and Taxonomy," in *2019 International Carnahan Conference on Security Technology (ICCST)*, 2019, pp. 1–8.

[15] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection," *Computers & Security*, vol. 31, no. 3, pp. 357–374, 2012.

[16] A. Bhardwaj, V. Mangat, R. Vig, S. Halder, and M. Conti, "Distributed denial of service attacks in cloud: State-of-the-art of scientific and commercial solutions," *Computer Science Review*, vol. 39, p. 100332, 2021.

[17] R. Vijayasarathy, S. V. Raghavan, and B. Ravindran, "A system approach to network modeling for DDoS detection using a Naìve Bayesian classifier," in *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, 2011, pp. 1–10.

[18] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *2015 International Conference on Computing, Networking and Communications (ICNC)*, 2015, pp. 77–81.

[19] A. Bhandari, K. Kumar, A. L. Sangal, and S. Behal, "An anomaly based distributed detection system for DDoS attacks in Tier-2 ISP networks," *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 1, pp. 1387–1406, Jan 2021.

[20] Q. P. Nguyen, K. W. Lim, D. M. Divakaran, K. H. Low, and M. C. Chan, "GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection," in *2019 IEEE Conference on Communications and Network Security (CNS)*, 2019, pp. 91–99.

[21] K. Wang and S. J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," in *Recent Advances in Intrusion Detection*, E. Jonsson, A. Valdes, and M. Almgren, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 203–222.

[22] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th USENIX Conference on System Administration*, ser. LISA '99. USA: USENIX Association, 1999, p. 229–238.

[23] X. Yuan, C. Li, and X. Li, "DeepDefense: Identifying DDoS Attack via Deep Learning," in *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2017, pp. 1–8.

[24] Y. Li and Y. Lu, "LSTM-BA: DDoS Detection Approach Combining LSTM and Bayes," in *2019 Seventh International Conference on Advanced Cloud and Big Data (CBD)*, 2019, pp. 180–185.

[25] M. A. Salahuddin, V. Pourahmadi, H. A. Alameddine, M. F. Bari, and R. Boutaba, "Chronos: DDoS Attack Detection Using Time-Based Autoencoder," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 627–641, 2022.

[26] R. Doshi, N. Apthorpe, and N. Feamster, "Machine Learning DDoS Detection for Consumer Internet of Things Devices," in *2018 IEEE Security and Privacy Workshops (SPW)*, 2018, pp. 29–35.

[27] GÉANT Security, "Nemo ddos software," accessed 2024-01-11. [Online]. Available: https://security.geant.org/nemo-ddos-software/

[28] K. Yang, J. Zhang, Y. Xu, and J. Chao, "DDoS Attacks Detection with AutoEncoder," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9.

[29] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martínez-del Rincón, and D. Siracusa, "Lucid: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 876–889, 2020.

[30] X. Liang and T. Znati, "A Long Short-Term Memory Enabled Framework for DDoS Detection," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[31] G. Olimpio, P. F. C. Silva, L. Camargos, R. S. Miani, and E. R. de Faria, "Intrusion detection over network packets using data stream classification algorithms," in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2021, pp. 985–990.

[32] B. P. Welford, "Note on a Method for Calculating Corrected Sums of Squares and Products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[33] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *AofA: Analysis of Algorithms*, ser. DMTCS Proceedings, P. Jacquet, Ed., vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). Discrete Mathematics and Theoretical Computer Science, 06 2007, pp. 137–156.

[34] A. Singh, S. Garg, R. Kaur, S. Batra, N. Kumar, and A. Y. Zomaya, "Probabilistic data structures for big data analytics: A comprehensive review," *Knowledge-Based Systems*, vol. 188, p. 104987, 2020.

[35] N. J. Harvey, J. Nelson, and K. Onak, "Sketching and Streaming Entropy via Approximation Theory," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, 2008, pp. 489–498.

[36] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data Streaming Algorithms for Estimating Entropy of Network Traffic," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '06/Performance '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 145–156.

[37] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[38] K.-H. Li, "Reservoir-Sampling Algorithms of Time Complexity O(n(1 + Log(N/n)))," *ACM Transactions on Mathematical Software*, vol. 20, no. 4, p. 481–493, Dec. 1994.

[39] S. Behal, K. Kumar, and M. Sachdeva, "Characterizing DDoS attacks and flash events: Review, research gaps and future directions," *Computer Science Review*, vol. 25, pp. 101–114, 2017.

[40] X. Luo and R. K. C. Chang, "On a New Class of Pulsing Denial-of-Service Attacks and the Defense," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society, 2005. [Online]. Available: https://www.ndss-symposium.org/ndss2005/new-class-pulsing-denial-service-attacks-and-defense/

[41] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, “Dos and don’ts of machine learning in computer security,” in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3971–3988.

[42] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, and F. Ahmad, “Network intrusion detection system: A systematic study of machine learning and deep learning approaches,” *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 1, p. e4150, Oct. 2021.

[43] R. Sommer and V. Paxson, “Outside the Closed World: On Using Machine Learning for Network Intrusion Detection,” in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 305–316.

[44] Z. Yang, X. Liu, T. Li, D. Wu, J. Wang, Y. Zhao, and H. Han, “A systematic literature review of methods and datasets for anomaly-based network intrusion detection,” *Computers & Security*, vol. 116, p. 102675, May 2022.

[45] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, “A survey of network-based intrusion detection data sets,” *Computers & Security*, vol. 86, pp. 147–167, 2019.

[46] “The CAIDA UCSD ”DDoS Attack 2007” Dataset,” accessed: 2023-10-03. [Online]. Available: https://www.caida.org/catalog/datasets/ddos-20070804_dataset/

[47] “The CAIDA UCSD Anonymized Internet Traces,” accessed: 2023-10-03. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[48] S. García, M. Grill, J. Stiborek, and A. Zunino, “An empirical comparison of botnet detection methods,” *Computers & Security*, vol. 45, pp. 100–123, 2014.

[49] N. Moustafa and J. Slay, “UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set),” in *2015 Military Communications and Information Systems Conference (MilCIS)*, Nov. 2015, pp. 1–6.

[50] T. Lukaseder, “Github: 2017-SUEE-data-set,” 2017, accessed: 2023-10-03. [Online]. Available: https://github.com/vs-uulm/2017-SUEE-data-set

[51] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao, “A Fast String-Matching Algorithm for Network Processor-Based Intrusion Detection System,” *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, p. 614–633, aug 2004.

[52] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, “Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems,” in *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 2010, pp. 111–126.

[53] P.-C. Lin, Y.-D. Lin, and Y.-C. Lai, “A Hybrid Algorithm of Backward Hashing and Automaton Tracking for Virus Scanning,” *IEEE Transactions on Computers*, vol. 60, no. 4, pp. 594–601, 2011.

[54] M. Zadnik and E. Carasec, “AI infers DoS mitigation rules,” *Journal of Intelligent Information Systems*, vol. 60, no. 2, pp. 305–324, Aug 2022.

[55] W. Zhijun, L. Wenjing, L. Liang, and Y. Meng, “Low-Rate DoS Attacks, Detection, Defense, and Challenges: A Survey,” *IEEE Access*, vol. 8, pp. 43 920–43 943, 2020.

[56] D. Senie and P. Ferguson, “Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing,” RFC 2827, May 2000. [Online]. Available: https://www.rfc-editor.org/info/rfc2827

[57] K. Sriram, D. Montgomery, and J. Haas, “Enhanced Feasible-Path Unicast Reverse Path Forwarding,” RFC 8704, Feb. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8704

[58] S. Yu, W. Zhou, W. Jia, S. Guo, Y. Xiang, and F. Tang, “Discriminating DDoS Attacks from Flash Crowds Using Flow Correlation Coefficient,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 6, pp. 1073–1080, 2012.

[59] G. Apruzzese, P. Laskov, E. Montes de Oca, W. Mallouli, L. Brdalo Rapa, A. V. Grammatopoulos, and F. Di Franco, “The Role of Machine Learning in Cybersecurity,” *Digital Threats*, vol. 4, no. 1, pp. 1–38, Mar. 2023.

[60] A. Patcha and J.-M. Park, “An overview of anomaly detection techniques: Existing solutions and latest technological trends,” *Computer Networks*, vol. 51, no. 12, p. 3448–3470, Aug. 2007.

[61] C. Nixon, M. Sedky, and M. Hassan, “Reviews in Online Data Stream and Active Learning for Cyber Intrusion Detection - A Systematic Literature Review,” in *2021 Sixth International Conference on Fog and Mobile Edge Computing (FMEC)*, 2021, pp. 1–6.

[62] G. Apruzzese, M. Andreolini, L. Ferretti, M. Marchetti, and M. Colajanni, “Modeling Realistic Adversarial Attacks against Network Intrusion Detection Systems,” *Digital Threats*, vol. 3, no. 3, feb 2022.