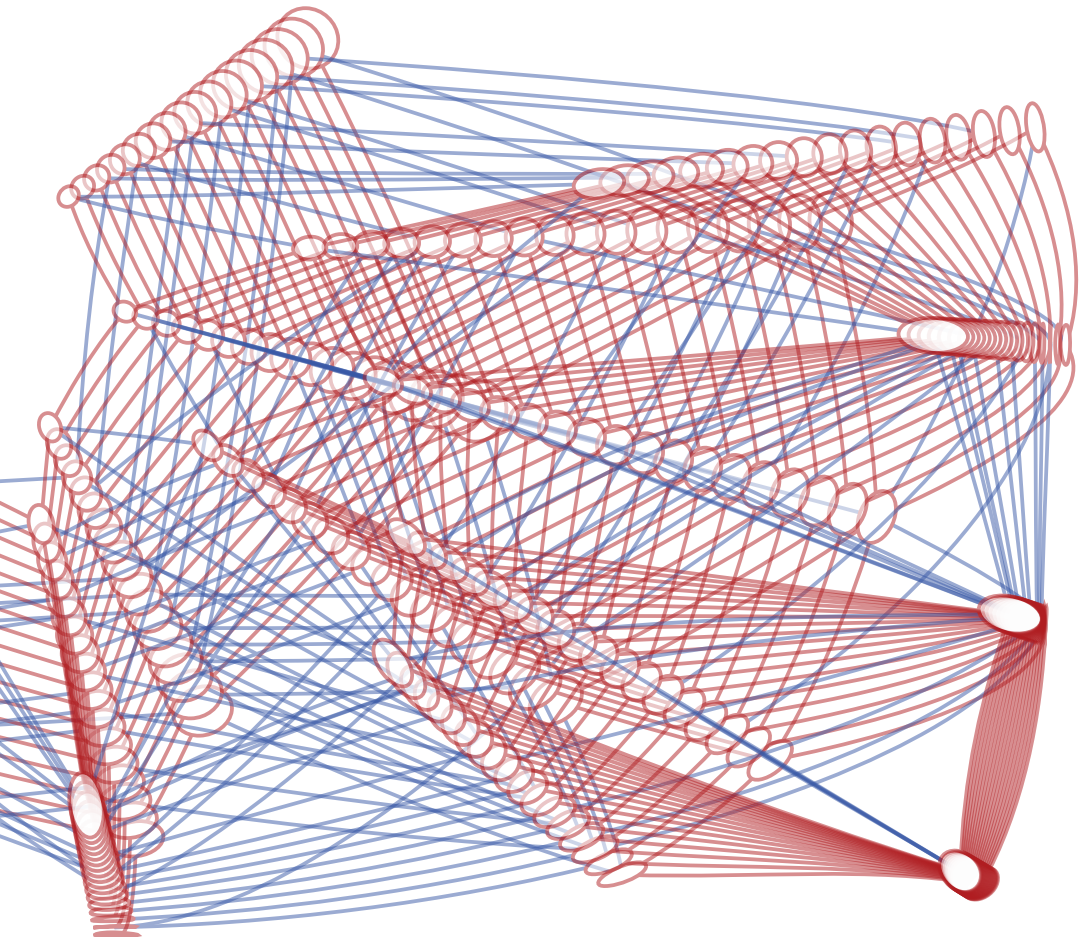


# Improving Network Security through Obfuscation

Diss. ETH No. 28635



**Roland Meier**

DISS. ETH NO. 28635

**IMPROVING NETWORK SECURITY  
THROUGH OBFUSCATION**

A dissertation submitted to attain the degree of  
DOCTOR OF SCIENCES of ETH ZÜRICH  
(Dr. sc. ETH Zürich)

presented by  
ROLAND MEIER  
MSc ETH EEIT  
ETH Zürich

born on July 27th, 1990  
citizen of Zeihen AG, Switzerland

accepted on the recommendation of  
Prof. Dr. Laurent Vanbever (Advisor)  
Dr. Vincent Lenders (Co-Advisor)  
Prof. Dr. Ang Chen  
Prof. Dr. Adrian Perrig

2022

Roland Meier: *Improving Network Security through Obfuscation*, © 2022

Diss. ETH No. 28635

TIK-Schriftenreihe-Nr. 203

## ABSTRACT

---

While it is impressive that many of the prevalent protocols and algorithms in today's networks and the Internet have remained essentially unchanged since the very first computer networks in the Sixties, they were not designed for today's security environment. Only thanks to protocol extensions and new technologies, today's network users are protected against many threats. For example, most hosts are behind firewalls that prevent some malicious traffic from reaching them, and most traffic is encrypted to prevent eavesdropping. However, today's protections are not enough. For example, denial-of-service attacks can cut a host's connection even if their traffic does not reach it, and encrypted traffic still leaks information about its contents.

In this dissertation, we explore how obfuscation can help to prevent such weak points. To this end, we present two solutions:

First, we present *NetHide*, a system that mitigates denial-of-service attacks against the network infrastructure by obfuscating the network topology. The key idea behind *NetHide* is to formulate topology obfuscation as a multi-objective optimization problem that allows for a flexible trade-off between the security of the topology and the usability of network debugging tools. *NetHide* then intercepts and modifies path-tracing probes in the data plane to ensure that attackers can only learn the obfuscated topology.

Second, we present *ditto*, a system that prevents traffic-analysis attacks by obfuscating the timing and size of packets. The key idea behind *ditto* is to add padding to packets and to introduce chaff packets such that the resulting traffic is independent of production traffic with respect to packet sizes and timing. *ditto* provides high throughput without requiring changes at hosts, which makes it ideal for protecting wide area networks.

Both systems leverage recent advances in network programmability. They show that programmable switches can increase the security of high-throughput networks without degrading their performance.

However, programmable switches do not only provide high performance for obfuscation, but they also allow analyzing traffic at scale. We complete this dissertation with a discussion of four use cases where programmable switches analyze traffic – for both benign and malicious purposes.



## ZUSAMMENFASSUNG

---

Viele Protokolle und Algorithmen, die in den heutigen Computernetzwerken und im Internet verwendet werden, sind seit den allerersten Computernetzen in den sechziger Jahren im Wesentlichen unverändert geblieben. Das ist zwar beeindruckend, aber es heisst auch, dass sie nicht für das heutige Sicherheitsumfeld entwickelt wurden.

Nur dank Erweiterungen und neuen Technologien sind die heutigen Netzwerkbenutzer vor vielen Bedrohungen geschützt. So befinden sich beispielsweise die meisten Geräte hinter Firewalls, die verhindern, dass schädlicher Datenverkehr zu ihnen gelangt. Und der meiste Datenverkehr ist verschlüsselt, um das Abhören zu verhindern. Die heutigen Schutzmassnahmen reichen jedoch nicht aus. So können beispielsweise sogenannte Denial-of-Service-Angriffe die Verbindung eines Endgerätes kappen, auch wenn ihr Datenverkehr das Gerät nicht erreicht. Und verschlüsselter Datenverkehr gibt immer noch Informationen über seinen Inhalt preis.

In dieser Dissertation untersuchen wir, wie Verschleierung helfen kann, solche Schwachstellen zu verhindern. Dazu stellen wir zwei Lösungen vor:

Zuerst präsentieren wir *NetHide*, ein System, das die Netzwerktopologie verschleiert, um Denial-of-Service-Angriffe auf die Netzwerkinfrastruktur zu verhindern. NetHide formuliert die Verschleierung der Netzwerktopologie als ein Optimierungsproblem und ermöglicht so einen flexiblen Kompromiss zwischen der Sicherheit der Topologie und der Benutzerfreundlichkeit von Netzwerk-Analyse-Programmen. Um sicherzustellen, dass Angreifer nur die verschleierte Topologie herausfinden können, verändert NetHide die Pakete von Netzwerk-Analyse-Programmen in Echtzeit.

Als zweites präsentieren wir *ditto*, ein System, das den Sendzeitpunkt und die Grösse von Netzwerkpaketen verschleiert, um Analysen des Datenverkehrs zu verhindern. Die Grundidee von *ditto* besteht darin, Pakete zu vergrössern und zusätzliche Pakete einzuführen, so dass der resultierende Datenverkehr in Bezug auf Paketgrössen und Sendzeiten unabhängig vom tatsächlichen Verkehr ist. *ditto* bietet einen hohen Durchsatz, ohne dass Änderungen an den Endgeräten erforderlich sind. Daher ist *ditto* ideal geeignet, um beispielsweise Netzwerke zwischen Rechenzentern zu schützen.

Beide Systeme nutzen die jüngsten Fortschritte bei der Programmierbarkeit von Netzwerken. Die zwei Systeme zeigen, dass programmierbare Netzwerkgeräte die Sicherheit von Computernetzwerken verbessern können, ohne deren Leistung zu beeinträchtigen.

Programmierbare Netzwerkgeräte können jedoch nicht nur für die Verschleierung von Netzwerken genutzt werden, sondern sie ermöglichen auch die Analyse des Datenverkehrs in grossem Umfang. Zum Abschluss dieser Dissertation diskutieren wir vier Anwendungsfälle, in denen programmierbare Netzwerkgeräte den Datenverkehr analysieren – sowohl für gute als auch für schlechte Zwecke.

## PUBLICATIONS

---

This dissertation is based on previously published conference proceedings. The list of accepted and submitted publications is presented hereafter.

### **NetHide: Secure and Practical Network Topology Obfuscation**

Roland Meier, Petar Tsankov, Vincent Lenders,  
Laurent Vanbever, Martin Vechev.

*USENIX Security*, Baltimore, MD, USA, 2018.

### **ditto: WAN Traffic Obfuscation at Line Rate**

Roland Meier, Vincent Lenders, Laurent Vanbever.

*NDSS*, San Diego, CA, USA, 2022.

The following publications were part of my PhD research and are referenced in this thesis, but they were led by other researchers.

### **pForest: In-Network Inference with Random Forests**

Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller,  
Tobias Bühler, Laurent Vanbever.

*arXiv preprint*, arXiv:1909.05680, 2019.

### **Mass Surveillance of VoIP Calls through Programmable Data Planes**

Ege Cem Kirci, Maria Apostolaki, Roland Meier,  
Ankit Singla, Laurent Vanbever.

*ACM Symposium on SDN Research (SOSR)*, online, 2022.



The following publications were part of my PhD research, but are not covered in this dissertation.

**FeedRank: A Tamper-resistant Method for the Ranking of Cyber Threat Intelligence Feeds**

Roland Meier, Cornelia Scherrer, David Gugelmann, Vincent Lenders, Laurent Vanbever.

*International Conference on Cyber Conflict (CyCon)*, Tallinn, Estonia, 2018.

**Machine Learning-based Detection of C&C Channels with a Focus on the Locked Shields Cyber Defense Exercise**

Nicolas Känzig, Roland Meier, Luca Gambazzi, Vincent Lenders, Laurent Vanbever

*International Conference on Cyber Conflict (CyCon)*, Tallinn, Estonia, 2019.

**Detection of Malicious Remote Shell Sessions**

Pierre Dumont, Roland Meier, David Gugelmann, Vincent Lenders.

*International Conference on Cyber Conflict (CyCon)*, Tallinn, Estonia, 2019.

**(Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs**

Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, Laurent Vanbever.

*ACM HotNets*, Princeton, NJ, USA, 2019.

**Towards an AI-powered Player in Cyber Defense Exercises**

Roland Meier, Arturs Lavrenovs, Kimmo Heinäaro, Luca Gambazzi, Vincent Lenders.

*International Conference on Cyber Conflict (CyCon)*, online, 2021.

**Generalizing Machine Learning Models to Detect Command and Control Attack Traffic**

Lina Gehri, Roland Meier, Daniel Hulliger, Vincent Lenders.

*Under submission*, 2022.

## ACKNOWLEDGMENTS

---

I was lucky to be surrounded by many helpful and smart people during the journey that led to this dissertation. I am deeply grateful to all of them. Below, I mention some that deserve a special thank you for making this journey possible and enjoyable.

First and foremost, I thank Prof. Laurent Vanbever for the opportunity to pursue a doctorate in his group. His great support and guidance already during my Master's thesis were significant contributing factors for me to decide that I want to do a doctorate. I enjoyed being part of his group, and I highly appreciate that Laurent was always available when I needed his help and that he also gave me the freedom to explore research topics outside the scope of this thesis. Laurent is not only a great role model for doing research, but he also helped me improve my writing and presentation skills – and he pushed me to new personal records on Strava.

I am also extremely grateful to Vincent Lenders, my second advisor. He always provided excellent advice, feedback, and insights from the industry. This greatly improved the contents of this thesis and many other research projects. Furthermore, I thank Vincent for giving me the opportunity to collaborate with him, other people from armasuisse, and students on several projects outside the scope of this thesis.

I thank Prof. Ang Chen and Prof. Adrian Perrig for participating in my dissertation committee and for their valuable feedback that helped me improve this dissertation.

I thank all the other members of the Networked Systems Group for all their support and for the fun we had together: Maria Apostolaki, Rüdiger Birkner, Tobias Bühler, Coralie Busse-Grawitz, Yu Chen, Edgar Costa Molero, Alexander Dietmüller, Ahmed El-Hassany, Georgia Frangkouli, Albert Gran Alcoz, Thomas Holterbach, Romain Jacob, Ege Cem Kirci, Roland Schmid, Tibor Schneider, Muoi Tran, and Rui Yang. Special thanks go to Tobias for being a great office mate during all the years and for his help on countless occasions; Rüdiger and Edgar for the pleasant conversations in their office and all their help; and to Coralie, Tobias, and Ege for their feedback on drafts of this thesis.

During my doctorate, I had the pleasure of working together with many incredibly talented collaborators and students. I thank all of them for the good collaboration and for all the things I learned from them. I also thank Beat Futterknecht for his help on various administrative topics.

I am deeply grateful to my family and friends. They supported me over all the years and always provided a relaxing atmosphere in which I could unwind from the stressful aspects of the doctorate. In particular, I am thankful to my parents – Monika and Leo – and my siblings – Daniela and Manuel – as well as their partners for all the joyful and relaxing times.

Last but not least, I thank Coralie for her unlimited readiness to help, for her incredible patience, and for always cheering me up when I needed it.

Roland Meier  
November 2022

# CONTENTS

---

1	Introduction	1
2	Background	5
2.1	Packet-switching networks . . . . .	5
2.2	Network programmability . . . . .	9
2.3	Network obfuscation . . . . .	13
3	Obfuscating network topologies	17
3.1	Model . . . . .	20
3.2	NetHide . . . . .	25
3.3	Generating secure topologies . . . . .	27
3.4	Topology deployment . . . . .	32
3.5	Evaluation . . . . .	38
3.6	Frequently asked questions . . . . .	49
3.7	Related work . . . . .	51
3.8	Conclusion . . . . .	54
4	Obfuscating network traffic	55
4.1	Model . . . . .	58
4.2	ditto . . . . .	61
4.3	Computing efficient traffic patterns . . . . .	64
4.4	Traffic shaping in the data plane . . . . .	65
4.5	Security analysis and limitations . . . . .	67
4.6	Implementation . . . . .	70
4.7	Evaluation . . . . .	73
4.8	Related work . . . . .	90
4.9	Conclusion . . . . .	93
5	De-obfuscating traffic and users	95
5.1	Case study: Proxy server detection . . . . .	97

5.2	Background on proxy servers . . . . .	99
5.3	Model . . . . .	100
5.4	Design overview . . . . .	102
5.5	Extracting features in the data plane . . . . .	107
5.6	Identifying proxies in the control plane . . . . .	107
5.7	Evaluation . . . . .	109
5.8	Discussion and future work . . . . .	116
5.9	Conclusion . . . . .	120
6	Conclusion and outlook . . . . .	121
6.1	Conclusion . . . . .	121
6.2	Open research problems . . . . .	122
	Bibliography . . . . .	125
	Own publications . . . . .	125
	References . . . . .	126

## INTRODUCTION

---

Seemingly innocent actions such as recording a workout to track one's training progress or ordering pizza during a stressful working day can impair national security, as these two examples show: Recorded workout routes – even if anonymized, aggregated, and published only as a heatmap – can reveal the locations of military bases [12]. And pizza deliveries to the White House and the Pentagon can predict when something important is about to happen [13, 14].

The main reason why these so-called side-channel attacks work is that their attack vectors were not considered when the systems were designed. This also happened to computer networks and the Internet, where the initial creators could not foresee the dimensions that these networks have today.

In 1969, the “Internet” consisted of four nodes and provided 50 kbit/s bandwidth between them [15, 16]. The network – created by the Advanced Research Projects Agency (ARPA) in the United States’ Department of Defense – was called ARPANET. It was the first wide-area packet switching network with a distributed routing algorithm.

Even though the term “Internet” did not exist and the dimensions of this network were vastly different from what we call Internet today, the technology behind ARPANET was surprisingly similar to today’s Internet: It used packet switching (as opposed to circuit switching, which was prevalent at this time), it was a fully distributed system, and it implemented the TCP/IP protocol suite [17].

All of this is still the case in today’s Internet and most of the individual networks that compose it. However, today there are not four but rather 40 billion connected devices [18], these devices do not belong to four organizations but rather to billions of people [19] and the bandwidth available to each of these users is not in the order of kilobits per second but more than 100 megabits per second on average [19].

With this growth, the threat landscape has changed drastically too. In the early days of computer networks, the few users knew and trusted each other [20]. There was no reason to design protocols in a way that prevents

malicious actions. Today, several decades later, there is an arms race between attackers and defenders in the cyber world. Some of the outcomes related to network security include that firewalls and other packet inspection devices prevent malicious traffic from reaching its destination [21]; applications encrypt their traffic before it crosses the network [22]; and people use anonymity networks or proxies to conceal their identities [23].

Unfortunately, it turns out that this is not enough. For example, attackers can interrupt network links to impair users even if no malicious traffic reaches them; and attackers can use traffic metadata to reconstruct user activities even if the traffic is encrypted.

Or, to come back to the initial examples: the workout routes leak information even if they do not say who was training; and pizza delivery still leaks information, even if the type of pizza is not visible.

In this dissertation, we present techniques to fix such weaknesses through obfuscation. In other words, we change the way the network and its traffic “looks” to an attacker such that she cannot perform her attacks.

To use the examples one last time: Our systems would modify the workout routes to hide the real hotspots, and they would hide the pizza consumption by ensuring that number of daily pizza deliveries is constant.

But defending against attackers is not the only challenge that arose since the early days of the Internet. One other big challenge is that today’s networks and the Internet carry vastly more traffic compared to a few decades ago. For example, the global Internet traffic in 2022 was estimated to be around 150 terabytes per second [24]. This makes it more and more challenging to implement security solutions that can keep up with the traffic volume without downgrading the network performance.

Traditionally, devices that process network traffic are either fast but very restricted in their functionality (e.g., switches that process traffic in ASICs), or slow but very flexible (e.g., general-purpose servers that process packets using their CPUs). This inflexibility of traditional networking hardware frustrated not only the developers of security solutions but also network operators and researchers in general. And it resulted in network devices becoming both fast and flexible in the past years.

First (in 2008), researchers presented an abstraction of the control- and data planes in network devices. This opened the control plane for custom applications which could interact with the data plane through a standardized protocol [25]. Then (in 2014), researchers presented a programming

language for the data plane of network devices, which opened the data plane for custom algorithms too.

Today – as a result of these initiatives – there are network devices that are fully programmable and yet achieve the same performance as traditional fixed-function devices [26].

Even though these devices are still limited in terms of the available resources (e.g., little memory) and operations (e.g., no floating point), they have already enabled many research works. Examples include improved network monitoring (e.g., [27–29]), better traffic management (e.g., [30–32]), protections against various attacks (e.g., [33–35]), and many more [36].

In this dissertation, we leverage these advances in network programmability to make our obfuscation systems not only secure but also highly performant. And we show that the resources and operations available on these devices are enough to implement in-network security solutions.

In summary, this dissertation focuses on the following research question:

*How can obfuscation and data-plane programmability increase the security of networks without degrading their performance?*

We answer by introducing two systems:

Our first system, *NetHide*, showcases how programmable networks can prevent link-flooding attacks. For these attacks, the attacker needs to (partially) know the network topology. The key insight behind our work is that programmable networks can respond to path tracing queries in a way that maintains the utility of these tools for benign purposes but prevents an attacker from learning sensitive information about bottleneck links.

Our second system, *ditto*, showcases how programmable networks can prevent traffic-analysis attacks. For these attacks, the attacker only needs to have access to packet metadata (such as packet sizes, timestamps, and directions). The key insight behind our work is that programmable networks can obfuscate these metadata by mixing real and chaff packets in a way that provides both high security and high performance.

Both systems show that obfuscation complements existing security measures (such as traffic encryption) and helps to improve the security of a network. Both systems also show that programmable networks are a valuable tool for implementing network obfuscation techniques for high-throughput networks.



*Outline* The rest of this dissertation is structured as follows.

In Chapter 2, we introduce the required background knowledge about computer networks and network programmability.

Afterwards, we present two use cases where network programmability allows to increase security through obfuscation in a way that was not possible before: In Chapter 3, we present NetHide, our solution to mitigate link-flooding attacks by obfuscating the network topology. In Chapter 4, we present *ditto*, our solution to mitigate traffic-analysis attacks by obfuscating the size and timestamps of packets.

In Chapter 5, we broaden our scope and present ways in which programmable networks can also help to de-obfuscate network properties at scale. We first sketch three systems that enable network operators – with benign or malicious intents – to identify participants of VoIP calls, perform traffic classification using machine learning models, and perform traffic-analysis attacks. Then, we present a detailed case study that shows how programmable networks can identify and identify proxy servers in an ISP network.

Finally, in Chapter 6, we conclude and sketch opportunities for future research.

## BACKGROUND

---

In this chapter, we provide the necessary background information for this dissertation. We first explain the basics of computer networks (Section 2.1) and what the “programmable” networks changed (Section 2.2). Then, we explain the concept of network obfuscation and discuss its use cases and existing work (Section 2.3).

### 2.1 PACKET-SWITCHING NETWORKS

In the most general sense, the purpose of a communication network is that two parties connected to the network can exchange data. This was – and still is – the case in the analog telephone network and for all modern computer networks [37].

However, while their high-level purpose is the same, these two types of networks differ in their technical implementation. The analog telephone network is a so-called *circuit switching* network, which means data is transmitted over a direct physical connection between the sender and the receiver (i.e., the caller and the callee). On the other hand, most computer networks and the Internet are so-called packet-switching networks. In these networks, the sender breaks a message into small chunks (the packets). These packets traverse the network individually while sharing the physical connections with packets from other senders and receivers. The receiver then assembles the packets back into the original message.

#### 2.1.1 *Types of networks*

Depending on their geographical spread, computer networks are classified as personal area networks, local area networks, metropolitan area networks, wide area networks or internetworks [37]. In this dissertation, local area networks, wide area networks, and internetworks play important roles and are thus explained further below.

**Local Area Networks (LANs)** typically cover one room, building, or campus. They connect resources such as personal computers, servers, or printers. And they provide a gateway to larger networks (such as wide area networks).

**Wide Area Networks (WANs)** connect multiple LANs over large geographical distances. Two prominent examples of WANs are those that connect multiple sites (e.g., datacenters or campuses) of the same organization and those that connect many LANs to the Internet (the Internet Service Provider (ISP) networks).

**Internetworks** connect networks operated by different organizations. The most widely known instantiation of an internetwork is the Internet which connects over 70 000 networks (“Autonomous Systems”, or ASes) [38].

### 2.1.2 Sending packets through networks

**Packet format** Since the sender and the receiver might be in different networks, there is a need for common conventions and protocols across networks. Today’s computer networks and the protocols used in them can be modeled best by defining five layers (L1–L5) [37]:

- L1: The *physical layer* is responsible for transmitting individual bits across different carriers (e.g., wires or radio waves).
- L2: The *link layer* is responsible for transmitting finite-length chunks of data over one link.
- L3: The *network layer* is responsible for transmitting finite-length chunks of data over multiple links and potentially multiple networks.
- L4: The *transport layer* is responsible for providing reliability properties and abstractions for sending byte-streams of infinite lengths between two hosts.
- L5: The *application layer* is responsible for the interface to applications and processes which communicate over the network (e.g., web browsers).

Typically, each packet contains information on all these layers and different devices process information in different layers. While the source and destination hosts parse all layers, intermediate devices only need information up to L2 or L3 to forward the packet along the right path.

**Forwarding packets through a network** In order to connect many hosts to a network without requiring direct connections between each pair of them, switches and routers relay packets from multiple sources to multiple destinations.

*Switches* operate on the link layer (L2). To know where to send a packet, they parse the link-layer destination address (e.g., the destination MAC address in Ethernet) and query their so-called forwarding table, which contains data about which egress port the packet needs to be sent to.

*Routers* are similar to switches, but they additionally operate on the network layer (L3). This allows them to route packets across the boundaries of a local network. Similar to a switch, they parse the destination address of the network layer (e.g., IP) and use their knowledge in the so-called forwarding information base (FIB) to determine the egress port of a packet.

To determine the contents of the switches' forwarding table and the routers' FIB, there exists a variety of algorithms and protocols (e.g., MAC learning and spanning tree protocol for L2 [39, 40] and routing protocols for L3 [41, 42]). We do not go into the details of these protocols because they are not relevant for this dissertation.

### 2.1.3 Important protocols

In this section, we describe the protocols relevant for the systems presented in this dissertation.

**Ethernet** is the predominant link-layer protocol used in today's local- and wide area networks. It was standardized in 1983 as IEEE 802.3 [37]<sup>1</sup>. A data unit transmitted by Ethernet is called a *frame* and it contains source and destination MAC (Media Access Control) addresses, the type of the next-higher (L3) protocol and a frame check sequence [43].

**Internet Protocol (IP)** is the predominant network-layer protocol in today's local networks as well as internetworks (including the Internet). Today, two different versions of IP are in use: IPv4, which is the predominant version [44] and is in use since 1980 [45], and IPv6, the new version which addresses some limitations of IPv4 [46]. Similar to Ethernet, IP specifies the source and destination address of a packet in its header and routers use this

---

<sup>1</sup> Originally, *classic* Ethernet was designed for communication over a shared medium, *switched* Ethernet – the variant that is used today – is designed for packet switching networks

information to route a packet through (potentially) multiple networks. In contrast to Ethernet addresses, IP addresses have a hierarchical format. The first part of the address (often the first 16 or 24 bits of a 32-bit IPv4 address) are called the *prefix*. This format allows routers to forward packets based on their prefix (instead of the full destination address), which significantly reduces the size of the FIB. Besides the source and destination addresses, the IP header includes information such as the packet length, the protocol that is used in the next-higher layer, and a “time to live” (TTL) value that specifies how many routers a packet can pass at most (in order to prevent infinite loops).<sup>2</sup>

***Internet Control Message Protocol (ICMP)*** is – in contrast to e.g., TCP and UDP, which we cover below – not used to transport data between hosts but it is a control protocol that allows routers to signal when a packet cannot be processed regularly. For this, ICMP defines 13<sup>3</sup> message types to notify the sender about events such as an unreachable destination, an invalid header, or an expired TTL value [47]. Several network debugging tools make use of ICMP features. Most prominently, this includes *ping*, a tool to measure the round trip time to a given device (it sends an ICMP ECHO REQUEST message and measures the time until the destination answers with ECHO REPLY) [48] and *traceroute*, a tool to determine the path that packets take through the network (it sends IP packets with increasing TTL values and leverages the fact that routers respond with ICMP TIME EXCEEDED when a packet’s TTL expires) [49].

***User Datagram Protocol (UDP)*** is a transport-layer protocol for unreliable data transfer [50]. Unreliable means that UDP does not provide congestion control or retransmission of lost packets (in contrast to TCP, see below). The key strength of UDP is that it provides a fast way to send data because it does not require establishing a connection first. UDP uses port numbers to specify the source and the destination “socket” within the sending and receiving device. Applications can open these sockets in order to send or receive network data. In principle, every application can open any socket, but by convention, the port numbers below 1024 should be used by their assigned applications only (e.g., port 80 for HTTP) [51].

***Transmission Control Protocol (TCP)*** is a transport-layer protocol for reliable data transfer [52]. Similar to UDP, it uses port numbers to trans-

---

<sup>2</sup> The field names are for IPv4, but similar fields exist in IPv6

<sup>3</sup> not including deprecated, experimental, or reserved types

fer data between the sender's and receiver's respective sockets. However, unlike UDP, TCP is connection-oriented. A TCP connection starts with a handshake before it can be used to transfer data. Within this connection, TCP monitors packet loss and reacts to network conditions (such as the available bandwidth): it automatically retransmits lost packets and it adapts the transmission rate depending on the network conditions.

**Transport Layer Security (TLS)** is an application-layer protocol that protects its payloads against eavesdropping, modifications, and injected messages [53]. To establish a TLS connection, the client and the server exchange handshake packets that contain cryptographic parameters and certificates. TLS is widely used to encrypt HTTP traffic (introduced below) to allow secure web browsing. In this case, only the server provides a certificate to prove that it is authorized to host the requested website. However, in other cases, TLS can be used to authenticate clients too.

**Hypertext Transfer Protocol (HTTP)** is an application-layer protocol for fetching resources over the network [54]. Most prominently, it is used to load data (e.g., websites) over the World Wide Web. It is designed as a request-response protocol where the client requests a resource (e.g., a website) from a server and the server returns the resource (i.e., the website's source code). HTTP can run on top of a transport layer protocol (usually TCP) directly, or it can run on top of TLS for secure connections. HTTP that runs over TLS is called HTTPS. Today, most popular websites support HTTPS [55].

## 2.2 NETWORK PROGRAMMABILITY

In this section, we explain the architecture of network devices (routers and switches), how these devices are programmable and what the strengths and limitations of programmable network devices are.

### 2.2.1 Network device architecture

At a high level, network devices (routers and switches) consist of three building blocks (cf. Figure 2.1): The data, control, and management planes [56].

The *data plane* consists of the physical ports which receive and send packets and the logic for forwarding packets between them (including

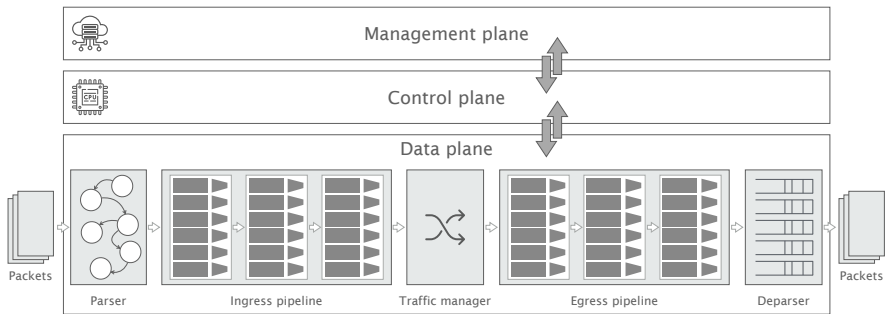


FIGURE 2.1: Architecture of a (programmable) switch. It consists of three planes: the management plane (for configuration), the control plane (for forwarding decisions), and the data plane (for handling packets).

buffering, scheduling, and some packet header modifications). The data plane is typically implemented in an application-specific integrated circuit (ASIC) that can process packets at line rate. For good performance, the vast majority of packets should be processed in the data plane. However, the data plane is usually optimized to apply simple actions to many packets in a short time. As a result, there are packets that the data plane cannot handle itself. Examples of such packets typically include routing protocol information or packets whose destination is not in the forwarding table.

The *control plane* typically runs on a general-purpose CPU. Thus, it provides high flexibility at the cost of packet processing performance. It can run more complex algorithms (e.g., to compute shortest paths) and it can update the data plane accordingly (e.g., through forwarding table entries).

The *management plane* provides an interface to network administrators for configuration and monitoring. It then configures the algorithms in the control plane accordingly. The management plane provides the highest flexibility since it can run on a general-purpose server and it does not interact with network packets directly.

Over time, the fact that only the management plane allowed a network administrator to configure her network became a limitation. This led to the desire for a more flexible control plane.

### 2.2.2 Programmability in the control plane

In 2008, McKeown et al. published an editorial note [25] in which they described the concept of a programmable control plane, an extended flow table, and a standardized interface between the two.

This allowed network operators (and researchers) (i) to develop their own control-plane applications; (ii) to use a central controller for multiple data planes (i.e., multiple switches); and (iii) to combine hardware of different vendors.

An OpenFlow-compatible switch (OpenFlow is the name of the protocol between the control plane and the data plane) provides a flow table with a given set of properties and allows modifications to the table entries through the OpenFlow protocol. Now, the biggest constraint for custom applications was the format of the flow table (i.e., the packet header fields that can be read and modified and the possible actions that can be executed). While an OpenFlow-compatible switch can typically perform all the actions that a traditional switch can (e.g., forward packets based on their destination MAC or IP address), it does not allow innovation in the form of new network- or link-layer protocols or data-plane algorithms.

To overcome this limitation of OpenFlow, the data plane had to become programmable too.

### 2.2.3 Programmability in the data plane

To enable innovation not only in the control plane but also in the data plane, Bosshart et al. introduced P4, a high-level programming language for “protocol-independent packet processors” [57]. Soon after, the first fully programmable switch was available on the market [26].

The so-called portable switch architecture (PSA) [58] describes a template for the architecture of programmable switches. Such switches allow running custom programs (implemented in P4) in the data plane and can process traffic at terabits per second [59].

At a high level, programmable switches process packets as follows (cf. illustration in Figure 2.1). When a packet arrives, a *parser* extracts information from the packet headers. These headers (together with metadata such as the ingress port) then traverse the *ingress* pipeline. There, match & action tables can modify the packet headers and metadata (e.g., set the egress port).



Afterwards, the packet arrives at the *traffic manager (TM)*, which (among others) sends the packet to an egress pipeline. The *egress pipeline* works like the ingress pipeline, except that it is attached to the packet's egress port. At the end of the egress pipeline, the *deparser* assembles the (potentially modified) headers and the (unmodified) payload back to a packet and transmits it.

Below, we provide more details about each of these building blocks.

The *parser* receives the incoming packet and extracts headers. The format of these headers is programmable (i.e., the parser can extract custom header formats). Only the parsed parts of the packet are accessible in the pipelines. The rest of the packet is considered as payload and cannot be modified.

The *ingress pipeline* receives the packet's headers together with metadata (e.g., its ingress port), which is all stored in the packet header vector (PHV). The pipeline consists of several stages in which match & action tables can match on data in the PHV and trigger actions to modify it.

The architecture and the focus on processing packets at line rate impose three main limitations concerning the ingress and egress pipeline: (i) the number of pipeline stages limits the number of sequential actions that can be performed on each packet; (ii) the size of the PHV limits the size of the parsed headers and metadata (which can be seen as local variables); and (iii) operations which take a non-constant time per packet are not possible (e.g., loops, splitting or merging packets).

The *Traffic Manager (TM)* switches packets from ingress pipelines to egress pipelines. If needed, the TM buffers packets in first-in, first-out (FIFO) queues. When the egress pipeline can process the next packet, the TM selects a queue and sends its next packet to the egress pipeline. To determine the queue, the TM can use different strategies [60]: (i) the queue's priorities; (ii) weighted round-robin; or (iii) a combination of both (round robin among queues with equal priorities).

The *egress pipeline* is identical to the ingress pipeline except that it is attached to an egress port. As a consequence, it is, for example, no longer possible to change a packet's egress port once the packet has passed the TM. Similarly, the *deparser* is the inverse of the parser: It takes the headers and the payload and assembles the final packet.

### 2.2.4 Strengths of programmable networks

In contrast to traditional networks where each device (routers, switches, firewalls, . . .) serves one fixed purpose and is only *configurable* by the operator (e.g., they can manage subnets, VLANs, or firewall rules), *programmable* network devices can run any data-plane program as long as it meets their resource constraints. This leads to the following advantages:

- *Flexibility*: Programmable network devices can be adapted to new protocols (e.g., future versions of IP) and one device (e.g., a switch) can run multiple applications tailored to the given network and the threat landscape. The operator can program the switch according to their needs.
- *Visibility*: Since the programs run in the data plane, they can interact with every packet that crosses the device without introducing traffic overhead (e.g., traditionally, the traffic had to be cloned to an other device).
- *Performance*: Data-plane programs process packets at line rate and they can update their state or take decisions instantly.

While the concept of programmable data planes was new when the research for this dissertation started in 2017, there exist many systems that show the usefulness of this concept for security applications in the meantime.

Among others, researchers have shown that programmable switches can be used to implement more flexible firewalls [33, 61–63]; new authentication schemes [64–69]; defenses against various attacks [70–85] and many more applications (cf. surveys in [36, 86]).

Since this dissertation focuses on using programmable networks for obfuscation, we mainly consider related work in this area. In the next section, as well as in Chapters 3 and 4, we discuss such work.

## 2.3 NETWORK OBFUSCATION

Obfuscation generally means making something difficult or impossible to understand [87]. In this section, we discuss how obfuscation is helpful in computer networks. Concerning computer networks, there are four main use cases for obfuscation:

- Protocol obfuscation to avoid censorship
- Packet header obfuscation to increase anonymity
- Topology obfuscation to improve resilience
- Traffic obfuscation to prevent information leakage

In the following paragraphs, we will explain each use case in more detail and summarize corresponding related work.

**Protocol obfuscation** The main application of network protocol obfuscation systems is to circumvent filtering or censorship. Protocol obfuscation systems typically achieve this by using randomization, mimicry, or tunneling. *Randomization* systems (e.g., [88–90]) randomize the traffic such that it is not possible to extract information (e.g., the used application) from it. *Mimicry* systems (e.g., [91, 92]) modify traffic such that it contains signatures of popular (non-blocked) protocols. For example, such systems embed strings that match regular expressions of deep packet inspection boxes. *Tunneling* systems (e.g., [93–98]) go one step further and embed the obfuscated traffic in a cover protocol (e.g., in HTTPS).

While most systems fall in one of these three categories, some systems are programmable to use a combination of these techniques (e.g., [99, 100]).

**Header obfuscation** Several systems use obfuscation to hide packet headers as an alternative to network-layer anonymity systems such as TOR [101], LAP [102], Dovetail [103], HORNET [104] PHI [105], and TARANET [106].

Below, we summarize four systems that leverage network programmability to obfuscate packet headers.

In previous work, we presented iTAP [5], a system to obfuscate packet headers within a local network using OpenFlow-compatible switches. The main idea behind iTAP is to replace the source and destination addresses with pseudo-random values. Some bits of these pseudo-random values represent the actual sender and the receiver, but an eavesdropper does not know which bits. Similarly, Lee et al. developed two systems [107, 108] that assign per-packet one-time addresses to its customers’ hosts. In contrast to iTAP, these systems are designed to operate between multiple ASes.

SPINE [109] obfuscates packet header fields (IP addresses and TCP fields) in traffic between two participating ASes. To be compatible with the operations available on programmable switches, SPINE uses one-time-pad-based encryption to obfuscate the header fields.

PINOT [110] also obfuscates the IP addresses of traffic that leaves a trusted AS. But in contrast to SPINE, PINOT runs at the edge of one AS and does not need the cooperation of multiple ASes.

MIMIQ [111] leverages the connection migration capability of QUIC [112] to change the client's IP address every few packets. Like PINOT, it runs at the edge of a trusted network and ensures that potential eavesdroppers outside of this network cannot infer the client that sent a packet.

**Topology obfuscation** Even though most network operators do not publish their topology, it is possible to determine it by using tools such as traceroute [49]. Unfortunately, knowledge of the topology can help an attacker to launch so-called link-flooding attacks [113, 114]. To prevent such attacks, there exist many systems to obfuscate a network's topology.

In Chapter 3, we will present NetHide, our system for topology obfuscation. The main idea behind NetHide and many related works (e.g., [115–118]) is to modify the responses that traceroute produces in such a way that the presented network topology does not show the bottlenecks in the real topology. The main challenges for these works are to determine a topology that does not contain sensitive information but is still realistic and to modify responses to path tracing tools in a way that the attacker cannot notice. For a detailed discussion of related work in the area of topology obfuscation, we refer to Section 3.7.

**Traffic obfuscation** The main reason for traffic obfuscation is to prevent so-called traffic-analysis attacks that can infer details about ongoing activities only based on packet sizes, directions, and timings [119–134].

In Chapter 4, we will present ditto, our system for traffic obfuscation. ditto and many related works (e.g., [104, 106, 134–138]) obfuscate traffic by adding padding to packets (to obfuscate their size) and by introducing chaff packets (to obfuscate the number and timing of packets). The main challenge for these works is to do these operations in a way that does not allow an attacker to identify padding or chaff packets and provides high performance. For a detailed discussion of related work in the area of traffic obfuscation, we refer to Section 4.8.



## OBFUSCATING NETWORK TOPOLOGIES TO PREVENT SOPHISTICATED DENIAL-OF-SERVICE ATTACKS

---

In this chapter, we present NetHide, a network topology obfuscation framework that mitigates sophisticated distributed denial-of-service attacks while preserving the practicality of network debugging tools.

Botnet-driven Distributed Denial-of-Service (DDoS) attacks constitute one of today’s major Internet threats [139, 140]. Such attacks can be divided in two categories depending on whether they target end hosts and services (volume-based attacks) or the network infrastructure itself (link-flooding attacks, LFAs).

Purely volume-based attacks are the simplest and work by sending massive amounts of data to selected targets. Recent examples include the 1.2 Tbps DDoS attack against Dyn’s DNS service [141] in October 2016 and the 1.35 Tbps DDoS attack against GitHub in February 2018 [142]. While impressive, these attacks can be mitigated today by diverting the incoming traffic through large CDN infrastructures [143]. As an illustration, CloudFlare’s infrastructure can now mitigate volume-based attacks reaching Terabits per second [144].

Link-flooding attacks (LFAs) [113, 114] are more sophisticated and work by having a botnet generate low-rate flows between pairs of bots or towards public services such that all of these flows cross a given set of network links or nodes, degrading (or even preventing) the connectivity for *all* services using them. LFAs are much harder to detect as: (i) traffic volumes are relatively small (10 Gbps or 40 Gbps attacks are enough to kill most Internet links [145]); and (ii) attack flows are indistinguishable from legitimate traffic. Representative examples include the Spamhaus attack which flooded selected Internet eXchange Point (IXP) links in Europe and Asia [146–148].

Unlike volume-based attacks, performing an LFA requires the attacker to know the topology *and* the forwarding behavior of the targeted network. Without this knowledge, an attacker can only “guess” which flows share a common link, considerably reducing the attack’s efficiency. As an illustration, our simulations indicate that congesting an *arbitrary* link without

knowing the topology requires 5 times more flows, while congesting a *specific* link is orders of magnitude more difficult.

Nowadays, attackers can easily acquire topology knowledge by running path tracing tools such as traceroute [49]. In fact, previous studies have shown that entire topologies can be precisely mapped with traceroute provided enough vantage points are used [149], a requirement easily met by using large-scale measurement platforms (e.g., RIPE Atlas [150]).

**Existing works** Existing LFA countermeasures either work *reactively* or *proactively*. Reactive measures dynamically adapt how traffic is being forwarded [151, 152] or have networks collaborating to detect malicious flows [145]. Proactive measures work by obfuscating the network topology so as to prevent attackers from discovering potential targets [115–117]. The problem with reactive countermeasures is the relative lack of incentives to deploy them: collaborative detection is only useful with a significant amount of participating networks, while dynamic traffic adaptation conflicts with traffic engineering objectives. In contrast, proactive approaches can protect each network individually without impacting normal traffic forwarding. Yet, they considerably lower the usefulness of path tracing tools [115, 117] such as traceroute which is the prevalent tool for debugging networks [149, 153, 154]. Further, they also provide poor obfuscation which can be broken with a small number of brute-force attacks [116, 117].

**Problem statement** Given the limitations of existing techniques, a fundamental question remains open: *is it possible to obfuscate a network topology so as to mitigate attackers from performing link-flooding attacks while, at the same time, preserving the usefulness of path tracing tools?*

**Key challenges** Answering this question is challenging for at least three reasons:

1. The topology must be obfuscated with respect to any possible attacker location: attackers can be located anywhere and their tracing traffic is often indistinguishable from legitimate user requests.
2. The obfuscation logic should not be invertible and should scale to large topologies.
3. The obfuscation logic needs to be able to intercept and modify tracing traffic at line rate. To preserve the troubleshooting-ability of network operators, tracing traffic should still flow across the correct physical

links such that, for example, link failures in the physical topology are visible in the obfuscated one.

**NetHide** We present NetHide, a novel network obfuscation approach which addresses the above challenges. NetHide consists of two main components: (i) a usability-preserving and scalable obfuscation algorithm; and (ii) a runtime system, which modifies tracing traffic directly in the data plane.

The key technical insight behind NetHide is to formulate the network obfuscation task as a multi-objective optimization problem that allows for a flexible trade-off between security (encoded as hard constraints) and usability (soft constraints). We introduce two metrics to quantify the usability of an obfuscated topology: *accuracy* and *utility*. Intuitively, the accuracy measures the similarity between the path along which a flow is routed in the physical topology with the path that NetHide presents in the virtual topology. The utility captures how physical events (e.g., link failures or congestion) in the physical topology are represented in the virtual topology. To scale, we show that considering only a few randomly selected candidate topologies, and optimizing over those, is enough to find secure solutions with near-optimal accuracy and utility.

We fully implemented NetHide and evaluated it on realistic topologies. We show that NetHide is able to obfuscate large topologies ( $> 150$  nodes) with marginal impact on usability. In fact, we show in a case study that NetHide allows to precisely detect the vast majority ( $> 90\%$ ) of link failures. We also show that NetHide is useful when partially deployed: 40% of programmable devices allow to protect up to 60% of the flows.

**Contributions** Our main contributions are:

- A novel formulation of the network obfuscation problem in a way that preserves the usefulness of existing debugging tools (Section 3.2).
- An encoding of the obfuscation task as a linear optimization problem together with a random sampling technique to ensure scalability (Section 3.3).
- An end-to-end implementation of our approach, including an online packet modification runtime (Section 3.4).
- An evaluation of NetHide on representative network topologies. We show that NetHide can obfuscate topologies of large networks in a reasonable amount of time. The obfuscation has little impact on benign users and mitigates realistic attacker strategies (Section 3.5).



### 3.1 MODEL

We now present our network and attacker models and formulate the precise problem we address.

#### 3.1.1 *Network model*

We consider layer 3 (IP) networks operated by a single authority, such as an Internet service provider or an enterprise. Traffic at this layer is routed according to the destination IP address. We assume that routing is deterministic, meaning that the traffic is sent along a single path between each pair of nodes. While this assumption does not hold for networks relying on probabilistic load-balancing mechanisms (e.g., ECMP [155]), it makes our attacker more powerful as all paths are assumed to be persistent and therefore easier to learn.

To deploy NetHide, we assume that some of the routers are programmable in a way that allows them to: (i) match on arbitrary IP Time-to-Live (TTL) values; (ii) change the source and destination addresses of packets (e.g., UDP packets for traceroute) depending on the original destination address and the TTL; and (iii) restore the original source and destination addresses when replies (e.g., ICMP packets) to modified packets arrive. Our implementation uses the P4 programming language [156], which fulfills the above criteria. Yet, NetHide could also be implemented on top of existing router firmware.

#### 3.1.2 *Attacker model*

We assume an attacker who controls a set of hosts (e.g., a botnet) that can inject traffic in the network. The attacker's goal is to perform a *Link Flooding Attack* (LFA) such as Coremelt [113] or Crossfire [114]. The objective of these attacks is to isolate a network segment by congesting one or more links. The attacker aims to congest links by creating low-volume flows from many different sources (bots) to many destinations (public servers or other bots) such that all these flows cross the targeted links (illustrated in Figure 3.1). An attacker's *budget* limits the number of flows she can run and we quantify the attacker's strength based on her budget. Because the additional traffic is low-volume, it is hard to separate it from legitimate

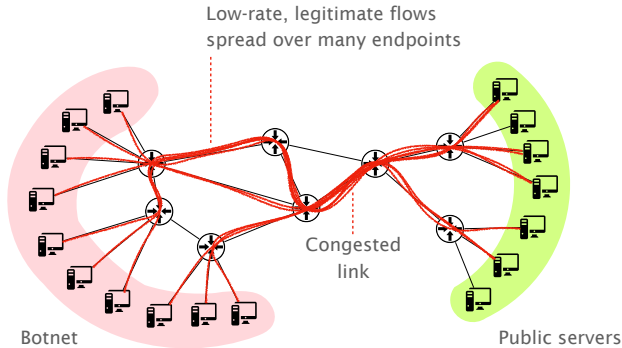


FIGURE 3.1: Link-Flooding Attacks (LFAs) work by routing many legitimate low-volume flows over the same set of physical links in order to cause congestion. LFAs assume that the attacker can discover the network topology, usually using traceroute-like tracing.

(also low-volume) traffic. This makes detecting and mitigating LFA attacks a hard problem [157].

To mount an efficient and stealthy LFA, the attacker must know enough (source, destination) pairs that communicate via the targeted link(s). Otherwise, she would have to create so many flows that she no longer remains efficient. Similarly to [113, 114], we assume the attacker has no prior knowledge of the network topology. And we assume that the attacker learns the network topology using traceroute-like tracing techniques [49]. traceroute works by sending a series of packets (probes) to the destination with increasing TTL values. In response to these probes, each router along the path to the destination sends an ICMP time exceeded message. More specifically, traceroute leverages the fact that TTL values are decremented by one at each router, and that the first router to see a TTL value of 0 sends a response to the source of the probe. For example, a packet with TTL value of 3 sent from  $A$  to  $B$  will cause the third router along the path from  $A$  to  $B$  to send an ICMP time exceeded message to  $A$ . By aggregating paths between many host pairs, it is possible to determine the topology and the forwarding behavior of the network [149]. We remark that in addition to revealing forwarding paths, traceroute-like probes also disclose the Round-Trip Time (RTT), i.e., the time difference between the moment a probe is sent and the corresponding ICMP time exceeded message is received, which can be used as a side-channel to gain intuition about the feasibility of a (potentially obfuscated) path returned by traceroute.

**Network components**

(Nodes)	$N \subseteq \mathcal{N} = \{n_1, \dots, n_N\}$
(Links)	$L \subseteq N \times N$
(Forwarding tree)	$T_n = (N, L_n)$ , tree rooted at $n$
(Forwarding trees)	$T = \bigcup_{n \in N} T_n$
(Flows)	$F \subseteq N \times N$

**Network topologies**

(Physical)	$P = (N, L, T)$
(Virtual)	$V = (N', L', T')$ $N \subseteq N'$

**Metrics**

(Flows per link)	$f(T, l) = \{(s, d) \in F \mid l \in T_d\}$
(Flow density)	$fd(T, l) =  f(T, l) $
(Capacity)	$c : L \rightarrow \mathbb{N}$
(Accuracy)	$acc : ((s, d), P, V) \mapsto [0, 1]$
(Utility)	$util : ((s, d), P, V) \mapsto [0, 1]$

FIGURE 3.2: NetHide notation and metrics

Finally, we assume that the attacker knows everything about the deployed protection mechanisms in the network (including the ones presented in this chapter) except their secret inputs and random decisions following Kerckhoff's principle [158].

## 3.1.3 Notation

We depict our notation and definitions in Figure 3.2. We model a *network topology* as a graph with nodes  $N \subseteq \mathcal{N}$ , where  $\mathcal{N}$  is the set of all possible nodes, and links  $L \subseteq N \times N$ . A *node* in the graph corresponds to a router in the network and a *link* corresponds to an (undirected) connection between two routers.

Given a node  $n$ , we use a tree  $T_n = (N, L_n)$  rooted at  $n$  to model how packets are forwarded to  $n$ . We refer to this tree as a *forwarding tree*. For simplicity, we write  $l \in T_n$  to denote that the link  $l$  is contained in the

forwarding tree  $T_n$ , i.e.,  $T_n = (N, L_n)$  with  $l \in L_n$ . We use  $T$  to denote the set of all forwarding trees.

A flow  $(s, d) \in F$  is a pair of a source node  $s$  and destination node  $d$ . Note that the budget of the strongest attacker is given by the total number  $|F|$  of possible flows. We use  $T_{s \rightarrow d}$  to refer to the path from source node  $s$  to destination node  $d$  according to the forwarding tree  $T_d$ . In the style of [114], we define the *flow density*  $fd$  for a link  $l \in L$  as the number of flows that are routed via this link (in any direction). The maximum flow density that a link can handle without congestion is denoted by the link's *capacity*  $c$ . A topology  $(N, L, T)$  is *secure* if the flow density for any link in the topology does not exceed its capacity, i.e.,  $\forall l \in L : fd(T, l) \leq c(l)$ . Note that no attacker (with any budget) can attack a secure topology as all links have enough capacity to handle the total number of flows from all the (source, destination) pairs in  $F$ .

#### 3.1.4 Problem statement

We address the following *network obfuscation problem*: Given a physical topology  $P$ , the goal is to compute an obfuscated (virtual) topology  $V$  such that  $V$  is secure and is as similar as possible to  $P$ . In other words, the goal is to deceive the attacker with a virtual topology  $V$ . For the similarity between the physical topology  $P$  and the obfuscated topology  $V$ , we refer to Section 3.2 where we present metrics which represent the *accuracy* of paths reported by traceroute and the *utility* of link failures in  $P$  being closely represented in  $V$ .

We remark on a few important points. First, if  $P$  is secure, then the obfuscation problem should return  $P$  since we require that  $V$  is as similar as possible to  $P$ . Second, for any network and any attacker, the problem has a trivial solution since we can always come up with a network that has an exclusive routing path for each (source, destination) pair. However, for non-trivial notions of similarity, it is challenging to discover an obfuscated network  $V$  that similar to  $P$ .

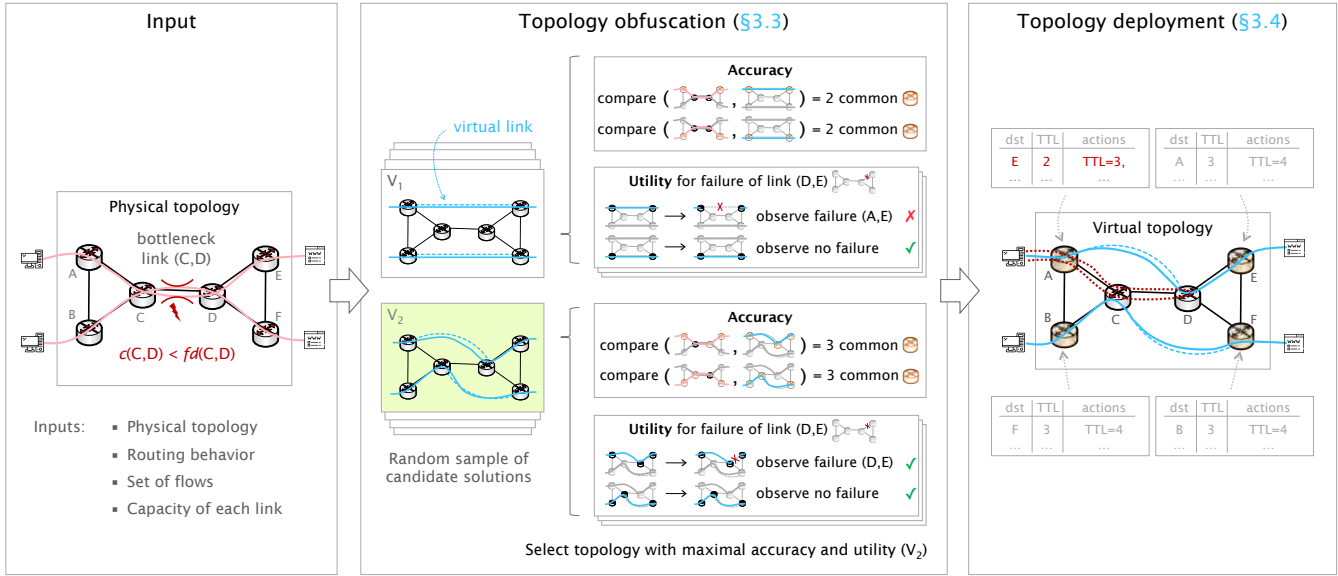


FIGURE 3.3: Nethide operates in two steps: (i) computing a secure and usable virtual topology; and (ii) deploying the obfuscated topology in the physical network.

### 3.2 NETHIDE

We now illustrate how NetHide can compute a secure and yet usable (i.e., “debuggable”) obfuscated topology on a simple example depicted in Figure 3.3. Specifically, we consider the task of obfuscating a network with 6 routers:  $A, \dots, F$  in which the core link  $(C, D)$  acts as bottleneck and is therefore a potential target for an LFA.

**Inputs** NetHide takes four inputs: (i) the physical network topology graph; (ii) a specification of the forwarding behavior (a forwarding tree for each destination according to the physical topology and incorporating potential link weights); (iii) the capacity  $c$  of each link (how many flows can cross each link before congesting it); along with (iv) the set of attack flows  $F$  to protect against. If the position of the attacker(s) is not known (the default), we define  $F$  to be the set of all possible flows between all (source,destination) pairs.

Given these inputs, NetHide produces an obfuscated virtual topology  $V$  which: (i) prevents the attacker(s) from determining a set of flows to congest any link; while (ii) still allowing non-malicious users to perform network diagnosis. A key insight behind NetHide is to formulate this task as a multi-objective optimization problem that allows for a flexible trade-off between security (encoded as hard constraints) and usability (encoded as soft constraints) of the virtual topology. The key challenge here is that the number of obfuscated topologies grows exponentially with the network size, making simple exhaustive solutions unusable. To scale, NetHide only considers a subset of candidate solutions amongst which it selects a usable one. Perhaps surprisingly, we show that this process leads to desirable solutions.

**Pre-selecting a set of secure candidate topologies** NetHide first computes a random set of obfuscated topologies. In addition to enabling NetHide to scale, this random selection also acts as a secret which makes it significantly harder to invert the obfuscation algorithm.

NetHide obfuscates network topologies along two dimensions: (i) it modifies the topology graph (i.e., it adds or removes links); and (ii) it modifies the forwarding behavior (i.e., how flows are routed along the graph). For instance, in Figure 3.3, the two candidate solutions  $V_1$  and  $V_2$  both contain two virtual links used to “route” flows from  $A$  to  $E$  and from  $B$  to  $F$ .

**Selecting a usable obfuscated topology** While there exist many secure candidate topologies, they differ in terms of *usability*, i.e., their perceived usefulness for benign users. In NetHide, we capture the usability of a virtual topology in terms of its *accuracy* and *utility*.

The *accuracy* measures the logical similarity of the paths reported when using traceroute against the original and against the obfuscated topology. Intuitively, a virtual topology with high accuracy enables network operators to diagnose routing issues such as sub-optimal routing. Conversely, tracing highly inaccurate topologies is likely to report bogus information such as traffic jumping between geographically distant points for no apparent reason. As illustration,  $V_2$  is more accurate than  $V_1$  in Figure 3.3 as the reported paths have more links and routers in common with the physical topology.

The *utility* metric measures the physical similarity between the paths actually taken by the tracing packets in the physical and the virtual topology. Intuitively, utility captures how well events such as link failures or congestion in the physical topology are observable in the virtual topology. For instance, we illustrate that  $V_2$  has a higher utility than  $V_1$  in Figure 3.3 by considering the failure of the link  $(D, E)$ . Indeed, a non-malicious user would observe the failure of  $(D, E)$  (which is not obfuscated) when tracing  $V_2$  while it would observe the failure of link  $(A, E)$  instead of  $(D, E)$  when tracing  $V_1$ .

Given  $V_1$ ,  $V_2$  and the fact that  $V_2$  has higher accuracy and utility, NetHide deploys  $V_2$ .

**Deploying the obfuscated topology** NetHide obfuscates the topology at runtime by modifying tracing packets (i.e., IP packets whose TTL expires somewhere in the network). NetHide intercepts and processes such packets without impact on the network performance, directly in the data plane, by leveraging programmable network devices. Specifically, NetHide intercepts and possibly alters tracing packets at the edge of the network before sending them to the pretended destination in the physical network. That way, NetHide ensures that tracing packets traverse the corresponding physical links, and preserves the utility of traceroute-like tools. Observe that any alteration of tracing packets is reverted before they leave the network, which makes NetHide transparent. In contrast, simpler approaches which answer to tracing packets at the network edge or from a central controller (e.g., [115, 117]) render network debugging tools unusable.

Consider again Figure 3.3 (right). If router  $A$  receives a packet towards  $E$  with  $TTL=2$ , this packet needs to expire at router  $D$  according to the virtual topology. Since the link between  $A$  and  $D$  does not exist physically, the packet needs to be sent to  $D$  via  $C$ , and it would thus expire at  $C$ . To prevent this and to ensure that the packet expires at  $D$ , NetHide increases the  $TTL$  by 1. Observe that, in addition to ensure the utility (see above), making the intended router answer to the probe also ensures that the measured round trip times are realistic (cf. Section 3.4).

### 3.3 GENERATING SECURE TOPOLOGIES

In this section, we first explain how to phrase the task of obfuscating a network topology as an optimization problem. We then present our implementation which consists of roughly 2000 lines of Python code and uses the Gurobi ILP solver [159].

#### 3.3.1 Optimization problem

Given a topology  $P = (N, L, T)$ , a set of flows  $F$ , and capacities  $c$ , the network obfuscation problem is to generate a virtual topology  $V = (N', L', T')$  such that: (i)  $V$  is secure; and (ii) the *accuracy* and *utility* metrics are jointly maximized; we define these metrics shortly.

NetHide generates  $V$  by modifying  $P$  in two ways: (i) NetHide adds *virtual links* to connect nodes in  $V$ ; and (ii) NetHide can modify the *forwarding trees* for all nodes in  $V$ .

We show the constraints that encode the security and the objective function that captures the closeness in terms of accuracy and utility in Figure 3.4 and explain them below.

**Security constraints** The main constraint is the *security* (C1) imposed on  $V$ . This being a hard constraint (as opposed to be part of the objective function) means that if NetHide finds a virtual topology  $V$ , then  $V$  is secure with respect to the attacker model and the capacities.

To ensure that the virtual topology  $V$  is valid, NetHide incorporates additional constraints capturing that: (C2) all physical nodes in  $N$  are also contained in the virtual topology with nodes  $N'$ ; (C3) there is exactly one



**Objective function**

$$\max_V \sum_{f \in F} (w_{acc} \cdot acc(f, P, V) + w_{util} \cdot util(f, P, V)) \quad \text{where} \quad \begin{aligned} w_{acc} &\in [0, 1] \\ w_{util} &\in [0, 1] \\ w_{acc} + w_{util} &= 1 \end{aligned}$$

**Hard constraints**

$$(Security) \quad \forall l \in L' : fd(V, l) \leq c(l) \quad (C1)$$

$$(Complete) \quad n \in N \Rightarrow n \in N' \quad (C2)$$

$$(Reach) \quad \forall n \in N' : |\{T_n | T_n \in T'\}| = 1 \quad (C3)$$

$$\forall T \in T' : \forall l \in T : l \in L' \quad (C4)$$

$$(n, n') \in L' \Rightarrow \{n, n'\} \in N' \quad (C5)$$

FIGURE 3.4: NetHide optimization problem. NetHide finds a virtual topology that is secure and has maximum accuracy compared with the physical topology.

virtual forwarding tree for each node; and (C4-5) links and nodes in the virtual forwarding trees are contained in  $N'$ .

**Objective function** The objective of NetHide is to find a virtual topology that maximizes the overall accuracy (cf. Section 3.3.2) and utility (cf. Section 3.3.3). As shown in Figure 3.4, we define the overall accuracy and utility as a weighted sum of the accuracy and utility values of all flows in the network.

### 3.3.2 Accuracy metric

The accuracy metric is a function that maps two paths for a given flow to a value  $v \in [0, 1]$ . In our case, this value captures the similarity between a path  $T_{s \rightarrow d}$  in  $P$  for a given flow  $(s, d)$  and the (virtual) path  $T'_{s \rightarrow d}$  for the same flow  $(s, d)$  in  $V$ . Formally, given a flow  $(s, d)$ , the accuracy is defined as:

---

**Algorithm 3.1:** Utility metric. It incorporates the likelihood that a failure in the physical topology  $P$  is visible in the virtual topology  $V$  and that a failure in  $V$  actually exists in  $P$ . Note that we treat  $T_{s \rightarrow d}$  as a set of links.

---

**Input:** Flow  $(s, d) \in F$ ,  
 Physical topology  $P = (N, L, T)$ ,  
 Virtual topology  $V = (N', L', T')$

**Output:** Utility  $u \in [0, 1]$

---

```

1 for  $n \in T'_{s \rightarrow d}$  do
2    $C \leftarrow T_{s \rightarrow n} \cap T'_{s \rightarrow d}[0 : n]$            // common links
3    $u_n \leftarrow \frac{1}{2} \left( \frac{|C|}{|T_{s \rightarrow n}|} + \frac{|C|}{|T'_{s \rightarrow d}[0:n]|} \right)$  // utility
4  $u \leftarrow \frac{1}{|T'_{s \rightarrow d}|} \sum_{n \in T'_{s \rightarrow d}} u_n$  // average

```

---

$$acc((s, d), P, V) = 1 - \frac{LD(T_{s \rightarrow d}, T'_{s \rightarrow d})}{|T_{s \rightarrow d}| + |T'_{s \rightarrow d}|}$$

Where  $LD(T_{s \rightarrow d}, T'_{s \rightarrow d})$  is Levenshtein distance [160] and  $|T_{s \rightarrow d}|$  denotes the length of the path from  $s$  to  $d$ .

The overall accuracy of a topology (as referred to in Section 3.5) is defined as the average accuracy over all flows in  $F$ :

$$Avg(P, V) = avg_{(s,d) \in F} acc((s, d), P, V)$$

We point out that the accuracy metric in NetHide can also be computed by any other function to precisely represent the network operator's needs.

### 3.3.3 Utility metric

While the accuracy measures the similarity between the physical and virtual paths for a given flow, the utility measures the representation of physical events, such as link failures. For our implementation, we design the utility metric such that it computes the probability that a link failure in the physical path is observed in the virtual path and the probability that a failure reported in the virtual path is indeed occurring in the physical path.

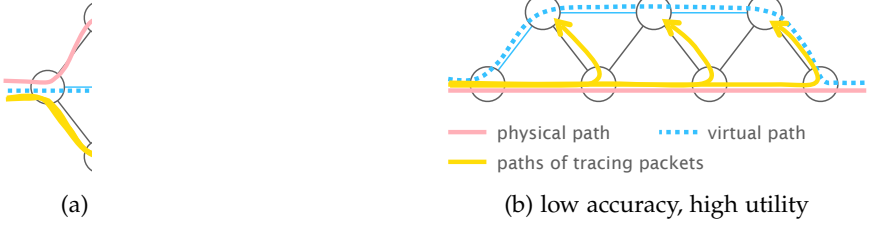


FIGURE 3.5: High accuracy does not always imply high utility (and vice-versa). In Figure 3.5a, the physical and virtual paths are similar but the tracing packets do not cross the physical links. In Figure 3.5b, the physical and virtual paths are dissimilar but the tracing packets do cross the physical links.

Algorithm 3.1 describes the computation of our utility metric for a given flow  $(s, d)$ . In the algorithm, given a virtual path  $T'_{s \rightarrow d} = s \rightarrow n_1 \rightarrow \dots \rightarrow n_k \rightarrow d$ , we write  $T'_{s \rightarrow d}[0 : n_i]$  to denote the prefix path  $s \rightarrow n_1 \rightarrow \dots \rightarrow n_i$ . NetHide computes the overall utility by taking the average utility computed over all flows:

$$U_{avg} = avg_{(s,d) \in F} util((s, d), P, V)$$

As with accuracy, a network operator is free to implement a custom utility metric.

In most cases, the accuracy and utility are strongly linked together (we show this in Section 3.5). However, as illustrated in Figure 3.5, there exist cases where the accuracy is high and the utility low or vice-versa.

### 3.3.4 Scalability

To obfuscate topologies with maximal accuracy and utility, a naive approach would consider all possible changes to  $P$ , which is infeasible even for small topologies.

NetHide significantly reduces the number of candidate solutions in order to ensure reasonable runtime while providing close-to-optimal accuracy and utility. The key insight is that NetHide pre-computes a set of *forwarding trees* for each node and later computes  $V$  as the optimal combination of them. Thanks to the reduction from modeling individual links or paths to forwarding trees, NetHide only considers *valid* combinations of paths (i.e., paths that form a tree rooted at  $n$ ,  $\forall n \in N'$ ).

For computing the forwarding trees, NetHide builds a complete graph  $G$  with all nodes from  $V$ , that is  $G = (V, E)$  where  $V = N'$  and  $E = N' \times N'$ , and assigns each edge the same weight  $w(e) = 1 \forall e \in E$ . Then, NetHide uses Dijkstra's algorithm [161] to compute forwarding trees towards each node  $n \in N'$ . That is, a set of paths where the paths form a tree which is rooted at  $n$ . This is repeated until the specified number of forwarding trees per node is obtained while the weights are randomly chosen  $w(e) \sim \text{Uniform}(1, 10)$  for each iteration.

As NetHide pre-computes a fixed number of forwarding trees per node, the ILP solver later only needs to find an optimal combination of  $\mathcal{O}(|N'|)$  forwarding trees instead of  $\mathcal{O}(|N'|^2)$  links and  $\mathcal{O}(|N'|^{|N'|})$  forwarding trees.

We point out that the reduction from individual links or paths to forwarding trees and the small number of considered forwarding trees does not affect the security of  $V$  as security is a hard constraint and thus, NetHide *never* produces a topology that is insecure. In fact, the small number of considered forwarding trees actually makes NetHide more secure because it makes it harder to determine  $P$  even for a powerful brute-force attacker that can run NetHide with every possible input.

### 3.3.5 Security

We now discuss the security provided by NetHide. We consider two distinct attacker strategies: (i) reconstructing the physical topology  $P$  from the virtual topology  $V$ ; and (ii) choosing an attack based on the observed virtual topology  $V$  (without explicitly reconstructing  $P$ ). We describe the two strategies below.

**Reconstructing the physical topology** If the attacker can reconstruct  $P$ , then she can check if  $P$  is insecure and select a link and a set of flows that congests that link. Reconstructing the physical topology is mitigated in two ways. First, the attacker cannot reconstruct  $P$  with certainty by simply observing the virtual topology  $V$ . NetHide's obfuscation function maps any physical topology that is secure to itself (i.e., to  $P$ ). The obfuscation function is therefore not injective, which entails that NetHide guarantees opacity [162], a well-known security property stipulating that the attacker does not know the secret  $P$ .

Given that the attacker cannot reconstruct  $P$  with certainty, she may attempt to make an educated guess based on the observed  $V$  and her

knowledge about NetHide’s obfuscation function. Concretely, the attacker may perform exact Bayesian inference to discover the most likely topology  $T$  that was given as input to the obfuscation function. Exact inference is, however, highly non-trivial as NetHide’s obfuscation function relies on a complex set of constraints. As an alternative, the attacker may attempt to approximately discover a topology  $T$  that was likely provided as input to NetHide. Estimating the likelihood that a topology  $T$  could produce  $V$  is, however, expensive because NetHide’s obfuscation is highly randomized. That is, the estimation step would require a large number of samples, obtained by running  $T$  using the obfuscation function.

**Choosing an attack** In principle, even if the attacker cannot reconstruct  $P$ , she may still attempt to attack the network by selecting a set of flows and checking if these cause congestion or not. As a base case for this strategy, the attacker may randomly pick a set of flows. A more advanced attacker would leverage her knowledge about the observed topology to select the set of flows such that the likelihood of a successful attack is maximized.

In our evaluation, we consider three concrete strategies: (i) *random*, where the attacker selects the set of flows uniformly at random, (ii) *bottleneck+random*, where the attacker selects a link with the highest flow density and selects additional flows uniformly at random from the remaining set of flows, and (iii) *bottleneck+closeness*, where the attacker selects a link with the highest flow density and selects additional flows based on their distance to the link. Our results show that NetHide can mitigate these attacks even for powerful attackers (which can run many flows) and weak physical topologies (with small link capacities) while still providing high accuracy and utility (cf. Section 3.5.7). For example, NetHide provides 90% accuracy and 72% utility while limiting the probability of success to 1% for an attacker which can run twice the required number of flows and follows the *bottleneck+random* strategy in a physical topology where 20% of the links are insecure.

### 3.4 TOPOLOGY DEPLOYMENT

In this section, we describe how NetHide deploys the virtual topology  $V$  on top of the physical topology  $P$ . For this, we first state the challenges NetHide needs to address. Then, we provide insights on the architecture using which we implemented NetHide and describe the packet processing software as well as the controller in detail. In addition, we explain the

design choices that make NetHide partially deployable and we discuss the impact of changes in the physical topology to the virtual topology.

### 3.4.1 Challenges

In the following, we explain the major challenges which need to be addressed by the design and the implementation of the NetHide topology deployment to provide high security, accuracy, utility and performance.

**Reflecting physical events in the virtual topology** Maintaining the usefulness of network tracing and debugging tools is a major requirement for any network obfuscation scheme to be practical. As we explained in the previous sections, NetHide ensures that tracing  $V$  returns meaningful information by maximizing the utility metric. As a consequence, NetHide must assure that the data plane is acting in a way that corresponds to the utility metric.

The key idea to ensure high utility in NetHide is that the tracing packets are sent through the physical network as opposed to being answered at the edge or by a central controller. Answering tracing packets from a single point is impractical as events in  $P$  (such as link failures) would not be visible.

**Timing-based fingerprinting of devices** Besides the IP address of each node in a path, tracing tools allow to determine the round trip time (RTT) between the source and each node in the path. This can potentially be used to identify obfuscated parts of a path.

While packets forwarding is usually done in hardware without noticeable delay, answering to an expired (TTL=0) IP packet involves the router control plane and causes a noticeable delay. Actually, our experiments show that the time it takes for a router to answer to an expired packet not only varies greatly, but is also *characteristic* for the device, making it possible to identify a device based on the distribution of its processing time.

NetHide makes RTT measurements realistic by ensuring that a packet that is supposedly answered by node  $n$  is effectively answered by  $n$ . As such,  $n$  will process the packet as any other packet with an expired TTL irrespective of whether or not obfuscation is in place and the measured RTT is the RTT between the source host and  $n$ .

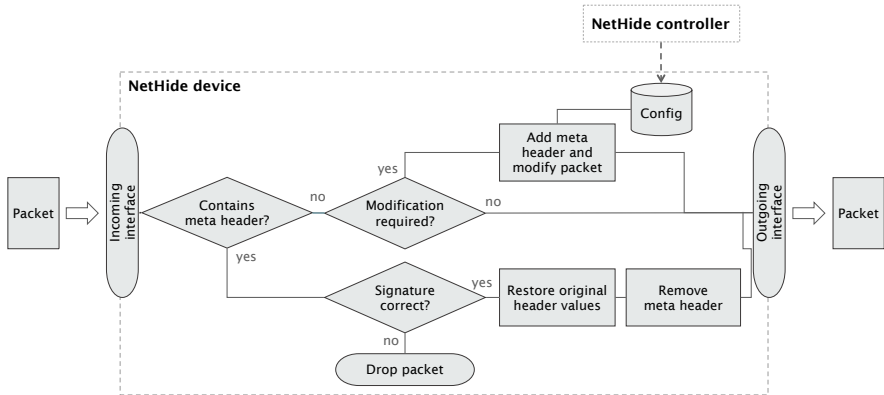


FIGURE 3.6: NetHide topology deployment architecture overview. A controller generates the configuration entries which are later used by the packet processing software running in NetHide devices.

**Packet manipulations at line rate** To avoid tampering with network performance, NetHide needs to parse and modify network packets at line rate. In particular, it needs to manipulate the TTL field in IP headers as well as the IP source and destination addresses. Since changing these fields leads to a changed checksum in the IP header, NetHide also needs to recompute checksums.

While there are many architectures and devices where the NetHide runtime can operate, we decided to implement it in P<sub>4</sub>, which we introduce in Section 2.2.

### 3.4.2 Architecture

NetHide features a controller to translate  $V$  to configurations for programmable network devices, and a packet processing software that is running on network devices and modifies packets according to these configurations.

The device configuration is described as a set of match & action table entries that are queried upon arrival of a packet (Figure 3.6). The entries are installed when  $V$  is deployed the first time and when it changes. At other times, NetHide devices act autonomously.

We describe the packet processing software as well as the controller in the following two sections.

### 3.4.3 *Packet processing software*

The packet processing software is running in the data plane of a network device and typically performs tasks such as routing table lookups and forwarding packets to an outgoing interface. For NetHide, we extend it with functionality to modify packets such that the behavior for a network user is consistent with  $V$ . In the following paragraphs, we explain the processing shown in Figure 3.6.

***Identifying potential tracing packets*** Upon receiving a new packet, a NetHide device first checks whether it is a response to a packet that was modified by NetHide (cf. below). If not, it checks whether the packet's virtual path is different from the physical path and it thus needs to be modified. Even though we often use traceroute packets as examples, NetHide does not need to distinguish between traceroute (or other tracing traffic) and productive network traffic. Instead, it purely relies on the TTL value, the source and destination of a packet and – if needed – it obfuscates traffic of all applications.

***Encoding the virtual topology*** If a packet needs to be modified, NetHide queries the match & action table which returns the required changes for the packet. Changes can include modifications of the destination address and/or the TTL value. If the packet's TTL is high enough that it can cross the egress router, NetHide does not need to modify addresses. However, if the virtual path for this packet has a different length than the physical path, the TTL needs to be incremented or decremented by the difference of the virtual and the physical path length.

If the packet has a low TTL value which will expire before the packet reaches its destination, NetHide needs to ensure that the packet expires at the correct node with respect to  $V$ . For this, NetHide modifies the destination address of the packet such that it is sent to the node that has to answer according to  $V$ . In addition, it sets the source address to the address of the NetHide device that handles the packet. Therefore, the modified packet is sent to the responding router and the answer comes back to the NetHide device. At this point, NetHide needs to restore the original source and destination addresses of the packet and forward the reply to the sender.

***Rewriting tracing packets at line rate*** The devices that we use to deploy NetHide are able to modify network traffic at line rate without impacting throughput. As described above, NetHide sometimes needs to modify the



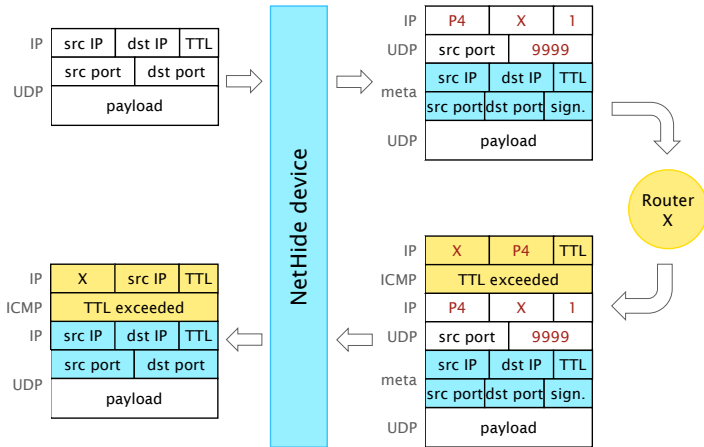


FIGURE 3.7: NetHide devices encode state information into packets in order to avoid maintaining state in the devices.

TTL value in production traffic and it needs to send tracing packets to different routers (which has an impact on the observed RTT; but only for tracing packets whose TTL expires before reaching the destination).

**Rewriting tracing packets statelessly** A naive way to be able to reconstruct the original source and destination addresses of a packet is to cache them in the device (which bears similarities with the operating mode of a NAT device – but the state would need to be maintained on a *per-packet* basis). Since this would quickly exceed the limited memory that is typically available in programmable network devices, NetHide follows a better strategy: instead of maintaining the state information in the device, it encodes it into the packets. More precisely, NetHide adds an additional header to the packet which contains the original (layer 2 and 3) source and destination addresses, the original TTL value as well as a signature (a hash value containing the additional header combined with a device-specific secret value) (cf. Figure 3.7). This *meta header* is placed on top of the layer 3 payload and is thus contained in ICMP time exceeded replies.

**Preventing packet injections** Coming back to the first check when a packet arrives: if it contains a *meta header* and the signature is valid (i.e., corresponds to the device), NetHide restores the original source and destination addresses of the packet and removes the meta header before sending it to the outgoing interface.

### 3.4.4 *NetHide controller*

Below, we explain the key concepts of the NetHide controller which generates the configurations mentioned above.

**Configuring the topology** Being based on P4 devices, configuration entries are represented as entries in match & action tables which are queried by the packet processing program. NetHide's configuration entries are of the following form:

$$(\text{destination, TTL}) \mapsto (\text{virtual destination IP, hops to virtual destination})$$

where the virtual destination IP can be unspecified if only the length of a path needs to be modified. P4 tables can match on IP addresses with *prefixes*, meaning that only one entry per prefix (e.g., 1.2.3.0/24) is required. For example, the entry "(1.2.3.0/24, 1)  $\mapsto$  (11.22.33.44, 5)" means that if the device sees a packet to 1.2.3.4 (or any other IP address in 1.2.3.0/24) with TTL=1, it will send it to 11.22.33.44 and change the TTL value to 5.

**Modifying packets distributedly** NetHide selects one programmable network device per flow which then handles all of the flow's packets. This device must be located before the first spoofed node, i.e., the first node in the virtual path that is different from the physical path.

While there is always one distinct device in charge of handling a certain flow, the same device is assigned to many different flows. To balance the load across devices, NetHide chooses one of the eligible devices at random (this does not impact the obfuscation). For more redundancy, multiple devices could be assigned to each flow.

**Changing the topology on-the-fly** Thanks to the separation between the packet processing software and the configuration table entries,  $V$  can be changed *on-the-fly* without interrupting the network.

### 3.4.5 *Partial deployment*

As deploying a system that needs to run on *all* devices is difficult, we design NetHide such that it can fully protect a network while being deployed on only a few devices. The key enabler for this is that NetHide only needs to modify packets at most at one point for each flow.

NetHide can obfuscate all traffic as soon as it has crossed at least one NetHide device. In the best case, in which NetHide is deployed at the network edge, it can protect the entire network. In the evaluation (Section 3.5), we show that even for the average case in which the NetHide devices are placed at random positions, a few devices are enough to protect a large share of the flows.

#### 3.4.6 *Dealing with topology changes*

NetHide sends tracing packets through  $P$  such that they expire at the correct node according to  $V$ . Changes in  $P$  can impact NetHide in two ways:

1. When links are *added* to  $P$  or the routing behavior changes: some flows may no longer traverse the device that was selected to obfuscate them. This can be addressed by installing configuration entries in multiple devices (which results in a trade-off between resource requirements and redundancy). Since  $V$  is secure in any case, there is no immediate need to react to changes in  $P$ . However, to provide maximum accuracy and utility, NetHide can compute a new  $V'$  based on  $P'$  and deploy it without interrupting the network.
2. When links are *removed* from  $P$ : this results in link failures in  $V$  and has no impact on the security of  $V$ . If the links are permanently removed, NetHide can compute and deploy a new virtual topology.

### 3.5 EVALUATION

In this section, we show that NetHide: *(i)* obfuscates topologies while maintaining high accuracy and utility (Sections 3.5.2 and 3.5.3); *(ii)* computes obfuscated topologies in less than one hour, even when considering large networks (Section 3.5.4). Recall that this computation is done offline, once, and does not impact network performance at runtime; *(iii)* is resilient against timing attacks (Section 3.5.5); *(iv)* is effective even when partially deployed (Section 3.5.6); *(v)* mitigates realistic attacks (Section 3.5.7); and *(vi)* has little impact on debugging tools (Section 3.5.8).




	Abilene	Switch	US Carrier
			
Nodes	11	42	158
Links	14	63	189
Max. flow density	35	390	11301
Avg. flow density	19	89	1587

TABLE 3.1: We evaluate Nethide based on three realistic topologies of different size.

### 3.5.1 Metrics and methodology

**Metrics** To be able to compare the results of our evaluation with different topologies, we use the average *flow density reduction factor*, which denotes the ratio between the flow density in the physical topology  $P = (N, L, T)$  and in the virtual topology  $V = (N', L', T')$ :

$$FR = 1 - \frac{\text{avg}_{l \in L'} fd(V, l)}{\text{avg}_{l \in L} fd(P, l)}$$

The flow density denotes the number of flows that are carried at each link (cf. Section 3.1.3). For example,  $FR = 0.2$  means that the links in  $V$  carry 80% less flows than those in  $P$  (on average). For the accuracy and utility of  $V$ , we use  $A_{\text{avg}}$  and  $U_{\text{avg}}$  as defined in Section 3.3.

**Datasets** We consider three publicly available topologies from [163]: a small (Abilene, the former US research network), a medium (Switch, the network connecting Swiss universities) and a large one (US Carrier, a commercial network in the US). Table 3.1 lists key metrics for the three topologies. For the forwarding behavior, we assume that traffic in  $P$  is routed along the shortest path or a randomly picked shortest path in case there are multiple shortest paths between two nodes.

**Parameters** We run all our experiments with the following parameters: All nodes in  $P$  can act as ingress and egress for malicious traffic (which is the worst case when an attacker is everywhere). We also assume that all links have the same capacity. Since tracing packets need to be answered by the

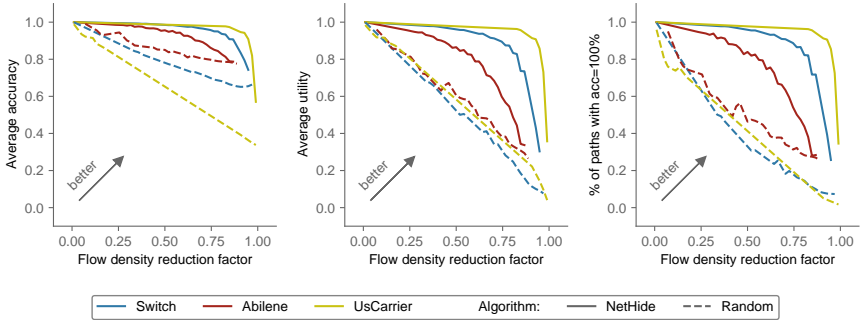


FIGURE 3.8: Accuracy and utility for different protection margins. NetHide achieves high accuracy (left plot) and utility (middle) and does not change most of the paths at all (right plot) while reducing the flow density by more than 75%.

correct node, NetHide only adds virtual links but no nodes (i.e.,  $N = N'$ ). We consider 100 forwarding trees per node. For the ILP solver, we specify a maximum relative gap of 2%, which means that the optimal results can be at most 2% better than the reported results (in terms of accuracy and utility, security is not affected). We run NetHide at least 5 times with each configuration and plot the average results.

### 3.5.2 Protection vs. accuracy and utility

In this experiment, we analyze the impact of the obfuscation on the accuracy and utility of  $V$ . For this, we run NetHide for link capacities  $c$  (the maximum flow density) varying between 10% and 100% of the maximum flow density listed in Table 3.1.

Figure 3.8 depicts the accuracy (left) and utility (center) achieved by NetHide according to the flow density reduction factor. An ideal result is represented by a point in the upper right corner translating to a topology that is both highly obfuscated and provides high accuracy and utility. As baseline, we include the results of a naive obfuscation algorithm that computes  $V$  by adding links at random positions and routing traffic along a shortest path.

NetHide scores close to the optimal point especially for large topologies. We observe that the random algorithm can achieve high accuracy and utility

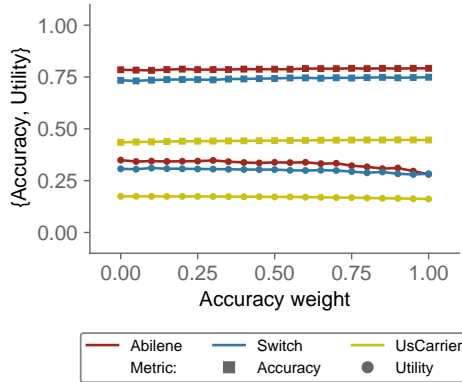


FIGURE 3.9: The accuracy weight has a small impact for our accuracy and utility metrics.

(when adding few links) or high protection (when adding many links) but not both at the same time. Though, in a small area (very high flow density reduction in a small topology), the random algorithm can outperform NetHide. The reason is that such a low flow density is only achievable in an (almost) complete graph. While adding enough links randomly will eventually result in a complete graph, the small number of forwarding trees considered by NetHide does not always contain enough links to build a complete graph.

In Figure 3.8 (right), we show the percentage of flows that do not need to be modified (i.e., have 100% accuracy and utility) depending on the flow density reduction factor.

Figure 3.8 (right) illustrates that NetHide can obfuscate a network without modifying most of its paths therefore preserving the usability of tracing tools. In the medium size topology, NetHide computes a virtual topology that lowers the average flow density by more than 80% while keeping more than 80% of the paths *identical*. This is significantly better than the random baseline where a flow density reduction by 80% only preserves about 15% of the paths. We observe that larger topologies generally exhibit better results than small ones. This is due to the fact that in bigger topologies, a small modification has less impact on average accuracy than in a small topology while still providing high obfuscation. Conversely, smaller

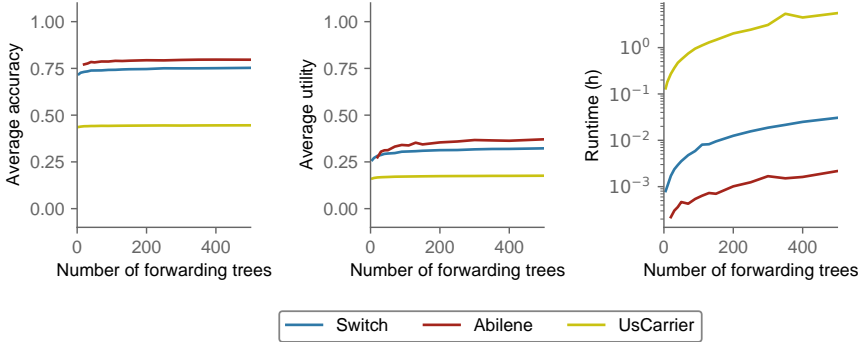


FIGURE 3.10: Accuracy, utility and runtime for different number of forwarding trees. Considering only a small number of forwarding trees per node does not significantly decrease the accuracy and utility of NetHide but drastically decreases the runtime. Thanks to this, NetHide can obfuscate large topologies ( $>150$  nodes) in less than one hour.

topologies lead to worse results as a small number of changes can have a big impact.

### 3.5.3 Accuracy vs. utility

In Figure 3.9, we analyze the impact of the accuracy weight ( $w_{\text{acc}}$  in Figure 3.4) on the resulting accuracy and utility. We specify the capacity of each link to 10% of the maximum flow density listed in Table 3.1 and observe that  $w_{\text{acc}}$  has a relatively small impact for our accuracy and utility metrics especially for large topologies. This confirms that a topology with a high accuracy typically also has a high utility. If the paths are similar (high accuracy), the packets are routed via the same links (high utility), too.

### 3.5.4 Search space reduction and runtime

In this experiment, we analyze the impact of the search space reduction – in terms of the number of forwarding trees per node – on the runtime of NetHide. As we explained in Section 3.3.4, NetHide considers only a small subset of forwarding trees to improve scalability. We again specify the capacity of each link to 10% of the maximum flow density listed in

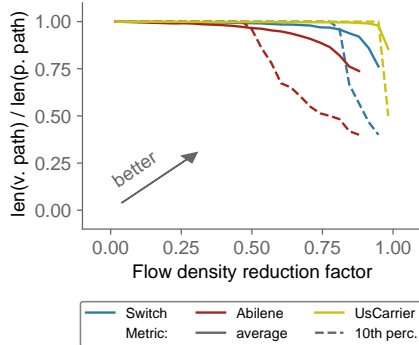


FIGURE 3.11: Length ratio between the virtual and the physical paths. Reducing the flow density by 80% changes path lengths by less than 20%.

Table 3.1 and run NetHide for a varying number of forwarding trees per node. The experiments were run in a VirtualBox VM running Ubuntu 16.04 with 20 Intel Xeon E5 CPU cores and 90 GB of memory.

In Figure 3.10, we show that a small number of forwarding trees is enough to reach close-to-optimal results. While the runtime increases exponentially with the number of forwarding trees, the accuracy and utility do not noticeably improve above 100 forwarding trees per node.

The runtime of NetHide when considering 100 forwarding trees per node is within one hour, even for large topologies (Figure 3.10). As the topology is computed offline (cf. Section 3.4.6), such a running time is reasonable.

### 3.5.5 Path length

In this experiment, we analyze the difference between the lengths of paths in  $P$  and  $V$ . Large differences between the length of the physical path and the virtual path can lead to unrealistic RTTs and leak information about the obfuscation (e.g., if the RTT is significantly different for two paths of the same length).

As the results in Figure 3.11 show, virtual paths are shorter than physical paths (the ratio is  $\leq 1$ ) – intuitively because removing a node from a path has a smaller impact on our accuracy and utility metrics than adding one. And – for the medium and large topology – the virtual paths are less than



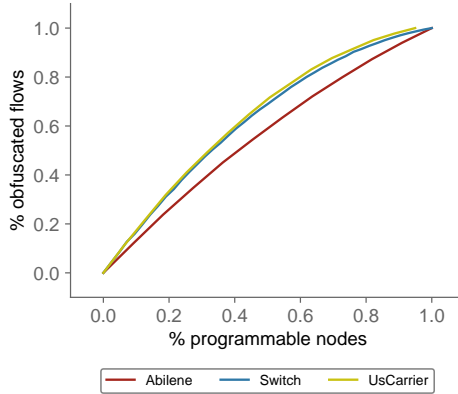


FIGURE 3.12: Partial deployment at random locations. 40 % NetHide devices allow to protect up to 60 % of the flows *that need obfuscation*

10 % shorter both on average and in the 10<sup>th</sup> percentile for a flow density reduction of 80 %.

The resulting small differences in path lengths support our assumption that timing information mainly leaks through the processing time at the last node and not through the propagation time (Section 3.4) as long as all links have roughly the same propagation delay.

### 3.5.6 Partial deployment

We now analyze the achievable protection if not all devices at the network edge are programmable. In NetHide, a flow can be obfuscated as long as it crosses a NetHide device before the first spoofed node (the first node that is different from the physical path). This is obviously the case if all edge routers are equipped with NetHide. Yet, as we show in Figure 3.12, a small percentage of NetHide devices (e.g., 40 %) is enough to protect the majority (60 %) of flows even in the average case where the devices are placed at random locations and all nodes are considered as ingress and egress points of traffic (i.e., as edge nodes).

To obtain the results in Figure 3.12, we set the maximum flow density to 10 % of the maximum value in Table 3.1 and vary the percentage of programmable nodes in  $V$  between 0 and 100 %. For each step, we compute

the average amount of flows that can be protected for 100 different samples of programmable devices.

The percentage of obfuscated flows in Figure 3.12 is normalized to only consider flows that need to be obfuscated. As we have shown in Figure 3.8, the vast majority of flows does not need to be obfuscated at all.

As an alternative approach to partial deployment, NetHide can be extended to incorporate the number and/or locations of NetHide devices as a constraint or as an objective such as to compute virtual topologies that can be deployed without new devices or with as few programmable devices as possible.

### 3.5.7 Security

As we explained in Section 3.3.5, inferring the exact physical input topology from the virtual topology is difficult.

However, an attacker can try to attack  $V$  directly, without trying to determine  $P$ . Such an attacker is limited by the fact that she does not know  $P$  and by a maximum number (budget) of flows that she can create. Therefore, the key challenge for the attacker is to select the flows such that they result in a successful attack on  $P$ .

Besides the attacker's budget, her chances of success also depend on the robustness of  $P$ : If  $P$  is weak (i.e., the capacity of many links is exceeded), it either needs to be obfuscated more or attacks are more likely to succeed.

In this experiment, we simulate three feasible strategies for an attacker to select  $b$  flows:

- *Random*: Samples  $b$  flows uniformly at random from the set of all flows  $F$ .
- *Bottleneck+Random*: Identifies the link with the highest flow density in  $V$  (a "bottleneck" link  $l_b$ ) and attacks by initiating all the  $fd(l_b)$  flows that cross this link plus  $(b - fd(l_b))$  random additional flows.
- *Bottleneck+Closeness*: Identifies the link  $l_b$  with the highest flow density in  $V$  and attacks by initiating all the  $fd(l_b)$  flows that cross this link plus  $(b - fd(l_b))$  nearby flows (according to the metric in Algorithm 3.2).

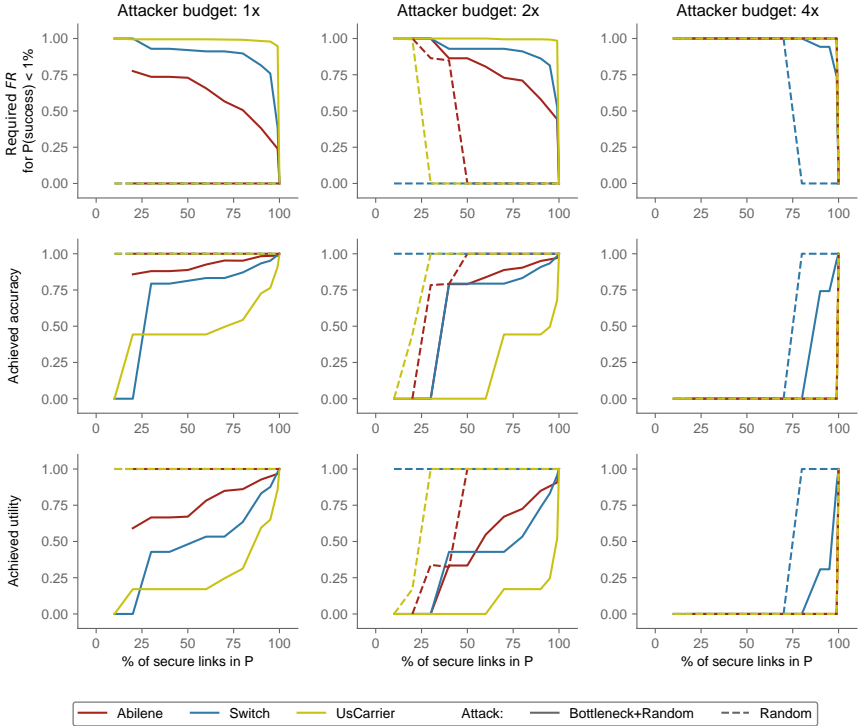


FIGURE 3.13: Attack simulations comparing the *Random* attacker with *Bottleneck+Random*. The plots show the required flow density reduction (FR) for making the attacker succeed with  $Pr < 1\%$  (first row) and the obtained accuracy and utility (second and third row) depending on the link capacity of the physical topology (measured as the percentage of secure links in the x-axis). For example, defending the Switch topology with only 60% secure links against *Bottleneck+Random* with  $2\times$  budget maintains 80% accuracy.

An attack is successful if running the selected set of flows in  $P$  exceeds any link's capacity (not necessarily the link that the attacker tried to attack).

In our simulations, we vary both the attacker's budget and the robustness of  $P$  (in terms of the link capacity). We vary the capacity such that between 10% and 100% of the links in  $P$  are secure (e.g., if 10% of the links are secure, an attacker could directly attack 90% of the links if there was no obfuscation). For each choice of the link capacity  $c$  in  $P$ , we vary the number

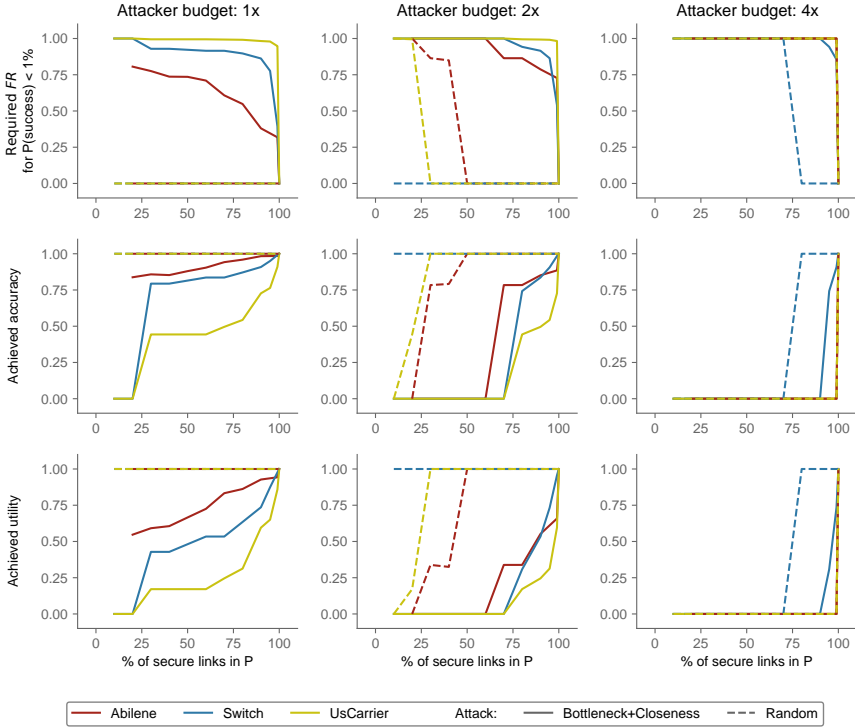


FIGURE 3.14: Attack simulations comparing the *Random* attacker with *Bottleneck+Closeness*. *Bottleneck+Closeness* is slightly more powerful than *Bottleneck+Random* (Figure 3.13), which results in more obfuscation that is required.

of flows that the attacker can initiate between  $b = c + 1$  (just enough to break a link) and  $b = 4 \times (c + 1)$  (four times the number of flows that the most efficient attacker would need).

To obtain the simulation results in Figure 3.13 and Figure 3.14, we simulated 10k attempts (*Random* and *Bottleneck+Random*) and 1k attempts (*Bottleneck+Closeness*) for each virtual topology from Section 3.5.2 and each combination of the link capacity and attacker budget.

In Figure 3.13 we compare the *Random* attacker with *Bottleneck+Random* and in Figure 3.14 we compare *Random* with *Bottleneck+Closeness*. In the first row of each figure, we plot how much obfuscation (i.e., in terms of the flow density reduction factor) is required to make the attacker successful

in  $< 1\%$  of her attempts. There, we observe that the *Random* attacker is (as expected) the least powerful because it requires less obfuscation to defend against it and that *Bottleneck+Closeness* is slightly more powerful than *Bottleneck+Random*. Considering the setting with the Abilene topology and the attacker with  $2\times$  budget: Mitigating this attacker requires no obfuscation when she follows the *Random* strategy, but 71% (*Bottleneck+Random*) or 86% (*Bottleneck+Closeness*) flow density reduction for the more sophisticated strategies.

The required flow density reduction naturally increases as the attacker's budget increases. In the right column where the attacker can run four times the number of required flows, even the *Random* attacker is successful because she can run so many flows (or even all possible flows in many cases) that it does not matter how the flows are selected.

The second and third row in the plots show the accuracy and utility that is preserved after obfuscating the topology. We observe there, that especially the Abilene and Switch topologies provide high accuracy and utility even if less than 50% of the links in  $P$  are secure. Comparing Figures 3.13 and 3.14 shows that since mitigating *Bottleneck+Closeness* requires more obfuscation, the achieved accuracy and utility are lower.

### 3.5.8 Case study: Link failure detection

We now show that `NetHide` preserves most of the usefulness of tracing tools by considering the problem of identifying link failures in obfuscated topologies. For our analysis, we use all three topologies and a flow density reduction factor of 50%. Then, we simulate the impact of an individual failure for each link. That is, we analyze how a failing *physical* link is represented in  $V$ .

Failing a link can have different effects in  $V$ : Ideally, it is *correctly observed*, which means that the exact same link failure appears in  $V$ . But since  $V$  contains links that are not in  $P$  or vice-versa, a physical link failure can be observed as *multiple link failures* or as the *failing of another virtual link*.

In Figure 3.15, we show that the vast majority of physical link failures is precisely reflected in the virtual topology. That is, `NetHide` allows users to use prevalent debugging tools to debug connectivity problems in the network. These results are a major advantage compared to competing

---

Algorithm 3.2: Flow preference metric. Flows that contain the bottleneck link or at least one of the endpoints of the link are more promising to be useful in the attack.

---

**Input:** Virtual topology  $V = (N', L', T')$ ,  
 Flow  $(s, d) \in N' \times N'$ ,  
 Flow path  $T'_{s \rightarrow d}$   
 Bottleneck link  $(n_1, n_2) \in L'$

**Output:** Preference  $p \in [0, 1]$

```

1 if  $(n_1 \in T'_{s \rightarrow d}) \wedge (n_2 \in T'_{s \rightarrow d})$  then
2    $p \leftarrow 1 / |\text{links between } n_1 \text{ and } n_2 \text{ in } T'_{s \rightarrow d}|$ 
3 else if  $(n_1 \in T'_{s \rightarrow d}) \wedge (n_2 \notin T'_{s \rightarrow d})$  then
4    $n_a \leftarrow \text{node after } n_1 \text{ in } T'_{s \rightarrow d}$ 
5    $n_b \leftarrow \text{node before } n_1 \text{ in } T'_{s \rightarrow d}$ 
6    $p_a \leftarrow \text{length of path from } n_2 \text{ to } n_a$ 
7    $p_b \leftarrow \text{length of path from } n_2 \text{ to } n_b$ 
8    $p \leftarrow 1 / \min(p_a, p_b)$ 
9 else if  $(n_1 \notin T'_{s \rightarrow d}) \wedge (n_2 \in T'_{s \rightarrow d})$  then
10   $p \leftarrow$  (see above with  $n_1$  and  $n_2$  flipped)
11 else
12   $p \leftarrow 0$ 

```

---

approaches [115, 117] that do not send the tracing packets through the actual network.

### 3.6 FREQUENTLY ASKED QUESTIONS

Below, we provide answers to some frequently asked questions and potential extensions of NetHide.

*Can a topology be de-obfuscated by analyzing timing information?* In NetHide, each probing packet is answered by the correct router and thus the processing time at the last node is realistic. Though, the propagation time can leak information in topologies where the propagation delay of some links is significantly higher than of others.

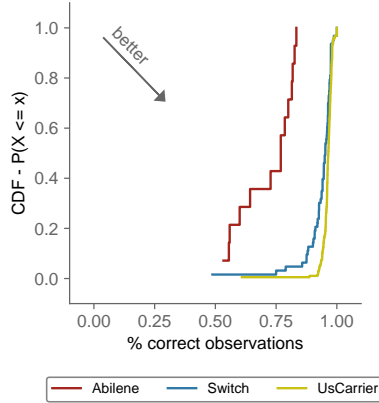


FIGURE 3.15: Link failures are correctly observed with high probability (e.g., for Switch: only 15% of the failures appear in less than 90% of the paths.)

However, extracting information from the propagation time in geographically small networks is hard for three reasons: (i) it is impossible to measure propagation time separately. Instead, only the RTT is measurable; (ii) the RTT includes the unknown return path; and (iii) NetHide keeps path length differences are small. For topologies exhibiting larger delays, NetHide can be extended to consider link delays as additional constraints.

The same arguments hold for analyzing queuing times or other time measurements. Moreover, delays often vary greatly in short time intervals, making it practically infeasible to perform enough simultaneous measurements.

*Can a topology be de-obfuscated by analyzing link failures?* Because some physical link failures are observed as multiple concurrent link failures in the virtual topology, an attacker can try to reconstruct the physical topology by observing link failures over a long timespan. However, this strategy is not promising for the following reasons: (i) most of the link failures are directly represented in the virtual topology (cf. Section 3.5.8). Observing them does not provide usable information for de-obfuscation; and (ii) analyzing link failures over time requires permanent tracing of the entire network between, which would make the attacker visible and is against the idea of LFAs.

**Is NetHide compatible with link access control or VLANs?** Not at the moment, but we can easily extend our model to support them. The required changes are: (i) link access control policies need to be part of the NetHide's input; (ii) the ILP needs additional constraints to respect different VLANs (i.e., model forwarding trees per VLAN); (iii) the output consists of VLAN-specific paths; and (iv), the runtime additionally matches on the VLAN ID and applies the appropriate actions.

**Does NetHide support load-balancing?** Not at the moment, but after the following extensions: (i) instead of an exact path for each flow, we specify the *expected* load that a flow adds to each link (e.g., using max-min fair allocation as in [164]); (ii) the constraints regarding the flow density now constrain the *expected* flow density; (iii) the virtual topology can contain multiple parallel paths and probabilities with which each path is taken; and (iv) the runtime randomly selects one of the possible paths.

**How close to the optimal is the solution computed by NetHide?** Computing this distance is computationally infeasible as it requires to exhaustively enumerate all possible solutions (one of the cruxes behind NetHide security). Instead, we measure the distance between the virtual and the physical topology (Section 3.5.2) and show that the virtual topology is already very close (in terms of accuracy and utility) to the physical one. The optimal solution would therefore only do slightly better, while being much harder to compute.

**Can NetHide be used with other metrics for computing the flow density?** At present, NetHide requires a static metric such that the flow density can be computed before obfuscating the topology. For simplicity, we assume that the load which each flow imposes to the network is the same and all links have the same capacity. However, this assumption can easily be relaxed to allow specific loads and capacities for each flow and link (therefore requiring more knowledge or assumptions about the topology and the expected traffic).

### 3.7 RELATED WORK

Existing works on detecting and preventing LFAs can be broadly classified into reactive and proactive approaches. Reactive approaches only become active once a potential LFA is detected. As such, they do not prevent LFAs



and only aim to limit their impact after the fact. CoDef [145] works on top of routing protocols and requires routers to collaborate to re-route traffic upon congestion. SPIFFY [151] temporarily increases the bandwidth for certain flows at a congested link. Assuming that benign hosts react differently than malicious ones, SPIFFY can tell them apart. Liaskos et al. describe a system [152] that continuously re-routes traffic such that it becomes unlikely that a benign host is persistently communicating via a congested link. Malicious hosts on the other hand are expected to adapt their behavior. Nyx [165] addresses the problem of LFAs in the context of multiple autonomous systems (ASes). It allows an AS to route traffic from and to another AS along a path that is not affected by an LFA. Routing traffic around a congested AS is achieved through BGP messages and does not require any cooperation from the other ASes. Later work [166]<sup>1</sup> has raised doubts about the feasibility of this approach because the required BGP messages would not be accepted in many actual ASes. Ripple [74]<sup>1</sup> provides a flexible framework to detect LFAs. Thanks to its policy language, Ripple can emulate several earlier reactive defense techniques. From these policies, Ripple compiles P4 programs and it leverages programmable switches to quickly detect LFAs.

On the other hand, proactive solutions – including NetHide – aim at preventing LFAs from happening and are typically based on obfuscation.

HoneyNet [115] uses software-defined networks to create a virtual network topology to which it redirects traceroute packets. While this hides the topology from an attacker, it also makes traceroute unusable for benign purposes.

Trassare et al. implemented topology obfuscation as a kernel module running on border routers [117]. The key idea is to identify the most critical node in the network and to find the ideal position to add an additional link that minimizes the centrality of this node. The border router replies to traceroute packets as if there was a link at the determined position. However, adding a single link has little impact on the security of a big network and even if the procedure would be repeated, an attacker could determine the virtual links with high probability. Further, traceroute becomes unusable for benign users as the replies come from the border router.

Linkbait [116] identifies potential target links of LFAs and tries to hide them from attackers. Hiding a target link is done by changing the routing of tracing packets from bots in such a way that the target link does not appear

---

<sup>1</sup> This work was published after NetHide

in the paths. As a prerequisite to only redirect traffic from bots, Linkbait describes a machine learning-based detection scheme that runs at a central controller which needs to analyze all traffic. Being based on re-routing of packets, Linkbait can only present paths that exist in the network. Therefore, a topology that does not have enough redundant paths cannot be protected. The paper does not discuss issues with an attacker that is aware of the protection scheme and sends tracing traffic that is likely to be misclassified and therefore not re-routed.

ProTO [167, 168]<sup>1</sup> operates in two steps: First, it uses machine learning to detect packets that show path tracing behavior. Then, it obfuscates the topology for these packets. Different from NetHide, ProTO focuses on topology inference through end-to-end delay measurements. This is why ProTO delays packets in order to obfuscate the topology instead of modifying the contents of path tracing packets.

NetObfu [169]<sup>1</sup> runs in the control plane of OpenFlow switches and follows a similar approach as NetHide: It modifies path tracing responses such that an attacker (or every other network user) infers the virtual topology instead of the physical one. NetObfu uses greedy algorithms to compute a virtual topology, which decreases the computation time compared to NetHide.

BottleNet [170]<sup>1</sup> also runs in the control plane of OpenFlow switches. It extends NetHide and other previous approaches in three dimensions: (i) it supports more metrics to determine bottleneck links (including dynamic metrics such as the actual link load); (ii) it generates more complex virtual topologies (thereby reducing the utility of path tracing tools); and (iii) it reroutes production traffic (to prevent “blind” LFAs where the attacker congests a link by chance).

AntiTomo [171]<sup>1</sup> focuses on topology inference through tomography (e.g., by measuring end-to-end delays, similar to ProTO [167, 168]). To prevent this type of topology inference, AntiTomo generates a virtual topology that is secure and adds little overhead in terms of the end-to-end delay. The authors evaluate AntiTomo only in simulations, but to deploy the virtual topology, production traffic would need to be delayed.

EqualNet [118]<sup>1</sup> identifies four limitations of NetHide, Trassare et al.’s approach [117], and LinkBait [116]: (i) adversaries can infer the popularity of links; (ii) virtual topologies are too similar to the physical ones; (iii) virtual topologies are not secure in the long term; and (iv) path tracing tools

---

<sup>1</sup> This work was published after NetHide

return false information. Regarding `NetHide`, (i) only occurs if the interface of multiple (physical or virtual) links have the same IP address; (ii) is not a problem since the virtual topology is secure (w.r.t. our definition); (iii) is indeed a limitation because `NetHide` would need to recompute a new virtual topology if the physical one changes; and (iv) is minimized since `NetHide` minimizes the amount of obfuscation in order to preserve the usefulness of path tracing tools. As a proposal that does not suffer from these limitations, `EqualNet` generates a virtual topology by adding not only virtual links, but also virtual nodes and by distributing the path tracing flows equally among all links and nodes.

Other approaches that are related to LFAs but not particularly to our work are based on virtual networks [172], require changes in protocols or support from routers and end hosts [173–175] or focus on the detection of LFAs [157].

### 3.8 CONCLUSION

We presented a new, usable approach for obfuscating network topologies. The core idea is to phrase the obfuscation task as a multi-objective optimization problem where security requirements are encoded as hard constraints and usability ones as soft constraints using the notions of accuracy and utility.

As a proof-of-concept, we built a system, called `NetHide`, which relies on an ILP solver and effective heuristics to compute compliant obfuscated topologies and on programmable network devices to capture and modify tracing traffic at line rate. Our evaluation on realistic topologies and simulated attacks shows that `NetHide` can obfuscate large topologies with marginal impact on usability, including in partial deployments.

While we focused on obfuscating network topologies to mitigate link-flooding attacks, an adapted version of `NetHide` could also be used to obfuscate topologies for other purposes (e.g., if an ISP wants to hide other aspects of its topology).

In the next chapter, we will present a system to obfuscate network traffic in order to prevent traffic-analysis attacks.

## OBFUSCATING WAN TRAFFIC TO PREVENT TRAFFIC-ANALYSIS ATTACKS

---

In this chapter, we present *ditto*, a traffic obfuscation system adapted to the requirements of wide area networks: achieving high-throughput traffic obfuscation at line rate without modifications of end hosts.

Many large organizations operate dedicated wide area networks (WANs) as a critical infrastructure distinct from the Internet to connect their data centers and remote sites. For example, cloud service providers such as Google [176], Amazon [177], and Microsoft [178] operate WANs to achieve low-latency, high-throughput inter data center communication. Public safety and security organizations rely on WANs to achieve secure and reliable communication between their sites (e.g., [179–183]). For large organizations, these WANs provide 100s of Gbps to Tbps of capacity over long distances and can cost 100s of millions of dollars per year [184].

WANs are an attractive target for eavesdropping attacks and mass surveillance because they are often used to transport large amounts of sensitive data. And because WANs spread over large geographical areas, it is impossible to secure the cables physically from wiretapping. Past revelations show that intercontinental fiber links were subject to tapping by governmental agencies [185, 186] or other entities [187] and many devices are available to tap on fiber links [188–192]. Indeed, major operators such as Amazon, Microsoft, and OVH acknowledge that WAN traffic is at risk and they use MACsec [193] to encrypt their traffic not only at the application layer, but also at the link layer [194–197].

However, it is well known that encryption alone is not sufficient to protect against traffic-analysis attacks [198, 199]. Even if the network traffic is end-to-end encrypted, metadata such as the traffic volume, the packet sizes and the timing information reveals a lot about ongoing activities. As a result, eavesdroppers intercepting the WAN communication can still perform traffic analysis attacks. Such attacks are mostly known from Internet traffic, where past work shows that it is possible to infer the contents of VoIP calls [119, 127], streamed movies [128, 129]; visited websites [130–134], or the device identities [120–125] without having to break the encryption.

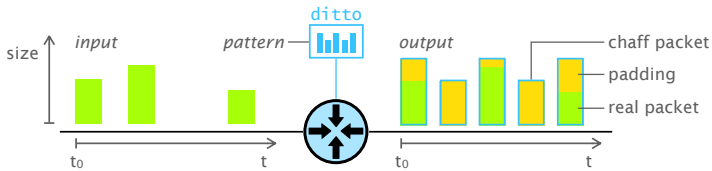


FIGURE 4.1: ditto adds padding and chaff packets such that the outgoing traffic always follows a predefined pattern

Nonetheless, the same attacks can be applied to WAN traffic if the WAN carries the incoming and outgoing Internet traffic (which is typically the case if a company sends all Internet traffic via a central firewall). More generally, it has been shown many times (e.g., in [200–202]) that traffic classification also works for encrypted traffic.

Many techniques have been proposed to protect against traffic-analysis attacks in the Internet. However, these techniques are not well adapted to the specific requirements of WAN traffic protection. Techniques such as BuFLO [121], CS-BuFLO [137], HORNET [104], or TARANET [106] add padding to obfuscate the size of individual packets and flows and require modifications on the software and protocols of the end hosts. For many organizations operating a WAN, it is impossible to adapt these protocols on all end hosts (e.g., because a cloud provider does not control the software that is running on its customer’s instances). Other techniques such as Loopix [203], PriFi [204], or Wang et al. [136] impose strict transmission schedules and rates per flow, and thus severely limit the achievable throughput. As WAN traffic is high-throughput in nature, these solutions are not efficient enough to deal with high link traffic rates up to 100 Gbps.

This chapter presents **ditto**, an in-network and hardware-based traffic obfuscation system specifically tailored to WANs. As illustrated in Figure 4.1, ditto shapes traffic according to a predefined pattern (a periodic sequence of packet sizes at a fixed rate) using three operations: (i) packet padding; (ii) packet delaying; and (iii) chaff packet insertion. When there are “real” packets to transmit, ditto pads and transmits them. When there are no real packets, it transmits dummy “chaff” packets. Therefore, ditto only adds overhead (padding or chaff packets) in a way that does not degrade the network performance for the real traffic. It fills the gaps when there is not enough real traffic to transmit and (slightly) delays real packets in order to make the resulting network behavior independent of the actual traffic being sent.

`ditto` runs on the gateway network devices (e.g., routers or switches) of the WAN sites and does not require any modification to the end hosts or protocols. Being network-based, it is further efficient and supports high-speed obfuscation even when the traffic is bursty and unpredictable. `ditto` devices can react locally to traffic changes in real time. This allows them to quickly adapt to different network loads and to add or remove chaff packets almost instantly depending on the actual link load. In contrast, application-based approaches that run on the end hosts lack the visibility of the network load and need to send chaff traffic independent of other applications, which creates a significant overhead that `ditto` does not have.

We implemented `ditto` using off-the-shelf programmable network hardware of the same type as major operators have already deployed [205, 206]. We show that `ditto` can obfuscate packet size, timing and volume information at line rate. In the evaluation, we run interactive applications over `ditto` and we show that a `ditto`-enabled device can obfuscate up to 70 Gbps of production traffic (on a 100 Gbps link) without any significant impact on the network performance (in terms of throughput, packet loss, latency and jitter). This performance is enough for typical WANs since they usually run at (much) less than 60 % utilization [176, 180, 183]. Even in highly optimized WANs such as the ones of Google and Microsoft where the utilization is close to 100 % [176, 184], `ditto` could protect all the non-background traffic (which accounts to less than 50 % [184]). We further show that the efficient patterns computed by `ditto` result in a significant performance increase compared to simpler approaches in previous work while not compromising security properties against traffic-analysis attacks.

Our main contributions are:

- a strategy to determine packet sizes that allow an efficient mixing of real and chaff packets (Section 4.3);
- an architecture to obfuscate the traffic volume and timing at line rate in network switches (Section 4.4);
- a full implementation (available as open source<sup>1</sup>) on off-the-shelf hardware (Section 4.6); and
- an evaluation on real Internet traffic and with interactive applications (Section 4.7).

The remainder of this chapter is organized as follows. In Section 4.1, we describe the network and attacker models as well as the security goals. In

---

<sup>1</sup> <https://github.com/nsg-ethz/ditto>

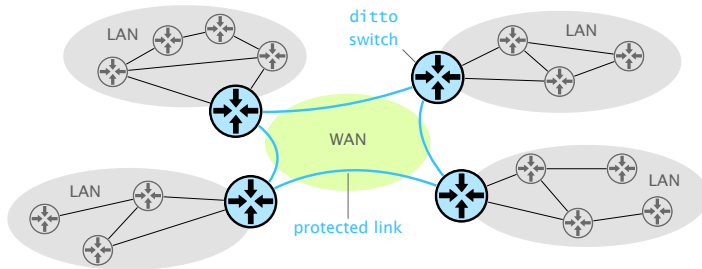


FIGURE 4.2: Network model. ditto protects WAN links which interconnect different sites of an organization.

Section 4.2, we provide an overview over ditto before we describe its main components in more detail (pattern computation in Section 4.3 and traffic shaping in Section 4.4). In Section 4.5, we discuss ditto’s security properties and limitations. In Section 4.6 we describe the hardware implementation and in Section 4.7 we evaluate it. Finally, we review related work in Section 4.8 and conclude in Section 4.9.

#### 4.1 MODEL

In this section, we describe the network model (Section 4.1.1), the attacker model (Section 4.1.2), and ditto’s security goals (Section 4.1.3).

##### 4.1.1 Network model

We consider a wide area network (WAN) which connects multiple sites of one organization over dedicated, encrypted tunnels as illustrated in Figure 4.2. These tunnels can be created at layer 2 (e.g., leased fibers and MACsec encryption) or at layer 3 (e.g., IPsec tunnels). Each tunnel has a guaranteed bandwidth which the organization can fully utilize.

Each site is connected to the WAN with a programmable switch. These switches act as gateways between the local area network (LAN) in each site and the link(s) which interconnect the sites. The operator has full control over these switches and the LANs, but it does not control the WAN tunnels.

We note that such programmable switches are widely-available already and being deployed in large-scale infrastructure including AT&T, Deutsche Telekom, and Alibaba [205, 206].

#### 4.1.2 *Attacker model*

We assume that the attacker has access to all devices and links in the WAN, but she does not have access to devices or links inside the LANs of the operator (including the gateways where ditto is running). The attacker can record timestamps and packet sizes but she cannot access the contents of packets since they are encrypted. We assume that the encryption happens at the same layer as the tunnel (e.g., MACsec [193] for a layer-2 tunnel, or IPsec [207] for a layer-3 tunnel). The attacker can also inject, modify, delay, or drop packets.

Our attacker model is realistic for typical organizations. As we elaborated above, several such wiretapping attacks happened in the past [185, 187] and major operators deploy link-layer encryption to mitigate them [194–197].

#### 4.1.3 *Security goals*

Similar to related work [106, 203], ditto shapes network traffic such that it satisfies the following security goals:

- *Volume anonymity*: The attacker cannot determine the real size of individual packets and flows which are sent over the WAN. This prevents attacks such as the one presented by Boshart and Rossow [208].
- *Timing anonymity*: The attacker cannot determine the timing between packets composing real traffic. This prevents attacks such as the ones presented by Wang et al. [209] and Feghhi and Leith [210].
- *Path anonymity*: The attacker cannot track packets across WAN tunnels. This prevents attacks such as the one presented by Wang et al. [209].

The key enabler for ditto to achieve these goals at line rate is that ditto operates in the network (on routers or switches) and not on end devices such as clients or servers. In the following section, we describe this deployment scenario.



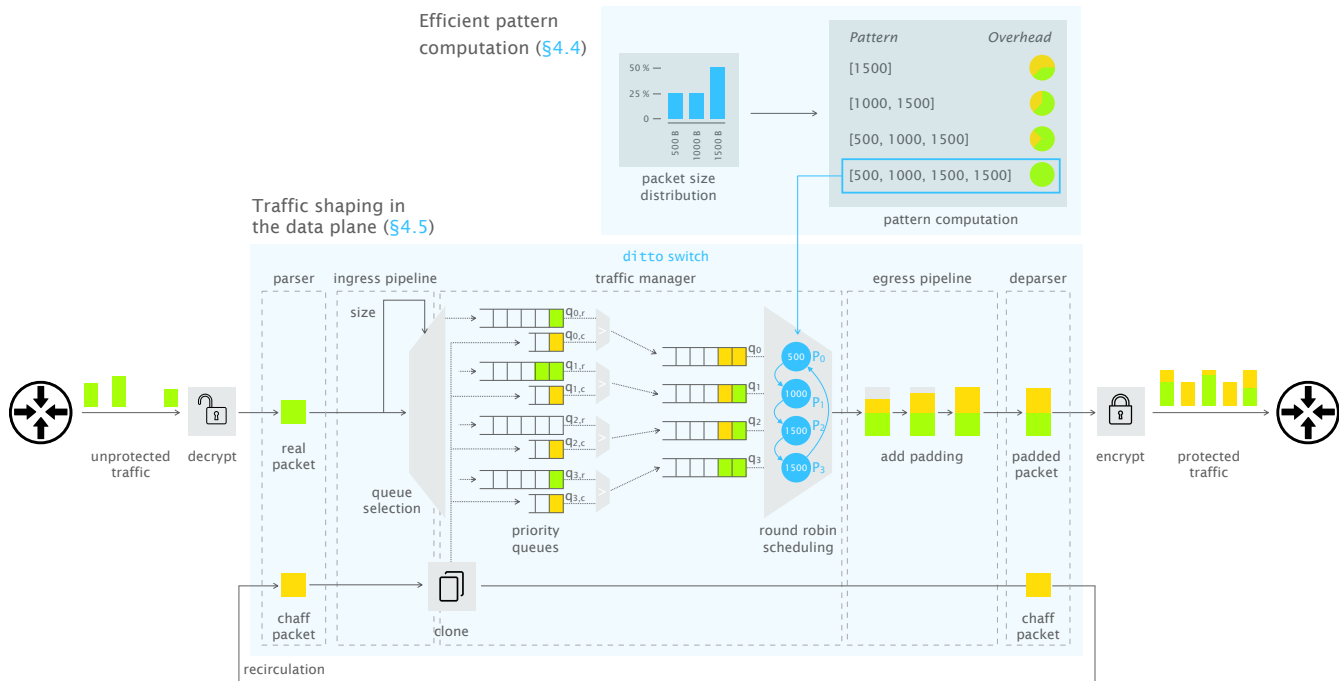


FIGURE 4.3: ditto overview. The combination of priority queuing (real packets have higher priority and there is always a chaff packet ready to send with lower priority) and round-robin scheduling (one queue per pattern state) ensures that the outgoing traffic follows the predefined pattern regarding packet sizes and timing. The figure does not show the removal of padding on the other end of a protected link.

## 4.2 DITTO

In this section, we explain the high-level concepts behind ditto using a running example and Figure 4.3.

**Design goals** The high-level goals of ditto are (i) to make the WAN traffic that an eavesdropper receives independent (in terms of packet sizes, inter-packet time and traffic volume) from the actual traffic that is exchanged over the network; (ii) to support high-throughput networks without degrading their performance; and (iii) to operate without requiring changes to end-devices (e.g., clients or servers).

**Workflow** ditto reaches these goals by running on programmable network devices (no changes to end-devices) and by shaping the incoming WAN traffic into a repeating sequence of packets with pre-defined sizes and timing. With ditto, the traffic actually flowing through the WAN is therefore perfectly independent of the real traffic entering it. A ditto-enabled switch shapes traffic by (possibly): (i) padding incoming packets, to regularize their sizes; (ii) buffering/delaying incoming packets, to regularize their timings and their relative order; and (iii) inserting chaff packets, to fill any possible gaps and ensure the consistency of the packet rates. Of course, enlarging packets and/or delaying them comes at a cost. ditto reduces this overhead by optimizing the shaping pattern.

In the paragraphs below, we rely on a simple example and Figure 4.3 to explain how ditto determines the “shape” of the packet stream (we refer to this as the obfuscation pattern) and how ditto modifies traffic such that it follows this pattern.

**Architecture** ditto has two components: (i) a *pattern computation algorithm* to compute a secure and efficient traffic pattern based on the packet size distribution; and (ii) a *data-plane program* to shape traffic according to this pattern at line rate by padding packets and introducing chaff packets.

**Simple example** We consider a simplistic WAN composed of two ditto switches connected by one link. In this WAN, packets are of three sizes: 25 % of the packets are 500 B, 25 % are 1000 B and 50 % are 1500 B. The ordering of the packets follows an unknown distribution.

**Pattern computation** Given the packet size distribution as an input, ditto first computes an efficient obfuscation pattern. The *obfuscation pattern* spec-

ifies the order and sizes of packets traversing a link protected by ditto. We define it as an ordered list of packet sizes (the pattern states). ditto then repeats this pattern infinitely. For example, if the pattern is [500,1000], ditto shapes the incoming traffic such that the outgoing packet sizes are [500,1000,500,1000,500,1000,...] at a fixed rate.

An efficient pattern minimizes the overhead in terms of padding (bytes added to a packet to make it larger) and chaff packets (dummy packets inserted to transmit at a constant rate). To minimize the amount of required padding, ditto computes the pattern such that it allows to distribute packets uniformly over all pattern states (this leads to minimal padding on average). To minimize chaff packets, ditto prefers short patterns (this reduces gaps between real packets). In Section 4.7, we show that patterns of length 3 to 6 achieve good results.

In the example from above, possible patterns include the ones listed below. For assigning packets to a pattern state, the objective is to minimize the amount of padding. Therefore, each packet is assigned to the next larger pattern state.

- [1500]: This would require to add 1000 B padding to 25% of the packets (the ones of original size 500 B) and 500 B to another 25% of the packets (the ones of 1000 B). But it minimizes the number of chaff packets since all packets can be padded to this size.
- [1000,1500]: Here, the 1500 B packets can be sent in the 1500 B state and the other packets in the 1000 B state. Therefore, ditto only needs to add 500 B of padding to 25% of the packets. However, it needs to send chaff packets if multiple 1500 B packets arrive subsequently.
- [500,1000,1500,1500]: Here, the pattern equals the input distribution and ditto can send each packet without padding. However, it might need to send chaff packets depending on the order in which the real packets arrive.

For the continuation of the example, we use the pattern [500,1000,1500,1500].

**Traffic shaping** The data-plane component of ditto merges incoming real packets with chaff packets such that the mix fits the pattern with minimal overhead. This is challenging because it needs to be performed in hardware to achieve high performance but typical networking hardware is not designed for this. ditto solves this challenges by combining switch queuing and scheduling to hierarchical queues with two levels:

When a packet arrives at a ditto switch, ditto first assigns it to a pattern state. A pattern state is one entry in the pattern; it defines the size which the packet has when it leaves the switch. Since ditto cannot split packets, it assigns a packet to the next-larger pattern state.<sup>2</sup> For example, a packet of size 800 B is assigned to the pattern state 1000 ( $P_1$  in Figure 4.3) such that ditto sends it next time it needs to send a 1000 B packet.

Each pattern state  $P_i$  has two first-in-first-out (FIFO) queues with priorities. A high-priority queue to which ditto assigns real packets ( $q_{i,r}$ ) and a low-priority queue which ditto fills with chaff packets ( $q_{i,c}$ ).

In the example, ditto assigns the 800 B packet to the high-priority queue  $q_{1,r}$  belonging to the pattern state  $P_1$ .

Filling the low-priority queues with chaff packets requires a way to generate these packets. ditto achieves this by continuously recirculating chaff packets and cloning them into the low-priority queues. This does not require a dedicated traffic generator and it does not affect the switch performance (except that it requires one switch port to perform the recirculation).

ditto then feeds the output of each pair of priority queues ( $q_{i,r}, q_{i,c}$ ) to a round-robin queue  $q_i$  and it configures their rates such that the output is  $1/L$ -th of the total rate (for a pattern of length  $L$ ). As a result, each pair of priority queues will output packets at a constant rate and irrespective of whether there is a real packet or not (since there is always at least one chaff packet in each low-priority queue).

In the next phase, ditto performs round-robin scheduling among the round-robin queues ( $q_0, q_1, q_2$  and  $q_3$  in Figure 4.3). Because the order of the round-robin queues corresponds to the obfuscation pattern, the scheduler then outputs packets according to the pattern.

After the queuing and scheduling in the traffic manager, ditto pads the packets in the egress pipeline. At this point, the ordering of the packets is already following the pattern and each packet is marked with its target size. ditto adds padding headers to compensate for the difference between the actual packet size and the target size. For example, it adds 200 B of padding headers for the 800 B packet from above.

Now that the packet has the right size, ditto sends it to the egress port. There, the packet needs to be encrypted (e.g., using MACsec, which can

<sup>2</sup> Fragmentation is often not available on switches or routers for performance reasons. The largest pattern state needs to correspond to the maximum size of any packet in the network (MTU). If multiple states have the same size, ditto distributes packets uniformly among them.

run at line rate [211]) before it leaves the switch. The encryption ensures that an attacker cannot see the padding and she cannot distinguish between real and chaff packets from content analysis.

### 4.3 COMPUTING EFFICIENT TRAFFIC PATTERNS

A naive pattern would be to make all packets equal size. However, this would be inefficient because network traffic contains a variety of different packet sizes and ditto would need to pad them all to the maximum size. For example, Internet backbone traffic is bi-modal (most packets are of size  $< 70$  B or  $> 1400$  B) [212].

In this section, we describe how ditto computes efficient patterns by minimizing the overhead for the expected traffic.

**Definition** An obfuscation pattern  $P$  for ditto is an ordered list of length  $L$  which specifies sizes of packets  $P_i$ :

$$P = [P_0, P_1, \dots, P_{L-1}], \quad P_i \in \mathbb{N} \quad (4.1)$$

Given such a pattern for a link protected by ditto, ditto orders and pads packets such that their size follows  $P$ . That is, the  $j^{\text{th}}$  packet is of size  $P_{j \bmod L}$ .

**Every pattern is secure** We first note that ditto achieves its security goals with every pattern because the pattern is static and therefore does not reveal information about the real traffic traversing a protected link.

**Efficient patterns** Obfuscation patterns differ in their overhead. Intuitively, a pattern is more efficient than another if it requires less padding and buffers/reorders real packets less. In the following paragraphs, we explain how we determine the pattern length  $L$  and its states  $P_i$  to obtain efficient patterns.

**Selecting the pattern length** The pattern length impacts the amount of: (i) padding required, longer patterns require less padding as they can better fit the original traffic distribution; (ii) chaff packets generated, shorter patterns generate less chaff packets because incoming packets are spread over fewer states; and (iii) packet reordering, longer patterns lead to more reordered packets because they require more queues.

In Section 4.7, we show empirically that patterns of length 3–6 achieve good results in all dimensions and for realistic traffic.

**Selecting the pattern states** Since *ditto* iterates over the pattern and sends the same number of packets from each of the states over time, we compute the pattern such that each pattern state fits for  $\frac{100}{L}$  % of the packets. This is the case if the pattern state  $P_i$  is equal to the  $((i + 1) \cdot 100/L)$ -th percentile of the expected traffic distribution  $\mathcal{D}$ :

$$P_i = \text{percentile}_{(i+1) \cdot 100/L} \mathcal{D} \quad i \in [0, 1, \dots, L - 1] \quad (4.2)$$

**When to compute and update the pattern**  $\mathcal{D}$  models the distribution of real packet sizes expected on the protected link. Ideally, the operator computes it based on the real traffic (e.g., recorded prior to using *ditto*). Since this distribution only reveals information about the average traffic characteristics, it is usually not confidential. Otherwise, the operator can use publicly available data such as [213].

When  $\mathcal{D}$  changes significantly, the operator can compute a new pattern and reconfigure *ditto* to use the new pattern without interruption. However, as we show in the evaluation (Section 4.7), the same pattern can be used for many months of real Internet traffic with almost constant overhead.

#### 4.4 TRAFFIC SHAPING IN THE DATA PLANE

In this section, we explain how *ditto* shapes traffic such that it follows the previously defined pattern.

**Problem** A switch running *ditto* receives packets with unpredictable size and at unpredictable times and it needs to ensure that the packets that leave the switch follow the predefined pattern (w.r.t. to packet sizes and inter-packet time). To achieve this, *ditto* needs to perform three operations using the capabilities of programmable switches: (i) add padding to real packets; (ii) buffer packets until they fit in the pattern; and (iii) insert chaff packets.

**Architecture** The architecture of a *ditto* switch is as follows (cf. illustration in Figure 4.3). When a real packet arrives at the *ditto* switch, the parser first extracts information such as the IP header. Then, *ditto* determines the egress port depending on the packet's destination address and assigns it to one of the queues which belong to this egress port. For a pattern of

length  $L$ , each egress port (these are the ports where obfuscated traffic leaves a ditto switch) has  $L$  queues and each queue corresponds to one state in the pattern (and therefore one packet size). The traffic manager then performs round-robin scheduling to send packets from the queues to the egress pipeline. There, ditto adds padding such that the packet's size eventually matches the target size determined by the pattern. Finally, the packet exits at the egress port.

Unfortunately, typical round-robin scheduling has a property that is not optimal for ditto: It skips a queue if it does not contain a packet. This is problematic for ditto because it leads to skipped states in the pattern. To avoid this, ditto makes sure that there is always at least one packet in each queue. If there is no "real" packet available, the switch sends a "chaff" packet.

*Assigning packets to queues* ditto selects the queue to which it assigns a packet such that the amount of padding is minimal. Since ditto can only make packets larger, it selects the next-largest queue  $i$  for a packet of size  $s$  and pattern states  $P_i$ :

$$i = \arg \min_i (P_i - s \mid s \leq P_i) \quad (4.3)$$

If the packet fits into more than one state with the same amount of padding, ditto randomly selects one of them.

*Round-robin scheduling to implement the pattern* ditto configures all queues of an egress port with the same priority such that the traffic manager (TM) performs round-robin scheduling. The TM therefore iterates over all queues and sends one packet from each non-empty queue.

The main challenge to ensure that the sent packets always follow the pattern is therefore to make sure that a queue is never empty when the TM tries to send a packet from it. Ideally, the hardware would allow to inject a "chaff" packet when the TM attempts to send a packet from an empty queue. While this would be a small extension in hardware, there was no need for such a feature so far and thus it does not exist. Below, we describe how we combine priority queueing and round-robin scheduling to overcome this limitation.

*Priority queueing to mix real and chaff packets* To ensure that round-robin queues are never empty, we implement *hierarchical queueing* with two levels. The idea is to combine priority queues with round-robin scheduling. For each pattern state, there is a pair of priority queues: A high-priority

queue ( $q_{i,r}$ ) which receives the real packets belonging to this state, and a low-priority queue ( $q_{i,c}$ ) which is flooded with chaff packets for this state. Each pair of priority queues produces a constant stream of packets while prioritizing real packets. These outputs are then fed into the round-robin scheduling which produces the pattern. We detail the implementation in Section 4.6.

**Custom headers to add padding to packets** After the TM ensured that packets reach the egress pipeline in the right order, ditto adds padding such that they have the right size. ditto pads packets by adding additional headers after the Ethernet header (adding and removing custom headers is something that programmable switches are designed to do at line rate). Because the structure of headers needs to be defined at compile time, ditto uses a combination of multiple headers of different sizes (32, 16, 8, 4, 2 and 1 Byte) to add the right amount of padding to every packet. To allow the receiver to identify padded packets and to remove the padding, ditto marks padded packets in the Ethernet header using the EtherType field. Since a device running ditto encrypts the traffic over the WAN links, ditto can add padding with arbitrary contents and include information about the packet's original size such that the endpoint knows how much padding to remove.

**Removing padding from packets** The receiving switch recognizes padded packets based on their value in the EtherType field and can thus remove the padding without additional information or overhead. To remove the padding from packets before they are sent over a link which is not protected (e.g., to an end host), ditto removes all the padding headers and restores the original EtherType.

## 4.5 SECURITY ANALYSIS AND LIMITATIONS

In this section, we explain why ditto achieves the security goals from Section 4.1.3 and we discuss ditto's limitations.

### 4.5.1 Security goals

**Volume anonymity** The obfuscation pattern together with the link bandwidth defines the traffic volume (in terms of bytes and packets) that is



transmitted over every protected link. Since this volume is static and independent of the real traffic, an attacker cannot learn anything from it other than the maximum number of bytes and packets sent over the link if the link was fully utilized. Naturally, an attacker with access to multiple links and additional background information can derive an upper bound for the total traffic volume (cf. Section 4.5.2).

**Timing anonymity** Because ditto always sends traffic according to the same pattern and at the same rate, it does not leak timing information. Since packets are encrypted such that the ciphertext changes for each WAN link, attackers cannot distinguish real and chaff packets and they cannot determine which packets belong to the same host, application or flow.

**Path anonymity** Even if an attacker can eavesdrop on all links connected to a ditto switch, she cannot link incoming and outgoing packets because (i) the pattern of incoming and outgoing packets is always the same and ditto would rather drop packet than violate the pattern; (ii) she does not know which packets contain real traffic; and (iii) packets are encrypted such that the ciphertext changes for every WAN link.

**Summary** ditto ensures that the traffic seen on each protected link is independent of the real traffic crossing this link. From an attacker's perspective, the traffic always looks the same: packets whose size follows a repeating pattern, with constant inter-packet time and random contents (because of the encryption). Assuming a bug-free implementation and properly working hardware, there is therefore no difference between the observed traffic when there is real traffic and when there is only chaff traffic. Thus, traffic-analysis attacks would produce the same result in both situations and do therefore not work. This also extends to other properties than timing and size, such as packet directions and to multiple colluding attackers.

#### 4.5.2 Limitations

While ditto achieves its security goals and prevents traffic-analysis attacks, there are some limitations and potential attack vectors outside of our threat model. We discuss them below.

**Malicious insider** An attacker who has compromised multiple hosts (e.g., servers in two datacenters connected through a ditto-protected link) can

(i) try to estimate the real traffic volume by measuring the performance of her own traffic; and (ii) exploit ditto for a DoS attack.

To *estimate the real traffic volume*, the malicious insider can leverage the fact that ditto enforces a predefined traffic pattern and that it does not split packets. We illustrate this with a simple example: Assuming the pattern is [500, 1500], the protected link runs at 100 packets per second (pps), the attacker knows that the link can carry at most 50 packets of size between 501 B and 1500 B per second. If the attacker can send and receive 50 packets of size 1500 B per second herself, she can use the observed delays or losses to roughly estimate the amount of other real traffic of this size because otherwise there would have been congestion or losses. Since the traffic created by this attacker is likely untypical for the compromised servers, it could be detected using anomaly detection techniques.

To *exploit ditto for a DoS attack*, the malicious insider can follow a similar strategy as above and she can create her traffic such that it requires a maximum amount of padding. For example, if the pattern is [500, 1500], she can repeatedly send packets of size 64 B (the smallest IP packet size) and 501 B. ditto would pad these packets to 500 B and 1500 B respectively and thereby help the attacker to amplify her volume. Since the attacker's packets compete with benign traffic for "transmission slots", the attacker can partially prevent benign traffic from being sent if her rate is high enough. We see two possible approaches for mitigating this attack: (i) Within each pattern state, packets could be prioritized based on the amount of padding which is required (lower priority for packets which require more padding). Then, packets from such an attacker would be dropped first but this would also impact benign traffic that requires a lot of padding. (ii) The rate at which each user can send packets of a certain size could be limited (e.g., 10 packets with 101 B per second). Packets exceeding this limit could be handled with lower priority such that the measure only has an impact when the network is congested.

**Attacker with insider knowledge** If the attacker has additional knowledge of the network topology, she can potentially also use this to *estimate the real traffic volume*. The best case (for the attacker) is a linear topology where the traffic crosses multiple links and each link uses a different pattern. An extreme example is a linear topology with two links where the first one used a pattern that sends only MTU-sized packets (i.e., 1500 B) and the second one uses a pattern that sends minimal-sized packets of 64 B. If both of these links run at 100 Gbps, the first one transmits around

8 Mpps (million packets per second) and the second one 195 Mpps. If the attacker knows that all traffic crosses both links, she can derive an upper bound of 8 Mpps (because ditto does not concatenate or split packets) and  $8 \text{ Mpps} \times 64 \text{ B} \approx 4 \text{ Gbps}$  throughput (because ditto only makes packets larger). However, this weakness is not harmful in practice because (i) both links would use the same pattern if the pattern was computed as described in Section 4.3; and (ii) it does not break the volume anonymity property.

**Compromised or faulty hardware** ditto runs on network switches and requires them to be trusted and to function properly. If this is not the case, it allows various attacks.

If an attacker has administrative access to a ditto switch, she can easily break ditto's security properties (e.g., by simply disabling ditto). While this is not in our threat model, there are existing techniques to mitigate such attacks [214].

If the used hardware leaks information (e.g., because the scheduling is not working properly or the encryption scheme is weak), this can naturally also weaken ditto's security.

#### 4.6 IMPLEMENTATION

We fully implemented ditto in P4 (traffic shaping) and Python (pattern computation). Since implementing the pattern computation is relatively straightforward, we focus on the P4 implementation, which is technically challenging (cf. Section 4.4).

The source code of our implementation is available on GitHub.<sup>3</sup>

**Hardware target** Our data-plane implementation runs on Intel's Tofino chipset [59], which powers several off-the-shelf switches [215–217] and is used by large operators (e.g., AT&T, Deutsche Telekom, and Alibaba [205, 206]).

**Architecture** The architecture of our implementation follows the description in Section 4.4. In the ingress pipeline, we (i) determine the egress port for the incoming packet; (ii) we check if the egress port is one that we want to obfuscate and whether it is a real packet (as opposed to a chaff packet); if so, we (iii) assign the packet to the right queue; and (iv) check if the switch

<sup>3</sup> <https://github.com/nsg-ethz/ditto>

can add enough padding in one pipeline pass or if the packet needs to be sent through the pipeline multiple times.

**Approximating hierarchical queueing** As explained in Section 4.4, one challenge behind *ditto* is that the switch needs to send a packet from each round-robin queue *even if the queue is empty*. This is not possible in existing switches. For *ditto*, we implemented an approximation of hierarchical queueing, where a packet traverses two queueing stages instead of just one. We achieve this by sending each packet through the switch data plane twice. As illustrated in Figure 4.4, the first queueing stage consists of one pair of priority queues for each pattern state. The queue with the higher priority receives all real packets belonging to the respective pattern state, while the queue with the lower priority is flooded with chaff packets.

We set the output rate of each priority-queue pair such that the sum of all pairs equals the total sending rate of the switch. For example, if the switch sends 10 Mpps and  $L = 4$ , each queue pair needs to transmit 2.5 Mpps. The outputs from the priority queues are fed back to the switch via loopback ports.

When the packets arrive at the switch for the second pass, *ditto* sends them to the round-robin scheduler attached to the actual egress port. Its output then follows the pattern.

The main cost of sending each packet through the switch twice is that the loopback modules occupy ports which cannot be used for interconnections with other switches. Since each physical 100 Gbps QSFP port can be split into 2 or 4 sub-ports (with 50 Gbps or 25 Gbps throughput each, respectively), one loopback port can be used for up to 4 pattern states. The bandwidth of one sub-port needs to be at least  $100/L$  Gbps where  $L$  is the length of the pattern. Since the bandwidth can only be 100, 50, or 25 Gbps, *ditto* configures it as follows:

$$bw = \begin{cases} 100, & L = 1 \\ 50, & L = 2 \text{ or } L = 3 \\ 25, & L \geq 4 \end{cases} \quad (4.4)$$

And the number of required loopback ports  $n$  computes to

$$n = \left\lceil \frac{L \cdot bw}{100} \right\rceil \quad (4.5)$$

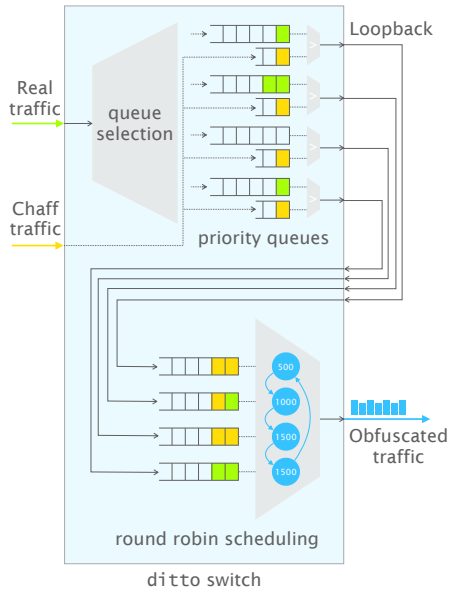


FIGURE 4.4: ditto implements hierarchical queuing by sending each packet through the switch twice.

For example, patterns of length 3 and 6 require 2 loopback ports for each obfuscated port. In the typical use case where an organization uses a ditto switch as its gateway to the WAN, switch ports are not scarce (e.g., a typical switch has 64 ports [59] and could therefore support around 20 WAN links).

**Adding padding** In the egress pipeline, ditto adds padding to the packet until it has the required size. ditto adds padding in the form of additional headers of different sizes (between 32 B and 1 B). ditto adds the padding headers in decreasing order in separate stages to reduce the number of match & action table entries: First, it adds as many 32 B padding headers as possible (and needed). Then, it tries to fill the remaining space with 16 B headers and so on until there is no more padding needed. For example, ditto adds  $2 \times 32$  B,  $1 \times 4$  B and  $1 \times 2$  B padding headers to increase a packet's size by 70 B.

**Recirculate packets if needed** If the required amount of padding is larger than what can be added in one pipeline pass, ditto *recirculates* the packet. Then, the packet traverses the switch multiple times (i.e., from the end of

the egress pipeline, it is sent to the beginning of the ingress pipeline). With each pass, ditto can add additional 254 B of padding. By default, ditto uses two 100 Gbps ports for the recirculations. Depending on the pattern and the traffic distribution, this can result in a bottleneck especially if ditto protects traffic on multiple ports. In this case, ditto can load-balance the recirculations over more ports (thereby reducing the number of available switch ports for other interconnections).

**Removing padding** Removing padding is straightforward: ditto removes all the padding headers and restores the original EtherType. Again, the limited size of the PHV (Packet Header Vector, cf. Section 2.3) can require recirculating the packet in order to remove all padding.

**Resource usage** The main bottleneck regarding resource usage of ditto is the amount of padding that can be added in one pipeline pass. Because ditto adds padding in the form of additional headers, the amount is limited by the size of the PHV and the deparser. In our current implementation, ditto can add up to 254 B of padding in one pipeline pass. Regarding other types of resources (e.g., SRAM and TCAM memory), our implementation uses only a small fraction of the switch's resources (less than 10% on average over all stages).

## 4.7 EVALUATION

We first describe our methodology in Section 4.7.1. Then, we evaluate our implementation with respect to performance (Section 4.7.2) and security (Section 4.7.3). In addition, we outline the potential of future hardware optimized for ditto through simulations (Section 4.7.4). Table 4.1 summarizes our main results.

### 4.7.1 Datasets and methodology

The distribution of traffic processed by ditto depends on the type of WAN in which it is deployed. As a typical use case, we envision an organization with several sites which uses its WAN to connect them and use one site as a gateway through which it sends all outgoing and incoming traffic. In this case, WAN traffic has similar characteristics as Internet traffic. In addition,

---

## Results from hardware measurements

### *Performance and efficiency*

Section 4.7.2

- Longer patterns reduce the padding and chaff overhead.
- ditto's padding strategy leads to less overhead compared to related work.
- Non-interactive traffic (replayed traces): no performance loss for link loads between 60 and 70 Gbps (depending on the dataset).
- Interactive traffic (iPerf, VoIP and web browsing): no performance loss for link loads between 70 and 80 Gbps (depending on the dataset).

### *Security*

Section 4.7.3

- Packet sizes and timing do not allow conclusions about real traffic.

## Results from simulations

### *Performance and efficiency*

Section 4.7.4

- Longer patterns reduce the padding and chaff overhead.
- 1 MB of buffer space is enough to obfuscate a traffic volume of up to 99% of the link bandwidth.
- The same pattern can be used for months without sacrificing efficiency.
- 92% of the packets remain in the correct order for highest load and the longest pattern.

### *Security*

- The pattern produced by ditto is secure by design.
- 

TABLE 4.1: Evaluation summary

we consider two extreme cases: (i) the best case for ditto where all packets have the same size (e.g., if there was heavy traffic shaping); and (ii) the worst case for ditto where the packet sizes are uniformly distributed.

**Datasets** We use real Internet traffic and synthetic traffic where the packet sizes follow given distributions. For the real Internet traces, we use the publicly available CAIDA anonymized Internet traces dataset [213] (CAIDA). Even though this dataset was collected on Internet links, we believe it is representative for a WAN where one site is used as a gateway for all incoming and outgoing traffic (e.g., for central compliance monitoring). We use the most recent dataset (captured in January 2019) and we preprocess it in two steps: (i) we remove all non-IPv4 packets because the current

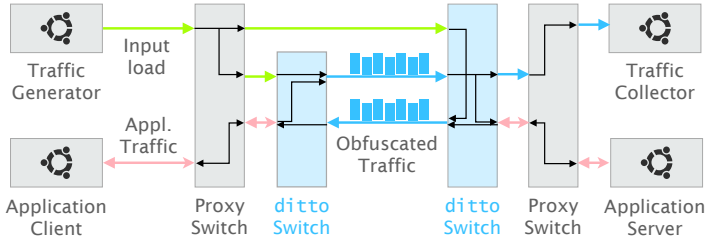


FIGURE 4.5: Evaluation setup. We use two ditto switches, two switches that act as proxies to perform measurements (e.g., timestamps), and two servers to send and receive traffic.

implementation of ditto can only handle IPv4 traffic;<sup>4</sup> and (ii) we extract the first 100 M packets to speed up our simulations.

In addition, we generate two synthetic traffic traces using Scapy [218]: one where all packets have size 1480 B (CONSTANT) and one with uniformly distributed packet sizes between 60 B and 1480 B (UNIFORM). While these traffic distributions are unlikely to occur in practice, we use them to represent two extreme cases: CONSTANT is the best case for ditto because it already follows a constant pattern while UNIFORM represents the worst case because fitting a pattern to a uniform distribution creates the highest overhead.

Many WANs will have different traffic distributions than the traces that we use in the evaluation, but we argue that our datasets are useful to show ditto’s performance on a wide range of different traffic characteristics. It is also important to note that the traffic distribution does not have an impact on ditto’s security properties.

All datasets mentioned above represent non-interactive traffic. That is, the traffic does not change depending on the network behavior. While this allows us to test ditto with high traffic volumes (up to 100 Gbps), it is not fully representative for real-world behavior (e.g., a dropped TCP packet would be retransmitted). To address this, we run *interactive applications* on top of the replayed traffic and we measure the performance achieved by these applications. We describe this in Section 4.7.2.

**Methodology** We evaluate ditto on off-the-shelf devices and we simulate how it would perform on ideal future hardware.

<sup>4</sup> This is only an implementation detail. The same approach would also work for IPv6 packets.



The *hardware prototype* is our implementation as described in Section 4.6. It runs on two Intel Tofino switches with  $32 \times 100$  Gbps ports. Between the switches, ditto obfuscates traffic on one link at 100 Gbps. We use a traffic generator (Moongen [219]) to inject traffic from one server, and to record traffic on another server (cf. Figure 4.5).

We also implemented a *simulator* for optimal hardware in Python. It receives a packet sequence as input (the real traffic) and produces a packet sequence as output (the obfuscated traffic). The simulator operates in discrete time steps: in each iteration, it receives and sends one packet. To shape traffic according to the pattern, it uses round-robin scheduling and when there is no real packet to send it sends a chaff packet.

#### 4.7.2 Performance and efficiency in today's hardware

In the following paragraphs, we show the performance of ditto with respect to these three aspects:

- *Throughput*: The ratio between the incoming and the outgoing real traffic
- *Recirculations*: The number of recirculations of each packet
- *Application performance*: The performance of interactive applications

**Throughput** Figure 4.6 shows the ratio between incoming real traffic and outgoing real traffic. To obtain these results, we send traffic from a server to a first switch where we add additional information to packets that we need for our measurements (e.g., we add a number to each packet). Then we send it to the switch which runs ditto. Afterwards, we record the obfuscated traffic on another server (see Figure 4.5).

The total outgoing traffic is always 100 Gbps (not shown in the plot). If there was no performance loss, the amount of incoming traffic would equal the amount of outgoing traffic. But since ditto makes packets larger and fits them into a predefined pattern, it creates overhead and therefore reduced the usable throughput. However, as Figure 4.6 shows, the hardware prototype operates almost without loss until 90% (CONSTANT), 70% (UNIFORM) or 60% (CAIDA) load.

The reasons for this sub-optimal performance include:

- *ditto* relies on precisely controlled output rates of priority queues such that the output does not fluctuate. However, today’s switches are typically not designed for that (they offer traffic shaping, but the rate is only correct “on average”). In our case, bursts of too much traffic lead to dropped packets.
- *ditto* adds padding in the form of additional headers. Treating padding like packet headers is expensive with respect to the required resources in the pipeline and the deparsing time, and a switch that could add padding without using these resources could perform better.
- *ditto* uses a maximum bandwidth of 100 Gbps for recirculation. Therefore, if packets need to be recirculated multiple times, this bandwidth is not enough and packets get lost.

*Comparison with related work* In Figure 4.6 we also show an upper bound for the performance of three systems that rely on end-host protocols to obfuscate traffic: HORNET [104], TARANET [106] and BuFLO [121]. To compute the performance of these systems, we only simulate their padding strategy (i.e., we neglect computational overhead and overhead due to mixing with chaff packets), hence the real results of these systems would be strictly worse than the plotted upper bound.

Our results show that *ditto* outperforms all of these approaches even if their computational overhead is ignored (the results for *ditto* are measurements on actual hardware and therefore include all forms of overhead). *ditto* outperforms these approaches because (i) *ditto* adds padding according to an efficient pattern, which produces less overhead than padding all packets to the same size; and (ii) operates on a per-link basis as opposed to obfuscating each flow separately.

Below, we provide more details about the simulated parameters.

*HORNET* pads all packets to the same size  $s$ . We set  $s$  to the size of the largest packet in the respective dataset, which minimizes the padding. Further, *HORNET* adds headers to the original packets. The size of these headers depends on the AS path length and a sample length. We set the AS path length to 1 and the sample length to 16 (as in the paper [104]).

*TARANET* shapes flow(lets) such that they transmit at a fixed rate (constant size and inter-packet time) and it adds chaff packets to obfuscate the real length of a flow(let). In contrast to *ditto*, *TARANET* can also split packets to make them smaller. For our simulations, we do not consider

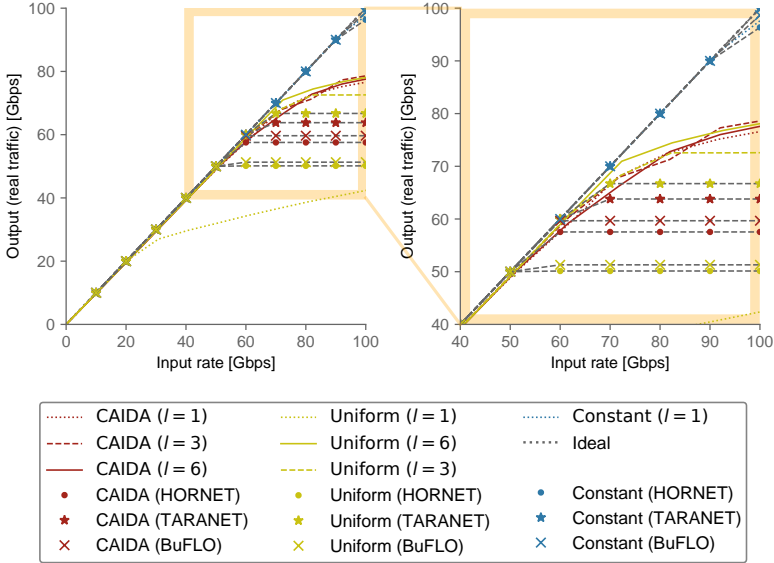


FIGURE 4.6: Input vs. output rate of real traffic. Longer patterns and constant size traffic lead to higher goodput. ditto outperforms related work even their computational overhead is ignored.

chaff packets (which makes the results only better) and we set the packet size to the average packet size of the respective dataset.

*BuFLO's* padding strategy is to pad all packets to the MTU (1500 B in our case) without modifying the timing. Again, we only simulate the padding overhead without considering chaff packets. Ideally, this approach can work without additional headers which is why we assume there is no such overhead.

**Recirculations** Since our hardware prototype can only add 254 B of padding per pipeline pass, some packets need to traverse the switch multiple times. For this, we use a technique called recirculation which sends a packet from the end of the egress pipeline back to the beginning of the ingress pipeline.

Recirculating packets increases reordering and packet delay and (because the bandwidth for recirculation is limited) can lead to packet loss. Therefore, it is better to keep the number of recirculations small. The best strategy to reduce recirculations is to make patterns long enough such that the

difference between each pattern state and the next smaller or larger one is less than the number of padding bytes per pipeline pass.

CONSTANT does not require recirculations. For CAIDA and UNIFORM, the number of recirculations decreases with an increasing pattern length. A pattern of length 1 requires 1.59 (CAIDA) or 0.99 (UNIFORM) recirculations on average. This numbers decrease to 0.23 and 0.40 for a pattern of length 3 and 0.18 and 0.03 for length 6.

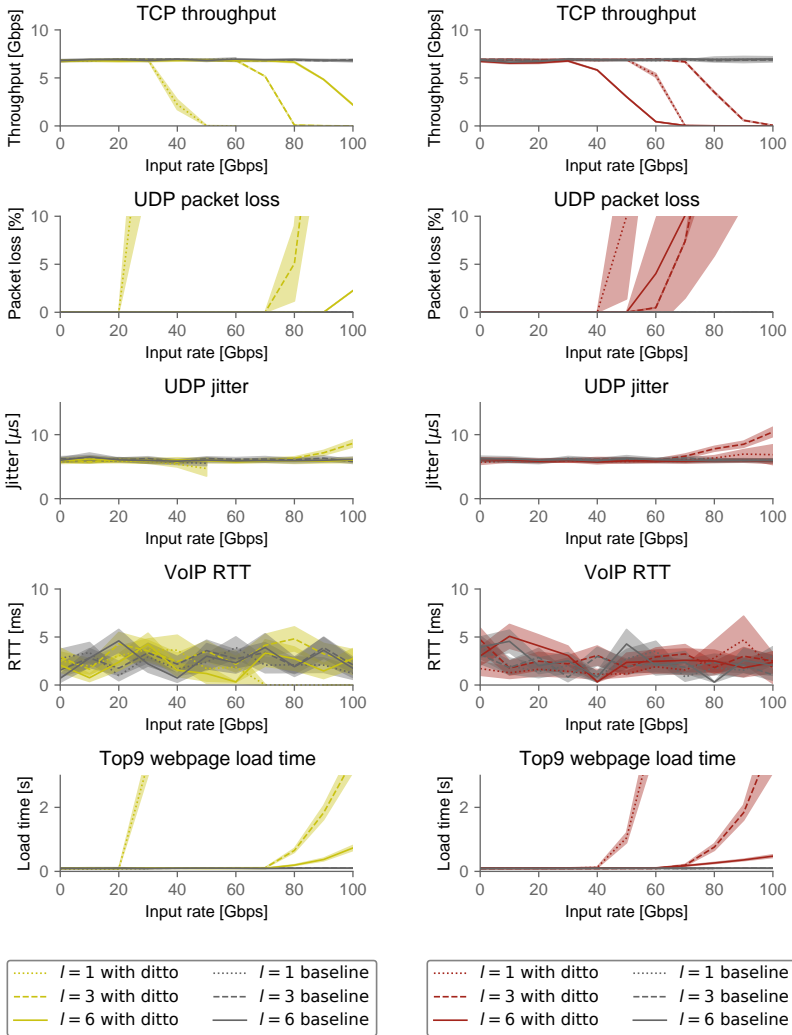
**Application performance** In addition to the raw throughput measurements above, where we replayed static traffic traces, we now measure the performance of interactive applications.

We measure the real-world performance of three typical types of applications: (i) high-bandwidth TCP and UDP traffic (using iPerf [220]); (ii) web browsing traffic (using WprGo [221]); and (iii) VoIP traffic (using pjsip [222]). Figure 4.5 illustrates the setup of this experiment. We run Docker containers with clients and servers for the respective application and we measure the performance of these applications when the traffic is sent via a ditto-enabled link versus the performance when the traffic is forwarded directly (we use this as the baseline).

Figure 4.7 shows the results of these measurements. We obtain the results by running 50 measurements for each input rate (0–100 Gbps) and we perform each measurement with and without ditto. We highlight that we perform the measurements *in addition* to the replayed traffic. Therefore, a TCP throughput of 7 Gbps at an input rate of 80 Gbps means that ditto is handling 87 Gbps in total. The measurements without ditto represent a baseline where traffic is directly forwarded.

We measure the following metrics:

- *TCP throughput*: Important for applications which demand high bandwidth and reliable throughput (e.g., file transfer). We measure TCP throughput by creating 30 s TCP flows with maximum throughput using iPerf.
- *Packet loss*: Important for applications using unreliable transport (e.g., video streaming over UDP) and reliable transport (e.g., TCP). We measure packet loss by creating 30 s UDP flows with 1 Gbps throughput using iPerf.
- *Jitter*: Important for real-time applications and streaming (e.g., video calls). We measure jitter by creating 30 s UDP flows with 1 Gbps throughput using iPerf.



(a) UNIFORM

(b) CAIDA

FIGURE 4.7: ditto compared to the baseline. Lines show mean values and colored areas indicate the 95% confidence interval. ditto affects the application performance after a certain link load, depending on the dataset, pattern length and metric.

- *Round-trip time (RTT)*: Important for real-time applications (e.g., VoIP). We measure the RTT by creating  $8 \times 30$  s VoIP calls using PJSIP.
- *Website load time*: Important QoE metric. We measure the load time for the 9 most popular websites according to Alexa [223].<sup>5</sup>

The results in Figure 4.7 show that – as expected – ditto does not degrade the network performance up to a certain point. Depending on the distribution of the traffic and the length of the pattern, ditto does not have significant impact on the network performance with an input rate of up to 80 Gbps (TCP throughput, UNIFORM, pattern of length 6). The main causes for the degraded performance are dropped packets (due to the same reasons as discussed above). If packets are not dropped, we highlight that ditto has no significant effect on timing-related metrics such as jitter and Round-Trip Time (RTT).

In general, longer patterns are more efficient because they produce less overhead. Most of our results confirm this hypothesis. However, there are two exceptions in the CAIDA dataset: TCP throughput and packet loss. The reason for this is that we do not take the interactive application traffic into account when we computed the pattern. Then, it can happen that the measurements modified the traffic distribution to an extent where the pattern does not fit well anymore. Especially for longer patterns (because each state of the pattern has a small share of the total amount of transmitted packets, e.g.,  $1/6$ th for  $L = 6$ ) and for flows with many equal-sized packets (because then all packets are assigned to the same pattern state). Both the TCP and the UDP measurements consist of constant-size packets because iPerf maximizes the throughput.

### 4.7.3 Security in today's hardware

We now show that the hardware prototype obfuscates traffic such that the observed inter-packet times and packet sizes are independent of the real traffic and we show that the accuracy of a state-of-the-art attack is on par with random guessing.

***Packet timings are independent of the real traffic*** Figure 4.8 shows the Inter-Packet Gap distribution (IPG, the time between the end (last byte) of the previous packet and the start (first byte) of the current packet) for

---

<sup>5</sup> amazon.com, facebook.com, netflix.com, reddit.com, youtube.com, zoom.us, bing.com, google.com, and wikipedia.org

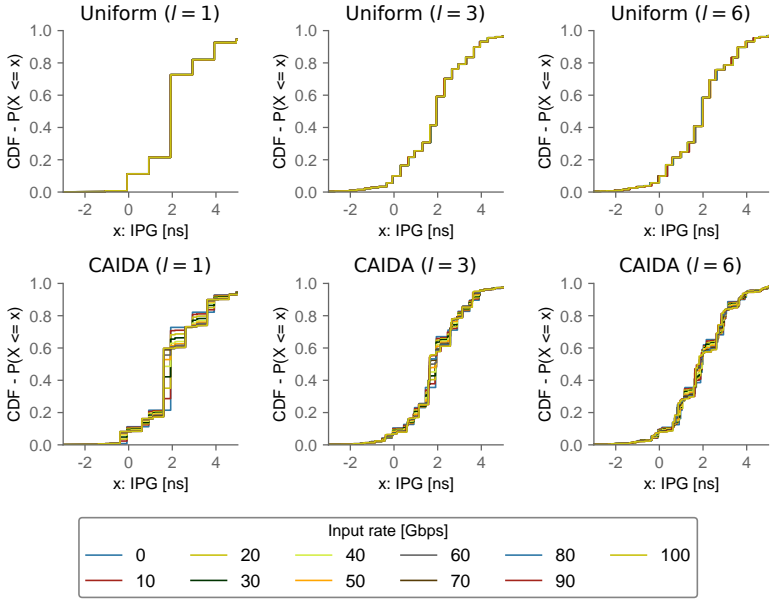


FIGURE 4.8: IPG distributions. The IPG does not depend on the input rate (the 11 lines in each plot are overlapping).

each dataset, pattern length, and network load. Visually, it is clear that the distributions do not depend on the network load (the lines largely overlap). Analyzing the measurements further shows that a large fraction is within a typical error margin around the median value. For example, an attacker who can measure timestamps with a precision of  $\pm 3.2$  ns (as in a state-of-the-art capturing device [224]), could not distinguish between 92% and 97% (depending on dataset and pattern length) of the measurements.

The numbers in Figure 4.8 are subject to imprecisions for two reasons: (i) we measure the timestamps in the ingress pipeline of the evaluation proxy switch (cf. Figure 4.5), i.e., not on the link directly and not on a calibrated measurement device; and (ii) we measure the timestamp at the beginning of a packet, while the IPG refers to the time between the last byte of the previous packet and the first byte of the current packet. We therefore adjust the timestamp computationally as follows. For two packets of sizes

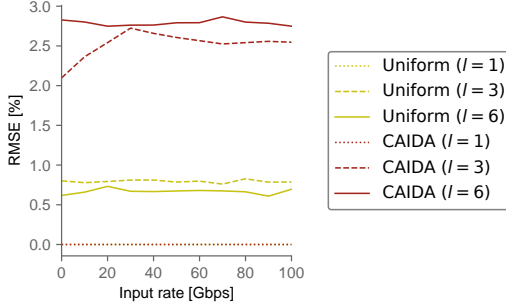


FIGURE 4.9: ditto performs round-robin scheduling up to an error which does not depend on the input rate.

$s_0$  and  $s_1$  bytes arriving at timestamps  $t_0$  and  $t_1$  ( $t_0 < t_1$ ) and a line rate of 100 Gbps, the IPG is

$$IPG = t_1 - t_0 + \frac{s_0 \cdot 8}{100 \cdot 10^9} \quad (4.6)$$

**Packet sizes are independent of the real traffic** We now evaluate whether the hardware prototype obfuscates traffic such that it follows the defined pattern. Round-robin scheduling in today’s switches is designed to follow round-robin behavior *on average*, not necessarily in microscopic detail. This means that the switch performing round-robin scheduling could send  $[P_1, P_1, P_2, P_2, P_3, P_3]$  instead of  $[P_1, P_2, P_3, P_1, P_2, P_3]$  (where  $P_i$  represents a packet from queue  $i$ ). However, since this behavior originates in the hardware implementation of the switch and not in the behavior of ditto, it does not affect ditto’s security.

In Figure 4.9 we show that the hardware prototype indeed performs round-robin scheduling on average and it produces a distribution which is close to uniform. In the plot, we show the Root Mean Square Error (RMSE) between the observed distribution and a uniform distribution. The error does not leak information about real traffic. Instead, it originates from the approximation of the 2-level hierarchical queueing and the required precise rate-control, which is more error-prone for small packets.

**State-of-the-art attack does not work with ditto** After showing that packet sizes and timings are independent of the real traffic and do not leak



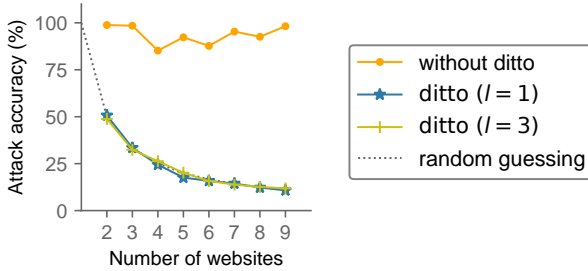


FIGURE 4.10: DF attack accuracy. For ditto-protected traffic, the accuracy is on par with random guessing.

information in general, we now run a state-of-the-art attack to showcase that ditto is robust against this particular attack.

We run the Deep Fingerprinting (DF) attack developed by Sirinam et al. in [126]. This attack uses convolutional neural networks to identify visited websites. The inputs for training, validation and testing are sequences of packet directions (e.g., [1, -1, -1] when loading a website required one outgoing packet and two incoming ones).

To run the attack, we used the code published by the authors and the same parameters and input dimensions (1000 samples per website, at most 5000 packet directions per sample). To load the websites, we used the same setup as for the application performance experiment above and we added 5 ms latency between the two containers to make it realistic for real Internet traffic. We loaded the Alexa top 9 websites and recorded the traffic on the link protected by ditto using tcpdump [225]. In order to be able to record the traffic, we run ditto at only 500 Mbps per direction.<sup>6</sup> From the recorded traffic, we extracted the packet directions for 5000 packets starting with the first packet sent from the client to request a website. This makes it easier for the attack as it would be in practice because a real attacker could not distinguish between real and chaff packets and thus she could not determine the first packet of a request. We run the attack in the closed world setting as in [126], where there is no other real traffic besides the loaded website (which makes it easier for the attack) and we run the attack for traffic recordings containing between 2 and 9 websites (identify fewer websites is easier for the attack).

<sup>6</sup> The attack would not perform better for a higher bandwidth.

We depict the accuracy of the attack in Figure 4.10 with and without ditto (each point in the plot is the average accuracy over 20 attack runs). We also depict the accuracy of an attacker randomly guessing (for reference). We see that the attack is unsuccessful on ditto-protected traffic: the accuracy is on-par with random guessing. Note that the results hold independently of the pattern length (here, 1 or 3). We also see that the attack is successful without ditto, as expected.<sup>7</sup>

#### 4.7.4 Performance and efficiency in future hardware

We now simulate how ditto would run on future hardware with two extensions compared to our hardware: (i) the round-robin scheduler can directly send a chaff packet if a queue is empty; and (ii) the number of padding bytes is not limited.

In the following paragraphs, we show the simulated performance of ditto with respect to these two aspects:

- *Overhead*: Amount of padding and chaff packets depending on the input load and the pattern length
- *Reordering*: Out-of-order packets in TCP flows depending on the input load and the pattern length

**Overhead** We simulate ditto with different input loads (10–100 %) and pattern lengths (1–32) to evaluate the overhead added to real traffic. To measure overhead, we use 4 metrics:

- *Chaff overhead*: The number of chaff bytes that ditto sends to always transmit at line rate
- *Padding overhead*: The padding that ditto adds to packets in order to fit into the pattern
- *Buffer space usage*: The required buffer space to store packets until they fit in the pattern
- *Switching delay*: The number of packets that ditto transmits between two packets that arrive subsequently

---

<sup>7</sup> We observed that the model is overfitting in some cases for the unprotected datasets. To limit this, we added an early stopping mechanism that stops the training when there was no significant improvement in the last 3 epochs [226].

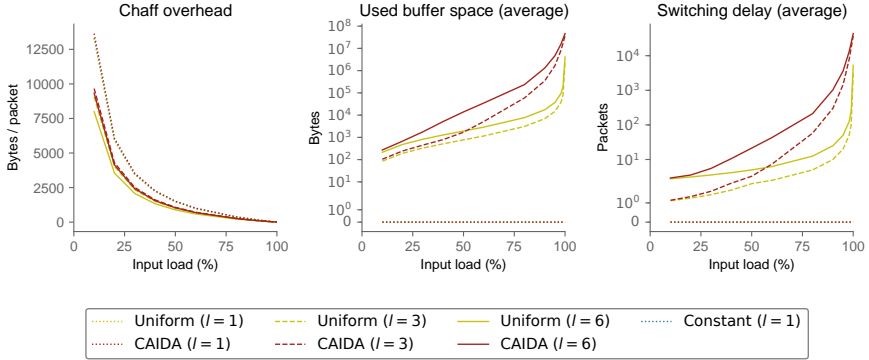


FIGURE 4.11: Overhead depending on the input load. ditto’s overhead in terms of buffer space and introduced switching delay is small up to an input load of around 80%.

**Overhead depending on the network load** Figure 4.11 shows how the network load impacts the overhead created by ditto.

We show the results for all three datasets. For CAIDA and UNIFORM, we show the results for different pattern lengths (1, 3 and 6). For CONSTANT, we only show a pattern of length 1 because this is already the most efficient pattern. As expected, longer patterns fit the actual traffic distribution better and therefore create lower padding- and chaff overhead.

The *chaff overhead* decreases with increasing input load because the more real traffic there is to send, the fewer chaff packets need to be added.

We observe that the chaff overhead is larger for short patterns. This is because short patterns need to consist of larger packets (e.g., a pattern of length 1 results in sending MTU-sized packets constantly) and therefore, even if there are fewer chaff packets, the amount of chaff *bytes* is larger.

ditto needs to buffer packets when the sequence of the pattern does not match the sequence of the incoming packets. For small loads, this is less critical because there is more time between two subsequent incoming packets (e.g., if the time between two incoming packets is larger than the time it takes to iterate over the pattern once, the buffer is always empty when the new one arrives). For high input loads, discrepancies between the sizes of incoming packets and the outgoing pattern have a higher impact.

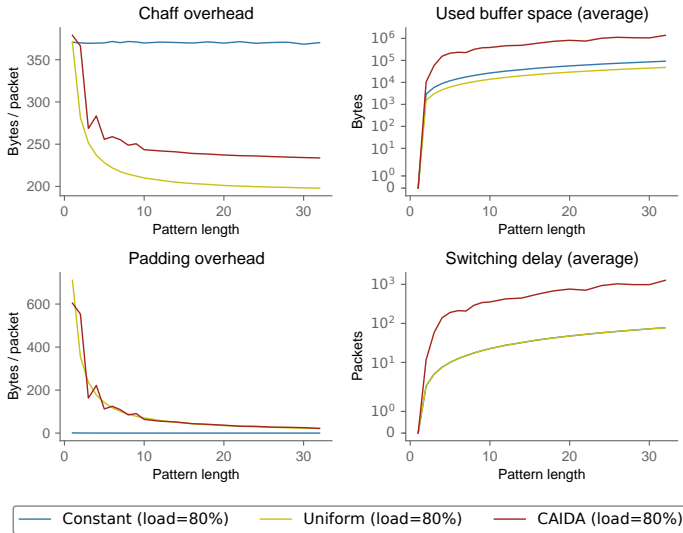


FIGURE 4.12: Overhead depending on the pattern length. Long patterns result in small padding and chaff overhead but require more buffer space and introduce more delay.

However, as Figure 4.11 shows, 1 MB of buffer space is sufficient for up to 90 % (CAIDA) or 99 % (UNIFORM) load. Patterns of length 1 do not require buffering because ditto pads and sends each packet immediately.

The *switching delay* measures the number of packets sent between two incoming packets. Therefore, it evolves similarly to the buffer overhead. Again we observe a slow increase until the switch starts to get congested around 90 % network load.

**Overhead depending on the pattern length** Figure 4.12 shows how the pattern length impacts the overhead. The results are for a high network load of 80 %, which means that there are usually real packets ready to be sent in each pattern state. In this case, longer patterns result in less *chaff overhead* and *padding overhead*. This is because longer patterns fit the actual traffic distribution better and therefore require sending less additional traffic. At the same time, longer patterns lead to increased *buffer usage* and *switching delay* because it is more difficult to fit the incoming packets to the right pattern state.

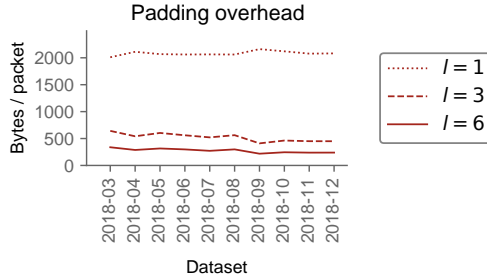


FIGURE 4.13: Overhead for using the same pattern (length  $l$ ) over 10 subsequent months. The padding overhead does not increase because the traffic distribution roughly stays the same.

**Overhead for long-term use of a pattern** ditto computes the pattern based on the packet size distribution and then applies it for future traffic. This is always secure, but not necessarily efficient. If the distribution of the real traffic changes, it is worth computing a new pattern (which can be deployed without interrupting the switch [227]). We confirm with the results in Figure 4.13 that ditto can apply the same pattern over a long period (10 months) with a nearly constant overhead.

**Packet reordering** We now evaluate the impact of ditto on the ordering of packets. We again simulate ditto with different input loads (between 10 and 100 %) and different pattern lengths (1–32).

We focus on CAIDA in this experiment because the other datasets do not contain TCP flows. We randomly select 100k TCP flows with at least 2 packets from CAIDA and count the number of reordered packets for each of them (the sampled flows are the same across all experiments).

To measure reordering, we use the following metrics:

- *Reordered packets*: Packets that were out of order (i.e., packet  $i$  arrived before packet  $i - 1$ )
- *Flows with reordered packets*: Flows with at least one reordered packet

**Reordering depending on the network load** Figure 4.14 shows how the network load impacts the reordering. As expected, a higher input load leads to more reordering. But even in fully loaded networks, less than 8 % of the packets are reordered. 8 % of reordered packets impact at most 47 % of flows because many short flows are reordered.

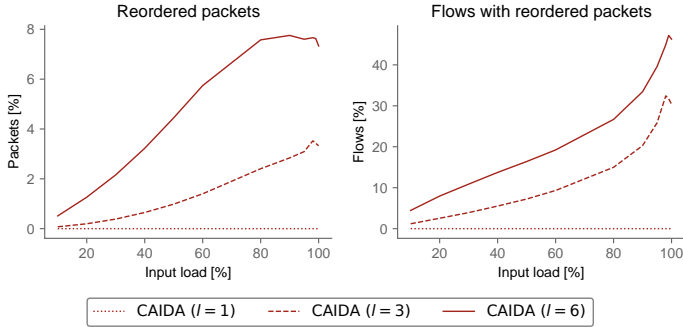


FIGURE 4.14: Packet reordering depending on the input load. 92 % of the packets remain in order for the highest load and the longest pattern ( $l = 6$ ).

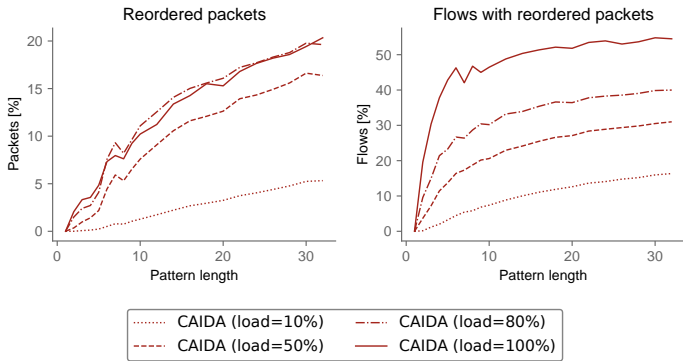


FIGURE 4.15: Packet reordering depending on the pattern length. Longer patterns lead to more reordering.

We point out that these results show a worst case because of the way how our simulator works. The original throughput of the CAIDA dataset is around 4 Gbps [212], which corresponds to an input load of 4 % in our simulation. To simulate higher input loads, we replay CAIDA at a higher speed (up to  $25\times$  the original speed), which creates unrealistically high-bandwidth flows. For example, a user downloading a file with 1 Gbps is simulated as a user with a 25 Gbps connection.

**Reordering depending on the pattern length** Figure 4.15 shows how the pattern length impacts reordering. As expected, longer patterns lead to more reordering because the sequence of outgoing packets is more constrained.

## 4.8 RELATED WORK

Preventing traffic-analysis attacks has been an active research area for many years. However, existing work focuses on preventing traffic-analysis attacks for Internet users. As we elaborated earlier, these systems are largely orthogonal to ditto because protecting WAN traffic presents both new challenges (e.g., high throughput) and opportunities (e.g., control over network devices). Even though existing systems could be applied in WANs too, they would not perform well enough (cf. simulations in Section 4.7) since they are not optimized for this setting or they require modifications of the end hosts.

Most existing work is application-specific (e.g., to prevent website fingerprinting [228–231], ensure anonymous communication [204, 232] or protect IoT devices [233]) and/or requires additional servers to relay traffic [136, 203, 234]. In contrast to these approaches, ditto operates at the network layer, protects all traffic and does not need additional servers or modifications at the clients. Below and in Table 4.2, we summarize the most relevant work related to ditto.

Widely used protocols and libraries already allow adding random number of bytes to the plaintext before encrypting it. Examples include SSH [235], GnuTLS [231], and IPSec [236]. However, this only adds a small amount of anonymity (the volume increases by a random amount within some bounds) and it does not provide timing- or path anonymity.

*Onion routing and mix networks* TOR [101, 234], the most widely used anonymity network today, and similar systems (e.g., [104, 237, 238] use onion routing [239] to hide the source and the destination of traffic. However, they do not prevent timing attacks and they do not hide the traffic volume.

HORNET [104] is similar to TOR in the sense that it uses onion routing but it operates on the network layer. To obfuscate the traffic volume, HORNET adds padding to packets such that all packets have the same size.

PriFi [204] is based on Dining Cryptographers networks (DC-nets) [240] where each participating node is assigned a time slot in which it can (and must) send a message. This provides volume-, timing- and path anonymity because the observed traffic is always the same, but it reduces the total throughput linearly with the participating nodes.

Loopix [203] mixes real and chaff traffic in dedicated mix nodes and achieves low-latency communication with up to 300 messages per second

while hiding the sender and receiver of messages as well as whether they are currently active.

***Padding and traffic shaping*** Like *ditto*, several works aim at hiding the traffic volume by adding padding, chaff packets and/or by sending packets according to a predefined schedule. Examples of such systems include the works of Guan et al. [135], Wang et al. [136] and Wright et al. [241].

In [242], Dyer et al. show that existing padding approaches do not provide enough security and they suggest BuFLO as a solution with high security. BuFLO [242] pads all packets to the same size; delays them such that the time between two packets is constant; and sends chaff packets such that each flow has a certain minimal length. However, as the authors state in the paper, this approach is inefficient because of the constant packet size and inter-packet time. The key difference to *ditto* is the deployment scenario: BuFLO runs on end hosts and obfuscates each flow individually. This leads to the large overhead mentioned in the paper. *ditto* runs in the network and obfuscates traffic on a per-link basis according to an efficient pattern (instead of making all packets constant-size), which leads to less overhead. Furthermore, *ditto* does not leak information about flow durations or sizes.

In [134] and [137], Cai et al. present CS-BuFLO, an improved congestion-sensitive version of BuFLO. CS-BuFLO adapts the transmission rate depending on how much traffic the client tries to send. This makes it more efficient but also less secure because it leaks information about the sender's volume. While CS-BuFLO has less overhead than BuFLO, it suffers from similar limitations: Since the padding happens per flow or per device, the overhead created by many flows or devices sums up in the network. Further, CS-BuFLO leaks information about the total volume of a flow or device and the sending rate while *ditto* does not because it runs in the network.

TARANET [106] shapes traffic into constant-rate flowlets at the hosts. The system then makes sure that these flowlets achieve the constant rate despite dynamic network events such as packet loss. Similarly to *ditto*, TARANET mixes real and chaff packets, but in contrast to *ditto*, TARANET requires support from the end host.



System	Year	Deploy- ment <sup>1</sup>	Tech- niques <sup>2</sup>	Volume anonymity	Timing anon.	Path anon.	Throughput <sup>3</sup>	Main overhead / bottleneck
TOR/Onion Routing [234, 239]	1999	C/S	P, (C, D), R	✗	~	✓	100 Mbps	Latency (send via relays)
NetCamo [135]	2001	C/S/N	P, C, R	~	~	✗	N/A <sup>4</sup>	Per-flow padding
Wang et al. [136]	2008	S	P, C, D	~	~	✓	10 Gbps	Per-flow padding, latency (via server)
BuFLO [242]	2012	C/S	P, C, D	~	✓	✗	320 Mbps	All packets have the same size
CS-BuFLO [134, 137]	2012	C/S	P, C, D	~	~	✗	400 Mbps	All packets have the same size
HORNET [104]	2015	C/N	P, R	~	✗	✓	8 Gbps	Onion routing and constant-size packets
WTF-PAD [138]	2016	C/S	P, C	~	~	✗	N/A <sup>4</sup>	Per-flow padding
PriFi [204]	2017	C/S	P, C, D	✓	✓	✓	100 Mbps	Throughput (1 client can send per slot)
Loopix [203]	2017	C/S	P, C, D, R	✓	~	✓	4 Mbps	Per-device obfuscation, computation
TARANET [106]	2018	C/S/N	P, C, D	✗	~	✓	4 Gbps	Per-flow(let) obfuscation
<b>ditto</b>	<b>2021</b>	<b>N</b>	<b>P, C, D</b>	<b>✓</b>	<b>✓</b>	<b>✓</b>	<b>100 Gbps</b>	Switch resources, pattern efficiency

<sup>1</sup> C: Client; S: Server, N: Network    <sup>2</sup> P: padding, C: chaff packets, D: delay, R: routing    <sup>3</sup> Results from the respective paper. When applicable, we use the following assumptions: throughput for 1 device port or link, 100 Gbps line rate, 1500 B packet or message size, 1000 users or devices, 1 server with a 10 Gbps connection, clients with 1 Gbps connections    <sup>4</sup> No throughput measurements in the paper. But the per-flow obfuscation makes the throughput is significantly worse compared to ditto.

TABLE 4.2: Comparison of ditto’s key properties with related work. Related work focuses on preventing traffic-analysis attacks on shared links, which adds other constraints compared to ditto and generally results in worse performance.

## 4.9 CONCLUSION

This chapter shows that it is possible to obfuscate volume- and timing properties of wide area network (WAN) traffic directly in the network data plane, using existing hardware, and with a small performance overhead.

ditto mixes real and chaff traffic and it adds padding to packets such that they follow a predefined pattern with respect to packet size and timing.

Two insights allow ditto to achieve high performance (up to 70 Gbps per 100 Gbps switch port for real Internet backbone traffic and interactive applications) and perfect security (observed traffic is independent of real traffic): *(i)* the traffic pattern is efficient because it fits the actual traffic distribution in the protected network; and *(ii)* existing network devices offer the features which are needed to perform packet padding and mixing with chaff traffic at line rate.

In the next chapter, we will change the perspective and discuss how programmable networks can also be used to de-obfuscate networks, their traffic and their users.



In the previous chapters, we presented two techniques that use programmable networks to obfuscate networks and their traffic. In this chapter, we change the perspective and discuss how programmable networks can also be used for the opposite: to de-obfuscate networks, their traffic, and their users.

In the following paragraphs, we first summarize two previous works on using programmable networks to (i) detect VoIP calls and reveal the caller and callee; and (ii) apply random forest models and classify network traffic at line rate. Then, we outline how programmable networks could improve the traffic-analysis attacks that ditto prevents.

Afterwards, we present a new case study that shows how programmable networks can detect proxy servers based on their traffic.

The main enabler for all these systems is that programmable switches can run custom programs to analyze every packet at line rate, without sampling or cloning or rerouting traffic to a different device.

***Identifying VoIP calls and their participants*** In previous work, we presented DELTA [4], a novel network-level side-channel attack that can efficiently identify VoIP calls in-path, in real-time, and at scale. DELTA leverages the VoIP signaling mechanism used for peer discovery and call setup: VoIP callers and callees first connect to a publicly known central application server shortly before communicating directly with each other. This fundamental call bootstrapping mechanism, used by prominent VoIP platforms (including Signal, Skype, Telegram, and WhatsApp), leads to a triangular connection pattern. DELTA takes advantage of this pattern by storing the signaling connections and linking the candidate call flows to these stored addresses. Our implementation of DELTA runs on Intel Tofino switches [243] and we evaluate our prototype against both real and synthesized packet traces. Our experiments show that a single switch's DBBF can hold connection state to simultaneously detect up to 100 000 unique VoIP calls for each VoIP application with virtually no hash collisions within up to 6.4 Tbps of traffic, the switch's maximum throughput. To put these numbers in

perspective, AMS-IX [244] – the busiest Internet location in the world – transits around 11 Tbps of traffic on its peak [245], whereas WhatsApp, the most popular VoIP service in the world, facilitates 100 million calls per day worldwide [246]. In other words, two switches would be enough to process worldwide VoIP traffic, provided it would cross these switches.

*Applying machine learning models to network traffic* In previous work, we presented pForest [3], a system that enables programmable switches to perform real-time inference according to supervised machine learning models (random forests), accurately and at scale. To save resources on the programmable switch and to speed up the classification, pForest classifies flows as soon as possible, after the first few packets. To do so, it trains a sequence of models that maps to the different phases of a flow. The data plane applies this sequence during the life of a flow until the candidate label is certain enough. Accordingly, pForest decides on a per-flow basis when the soonest classification is possible. Our evaluation shows that pForest can perform traffic classification at line rate, for hundreds of thousands of concurrent flows, and with a classification score that is on-par with software-based solutions.

*Accelerating traffic-analysis attacks* Most traffic-analysis attacks assume that the traffic can be analyzed on a server. However, this is unrealistic for high traffic volumes (as for example in the WANs considered in Chapter 4). Here, programmable switches could reduce the amount of traffic that needs to be sent to the server. We illustrate this with the traffic-analysis attacks that we used to motivate ditto. They require only packet sizes, timings, and directions as inputs [119–134]. Programmable switches can help scaling these attacks to high traffic volumes in two ways: (i) by filtering traffic that is relevant for the attack; and (ii) by computing the features for the attack directly in the data plane. For example, in the case of the deep fingerprinting attack [126] that we used to evaluate ditto, the feature vector consists of at most 5000 packet directions – a feature that is trivial to compute in a programmable switch. Even though the resources available in today’s programmable switches do not allow assembling such long feature vectors for a reasonable number of concurrent flows, they can assemble shorter vectors and leverage the fact that most flows are much shorter anyways [247].

## 5.1 CASE STUDY: PROXY SERVER DETECTION

In the remainder of this chapter, we present a case study to show how programmable switches can help detect proxy servers in an ISP network.

If Internet users do not want to reveal their identity (especially their public IP address) to a contacted server, they can use a proxy server. Proxy servers (or proxies) relay requests and responses between their clients and the contacted servers such that the proxy appears as the source of the requests and therefore the clients are not visible to the servers.

Many people and organizations use proxies to access geo-restricted content, to perform market research, or to monitor website rankings on search engines [248]. However, proxies also help criminals to conceal their identity. For example, they help to perform credit card fraud [249, 250], ad fraud [251] and scanning for vulnerable devices [252]. They are also useful for performing distributed denial-of-service attacks such as the one originating from the Mirai botnet with 65 000 residential hosts [253] or link-flooding attacks that require a many sources at specific locations [113, 114].

Proxies can run at various places. Traditionally, proxies were either operated by an organization as their exit point for all outgoing connections or they were operated by third-party providers in data centers. We focus on the latter case, where everybody can use the proxy (sometimes for free, sometimes for a fee). However, the fact that these proxies were running on dedicated machines in data centers (sometimes even on publicly listed IPs [254, 255]) made it possible for network operators or content providers to identify and block these IPs. Since detectable proxies are often of little use, this has led to a new type of proxy which is much harder to detect: residential IP proxies (RESIPs). These proxies are not running in data centers controlled by the service provider, but instead in residential networks (e.g., people's homes). As a result, these proxies allow their customers to hide behind actual residential IP addresses, which makes them much harder to detect using existing methods.

Leading proxy service providers (including Bright Data [256], Storm-Proxy [257], and GeoSurf [258]) offer their customers tens of millions of residential IP addresses that can be used as proxies. Their proxy servers usually run on home users' devices with the owner's consent and give the owner a small reward for every gigabyte that was sent through their device [259]. However, previous work has shown that many residential proxies run on likely compromised IoT devices without the owner's consent [251].

Proxy servers – especially residential ones – are not only troublesome for content providers, they can also be problematic for Internet service providers (ISPs) because (i) they can be used for attacks (e.g., DDoS) against the ISP and thereby downgrade its reputation; (ii) they can be used for attacks against other targets and thereby associate the ISP’s IP range with malicious actions; and (iii) they can consume the ISP’s customers’ bandwidth without consent and thereby degrade the perceived performance for the customer. Furthermore, since residential proxy servers often run without the owner’s consent [251, 260], the ISP could also perform the proxy detection as a service for its customers.

***Detecting proxies with programmable networks*** In this case study, we show that programmable switches can help a network operator (especially an ISP) to detect potential proxy servers at scale and without impacting the network performance for its customers.

We present a proof-of-concept implementation that analyzes network traffic, extracts features, and uses clustering to distinguish between normal clients and proxies. Even though the approach is simple, we show in the evaluation that it can distinguish between normal clients and proxies with an  $F_1$  score of more than 90%.

***Related work*** Early proxy-detection approaches used fixed rules or deep packet inspection to identify proxies [261–264]. However, since most traffic today is encrypted and proxies can run on various devices, newer approaches use more sophisticated techniques. For instance, they use packet-level features such as inter-arrival times, packet sizes, and flow durations [265–271]. However, these systems assume that the whole network traffic is available for their analysis, which is infeasible in large networks due to their high traffic volume.

In our case study, we demonstrate how programmable switches can allow proxy detection systems to scale by extracting some of the features in the data plane. While programmable switches could accelerate all of the systems mentioned above, we use this case study to present a system that is different from existing ones in the sense that it only requires one feature: timestamps of TLS connections.

***Outline*** In the following sections, we provide background information about how proxies work (Section 5.2); describe our network model and terminology (Section 5.3); present a system to detect proxies based on timing information in TLS connections (Section 5.4); describe how the

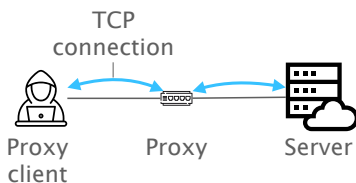


FIGURE 5.1: Basic proxy setup. A client connects directly to the proxy and the proxy maintains separate connections with the client and the server.

system extracts features in the data plane (Section 5.5) and identifies proxies in the control plane (Section 5.6); evaluate the system (Section 5.7); and discuss the system’s limitations and ways to resolve them (Section 5.8).

## 5.2 BACKGROUND ON PROXY SERVERS

At a high level, proxies act as intermediaries between their clients and the servers that these clients want to connect to. As illustrated in Figure 5.1, instead of connecting directly to the server, a client then sends its request to the proxy and the proxy sends the request to the server through a separate connection [272].

There are various types of proxies. The ones we consider in this case study are so-called forward proxies, which relay traffic between many sources (e.g., subscribers of a service) and all destinations (e.g., any website). The other type of proxies is reverse proxies, which accept requests from all sources but only to specific destinations (e.g., one website). These proxies are often used for load balancing and redundancy.

Forward proxies exist in many different sub-categories. For example, *open proxies* are listed publicly and are available to every Internet user [254, 255]. Other proxy servers are only accessible to subscribers of commercial services or to clients within a certain network (e.g., a company network).

Depending on the setup, proxies differ in how they receive their clients’ requests. The most prevalent types are HTTP(S) proxies and SOCKS<sub>5</sub> proxies. HTTP(S) proxies receive the requests from the client through HTTP(S) and therefore only work for HTTP(S) traffic. On the other hand, SOCKS<sub>5</sub> proxies use the SOCKS protocol that can work with any application-layer protocol [273].



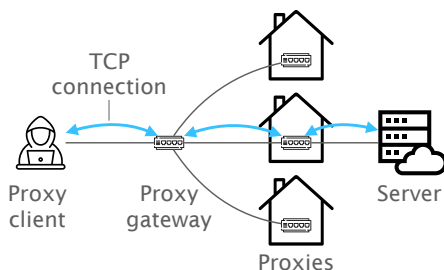


FIGURE 5.2: Proxy setup with a gateway. The client connects to a gateway, which relays its traffic to a proxy.

Generally, a client establishes a direct connection with the proxy (Figure 5.1). However, many commercial service providers operate a gateway server to which the client connects. This gateway server then forwards the request to the actual proxy server (Figure 5.2) [251]. This setting makes management and monitoring easier for the proxy providers and allows them to distribute requests from one client over many proxies (to offer so-called rotating IP proxies).

### 5.3 MODEL

In this section, we describe the networks in which we want to detect proxies and we define the terminology used in the remainder of this case study.

**Network model** Our goal is to identify IP addresses that act as proxy servers in an Internet Service Provider (ISP) network. These networks connect their customers (private households or businesses) to the Internet. We assume that the network forwards traffic such that each flow traverses at least one programmable switch that runs our proxy detection system.

The ISP assigns one public IP address to each customer, but the customers can operate many devices that share this IP address through network address translation (NAT) [274]. Each of the customer’s devices can act as a normal client and/or as a proxy. We do not differentiate between individual devices sharing one IP. We say that an IP acts as a proxy if at least one device that uses this IP acts as a proxy.

Figure 5.3 illustrates an ISP network and the following paragraph explains the terminology in more detail.

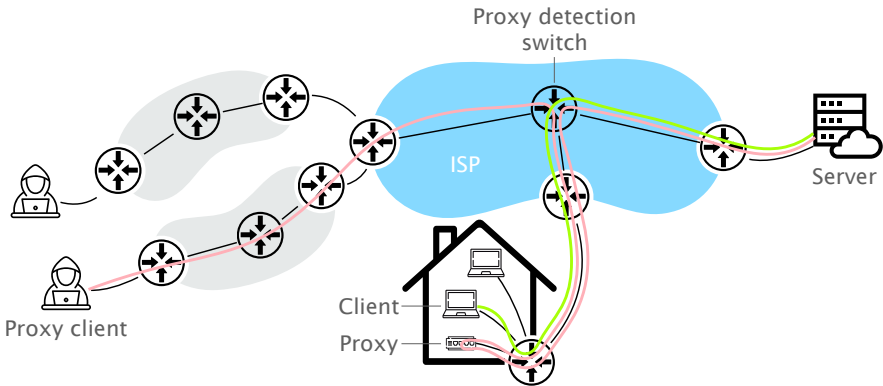


FIGURE 5.3: Network model. We consider an ISP network where customers can have multiple devices acting as normal clients and/or as proxies.

**Terminology** Below, we explain some essential terms used in this case study. Parts of our terminology are borrowed from [275] and [276].

- *Server*: device that hosts websites
- *Proxy*: device that acts as a proxy
- *Client*: device that is used to browse the Internet. We distinguish between two types of clients:
  - *Normal client*: a client that does not use a proxy
  - *Proxy client*: a client that uses a proxy
- *Stub*: local network of one customer of an ISP, identified by its public IP assigned by the ISP
- *Proxy provider*: an entity that provides access to proxies (but does not necessarily operate them).
- *Website request*: action taken by a user that triggers one or more HTTP(S) requests (e.g., a click on a link or entering a URL in the browser’s address bar)
- *Head request*: the first HTTP(S) request triggered by a user request
- *Embedded requests*: Other HTTP(S) requests triggered by a user request
- *Isolated requests*: HTTP(S) requests that are neither head requests nor embedded requests

## 5.4 DESIGN OVERVIEW

In this section, we explain how we distinguish between normal clients and proxy servers based on their traffic.

We first state our design goals. Then, we describe how the traffic of proxies and normal clients differs and how website requests allow estimating the distance between the client and the server. Afterwards, we explain the metric we use to measure these differences and our underlying hypothesis. Finally, we outline the architecture for the implemented system.

**Design goals** The goal of this case study is to develop a system that can distinguish between normal clients and proxies in an ISP network while fulfilling these requirements:

- The system’s input is the traffic exchanged in the ISP network
- The system cannot decrypt encrypted traffic
- The system has no control over the end devices (clients, proxies, servers)
- The system must not degrade the network performance

**Peculiarities of proxies** As explained above, proxies relay connections from their clients to the connected server. This leads to several peculiarities in the network traffic from and to proxies. For example: *(i)* there is a strong correlation between incoming and outgoing traffic regarding timing and volume; *(ii)* compared to typical ISP customers, proxies receive more incoming connections; and *(iii)* interactions that involve the client happen slower because traffic first needs to reach the client.

We focus our system on the third property because it is more difficult to obfuscate for the proxy provider than the other two. Obfuscating the first two properties is possible for the proxy provider or the client by introducing chaff packets. Especially if the proxy client connects to the proxy through a gateway server controlled by the proxy provider (illustrated in Figure 5.2) [251]. Such a gateway server allows the proxy provider to add chaff packets between the gateway and the proxy and to “reverse” the connection between the proxy and the relay server (i.e., the proxy initiates the connection to the relay server).

On the other hand, the time it takes until the proxy client has received the content depends on its geographical location and the distance to the

proxy. While it is not possible to measure this time directly (after all, the main purpose of a proxy is to hide the client), we explain below how we approximate it using the characteristics of HTTP and HTML.

***Peculiarities of web browsing*** Almost all websites contain multiple elements (e.g., text, images, videos, and client-side code) which a browser needs to download before displaying the website. These elements are stored in separate files and potentially on separate servers. When a user requests a website, the browser first downloads the HTML code that specifies the structure of the website (e.g., the text and the position of images). Based on the HTML code, the browser then knows which additional elements it needs to download.

We leverage this behavior to estimate the time it takes for the content to reach the client which is potentially hiding behind a proxy. More precisely, we leverage two key observations.

The first observation is that loading a website almost always results in the following pattern:

- ① The client requests the website's HTML code
- ② The server returns the code
- ③ The client parses the code
- ④ The client requests additional elements (potentially from multiple servers)
- ⑤ The server(s) return additional elements

The second observation is that the time between ② and ④ depends on the geographical distance between the client and the server, even if the client uses a proxy. The time between ② and ④ is approximately proportional to the distance client—proxy—server if the client uses a proxy or client—server otherwise.

In Figures 5.4 and 5.5, we illustrate this pattern for the case where the client loads a website without a proxy (Figure 5.4) and where it loads it through a proxy (Figure 5.5). Because most web-browsing traffic today is encrypted using TLS [53], we design our system to work even if traffic is encrypted. Figures 5.4 and 5.5 show the TLS handshakes between the proxy and the server.

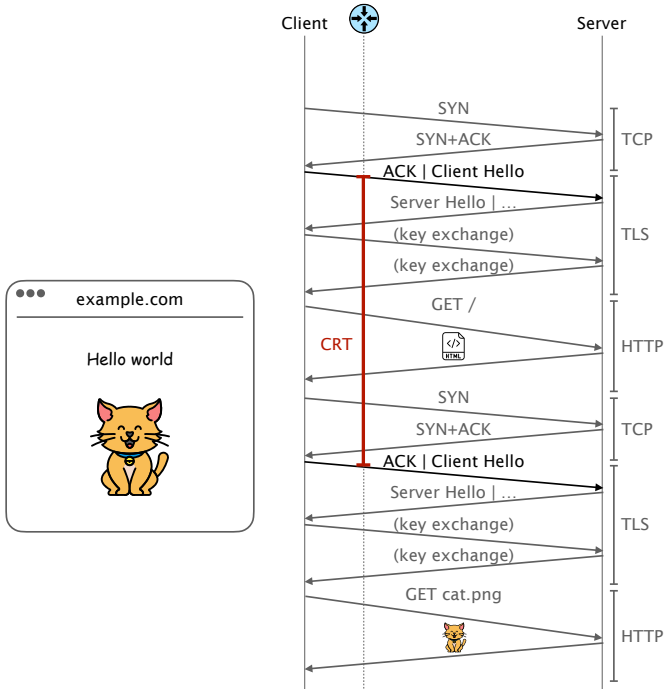


FIGURE 5.4: Loading a website without a proxy. The client establishes a first TLS connection to load the HTML code. After it receives the HTML code, it establishes a second connection to request the embedded image.

While TLS hides the packet contents, it still allows identifying the requests in steps 1 and 4 because both requests trigger the establishment of a new TLS connection with the server and the time between these two TLS connections also depends on the geographical distance between the client and the server.<sup>1</sup>

**Client reaction time (CRT)** We call the time that it takes a client to react to new instructions for loading elements the client reaction time (CRT). As an example, consider the case where the HTML code and the embedded elements are hosted on the same server. Then, the CRT computes to the time difference between when the server returns the HTML code and when the server receives the request for the first embedded element. However, computing this time solely based on TLS-encrypted traffic is not possible,

<sup>1</sup> In Section 5.8, we discuss the impact of future HTTP versions.

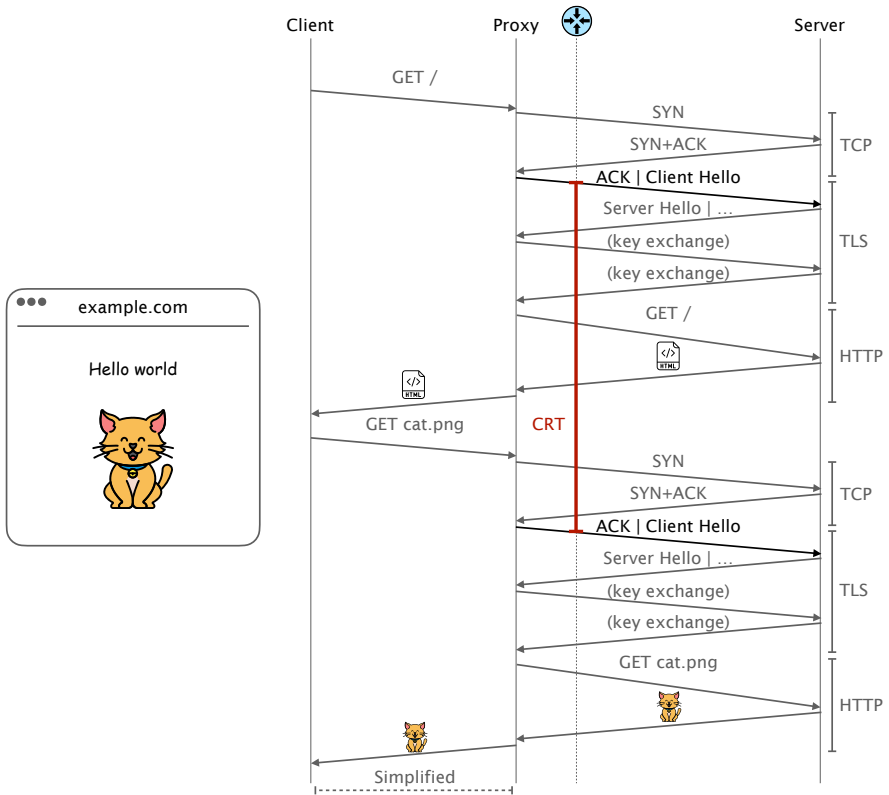


FIGURE 5.5: Loading a website through a proxy. The time between the response with the HTML code and the request of the image is now longer because the HTML code first needs to reach the client behind the proxy. Note that the channel between the client and the proxy is simplified here (e.g., we do not show the handshakes to establish a TLS connection)

which is why we approximate it. In our case, we compute the CRT as the time between the two TLS handshakes (more precisely: the client hello packets in the two handshakes) shown in Figures 5.4 and 5.5. This approximation works well for our purposes because it still captures the difference between requests that go through a proxy and those that do not.

**Hypothesis** Given that the CRT depends on the geographical distance between the client and the server, it also depends on the distance between

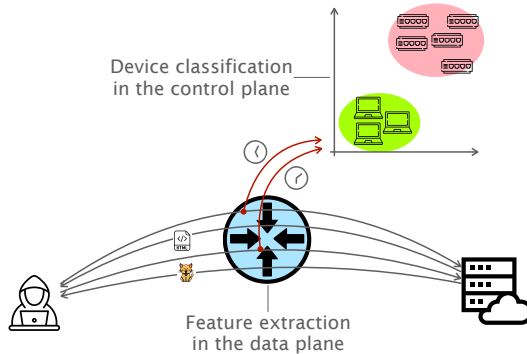


FIGURE 5.6: Architecture of our proxy detection system. A programmable switch extracts features from network traffic, and the control plane computes clusters of similar devices and distinguishes between proxies and normal clients.

the proxy and the client if a proxy is involved. We therefore expect that it reveals whether an IP address is used as a proxy or not.

More precisely, given the IP address of a stub in the ISP network, we expect the following characteristics of the CRT depending on whether this IP address is used as a proxy:

If the IP address is *not used as a proxy*, we expect the CRT to be (i) relatively small compared to other IPs in the same network (because the client is identical or close to the device with the IP address); and (ii) relatively constant (because all devices behind this IP are close by).

On the other hand, if the IP address is *used as a proxy*, we expect the CRT to be (i) relatively large compared to other IPs in the same network (because the clients behind the proxy are further away); and (ii) relatively variable (because the clients are in many different locations).

**Architecture** Figure 5.6 shows the architecture of our proxy detection system. The system consists of two main building blocks: A *data plane component* to extract the metrics that we use to identify proxies from traffic in real-time; and a *control plane component* to analyze the extracted metrics and perform the classification.

In the following two sections, we explain each component in more details.

## 5.5 EXTRACTING FEATURES IN THE DATA PLANE

In [277], we developed a prototype implementation of the data-plane component that runs on Intel Tofino switches. Below, we summarize how this P4 program identifies TLS client hello packets in the data plane and notifies the control plane.

**Identifying TLS client hello packets** The parser identifies TLS client hello packets based on the following criteria: (i) it is a TCP packet (i.e., the etherType in the Ethernet header is equal to 0x0800 and the protocol in the IP header is equal to 0x6); (ii) the SYN flag is 0 (i.e., it is not a TCP handshake packet that does not contain TLS records); (iii) the destination port is 443 (i.e., the default for HTTPS); (iv) the content type of the TLS record is 0x16 (i.e., it is a TLS handshake packet); and (v) the handshake type 0x1 (i.e., it is a client hello packet).

**Notifying the control plane** The switch notifies the control plane about any new client hello packet through a so-called digest message. Digest messages provide an efficient way of sending data from the data plane to the control plane without cloning the entire packet [278]. In our case, the digest message contains the packet's source and destination IP address as well as the timestamp at which the packet was received.

This implementation serves only as a prototype and is not the most efficient one. As we show in [277], the switch could already compute the CRT in the data plane – at least for some requests, depending on how much memory is available. In Section 5.8, we discuss additional possibilities for improvements.

## 5.6 IDENTIFYING PROXIES IN THE CONTROL PLANE

In this section, we explain how we use the timestamp measurements from the data plane to distinguish between normal clients and proxy clients.

**Inputs** The inputs to the control plane algorithm are timestamp measurements from the data plane as described above.

**Features** We hypothesize that the CRT of proxy clients is higher and follows a broader distribution than the CRT of normal clients. Therefore, our two main features are the mean value and the standard deviation of



the CRTs of one stub. But there are cases when we cannot compute the CRT, namely when the head request and the embedded requests are distributed over multiple proxies (i.e., they have different source IPs). This is the case for so-called rotating IP proxies. To cover this, we add an additional feature that computes the percentage of isolated requests per IP. Isolated requests are those that are not preceded or followed by another request with the same (source IP, destination IP) pair.

In summary, our features are:

- $CRT_m$ : The mean value of the CRT
- $CRT_{std}$ : The standard deviation of the CRT
- $Isol_{\%}$ : The percentage of isolated requests

**Feature extraction** To compute these features, the proxy detection system must first extract website requests from the inputs. To do this, it analyzes the individual inputs (i.e., client hello packets) and labels each of them as one of three request types:

- *Head requests* denote the first request that loads the HTML code of a website
- *Embedded requests* denote the requests that follow a head request and load additional elements (images etc.) contained in the website
- *Isolated requests* denote requests that are neither head nor embedded requests

For the labeling, we use a simple heuristic based on two timeouts ( $T_{idle}$  and  $T_{head}$ ):  $T_{idle}$  is the minimum idle time between two head requests. In other words, we assume that a user waits for  $T_{idle}$  until opening the next website.  $T_{head}$  is the maximum time it takes until all elements of a website are loaded after the head request.

Based on these two parameters and the information in the inputs (timestamps, source IP addresses and destination IP addresses), we then label each request in two stages.

In the first stage, we assign the labels as follows:

- potential head if there is no other request from the same client in the last  $T_{idle}$
- embedded if it is within  $T_{head}$  after a request labeled as potential head or head and it has the same source and destination IP as the

(potential) head. We then change the label of the potential head to head

- potential embedded if it is within  $T_{head}$  after a request labeled as potential head or head and it has the same source but a different destination IP

In the second stage, we change the label of all requests previously labeled as potential head to isolated if they were not followed by at least one embedded or potential embedded.

Once all requests are labeled, computing the features is straightforward. The CRT is the time between a head request and its first embedded request.  $CRT_m$  is the mean CRT of one source IP and  $CRT_{std}$  is the standard deviation.  $Isol\%$  is the ratio of isolated requests over the number of total requests for each source IP.

We acknowledge that this heuristic is not optimal, but building an optimal heuristic is out of scope for this case study. For example, the heuristic gets disturbed when a client starts multiple requests simultaneously or when websites load additional elements over a long time. In Section 5.8, we discuss the limitations in detail.

**Classification** After we extracted the features ( $CRT_m$ ,  $CRT_{std}$ , and  $Isol\%$ ) for each source IP, we put them into two clusters using the k-means clustering algorithm [279]. According to our hypothesis, clients and proxies show different behaviors with respect to our features and therefore end up in different clusters. Since we expect the proxies to have higher values for  $CRT_m$  and  $CRT_{std}$ , we interpret the cluster with the higher center (w.r.t.  $CRT_m$  and  $CRT_{std}$ ) as the proxy cluster and label all IPs in this cluster as proxies and all other IPs as clients.

## 5.7 EVALUATION

In this section, we show that our proof-of-concept implementation can identify proxies with high accuracy ( $F_1$  score over 90%) even if the proxy clients are in close geographical proximity, they use a large range of different bandwidths, and the proxy uses rotating IPs.

After we describe our evaluation methodology, we show how the detection accuracy of our system depends on the number of observed requests,

the clients' bandwidths, the clients' locations, and the number of IP addresses per proxy.

### 5.7.1 *Methodology*

We now describe how we generate and collect the data we use to evaluate the proxy detection system.

**ISP topology** We consider a hypothetical ISP that operates in New York City and wants to detect proxy servers among its customers.

**Infrastructure** We perform all experiments on the cloud infrastructure provided by DigitalOcean [280]. DigitalOcean allows creating virtual machines (VMs) in eight different regions (Amsterdam, Bangalore, Frankfurt, London, New York City, San Francisco, Singapore, Toronto). Depending on the experiment, we host our clients in a varying selection of these regions.

**Clients** Our clients are DigitalOcean VMs with 2 virtual CPU cores and 4 GB memory. The clients run Ubuntu 20.04.4 LTS and use Firefox to browse the web. To automate the experiments, we use Selenium [281]. The clients use HTTP/1.1 and TLS 1.3 to access our webserver.

**Proxies** We use proxies provided by BrightData [256] (previously called Luminati), one of the largest providers of proxy servers [251]. The Bright Initiative [282] granted us free access to BrightData's proxy services. BrightData offers different types of proxies: residential proxies, ISP proxies, datacenter proxies and mobile proxies. We use ISP proxies for our experiments because they fit our use case (detecting proxies in ISP networks) and they do not involve devices which potentially act as a proxy without the owner's consent (as it can be the case for residential proxies [251], cf. ethics discussion below).

**Server** We run an Apache webserver [283] version 2.4.41 in one VM with 4 virtual CPU cores and 8 GB memory in Frankfurt. Apache runs in its default configuration. To obtain TLS certificates, we use LetsEncrypt [284]. The website consists of three elements besides the HTML code: an image, a CSS stylesheet in a separate file, and javascript code in a separate file.

**Data collection** To create the datasets, we capture the traffic on the webserver's network interface using tcpdump [225] and we create logs with the

		<i>True class</i>	
		Proxy	Normal client
<i>Predicted class</i>	Proxy	True positive (tp)	False positive (fp)
	Normal client	False negative (fn)	True negative (tn)

TABLE 5.1: Confusion matrix

timestamps of each website request on all clients. We only use these logs as ground truth, the proxy detection system receives only the pcap file as input.

**Metrics** We measure the accuracy of our proxy detection system using the typical metrics precision, recall, and the  $F_1$  score. These metrics are based on the confusion matrix in Table 5.1.

The *precision* denotes the ratio of correctly identified proxies over all IPs labeled as proxies:

$$\text{precision} = \frac{\text{tp}}{\text{tp} + \text{fp}}$$

The *recall* denotes the ratio of correctly identified proxies over all proxies in the dataset:

$$\text{recall} = \frac{\text{tp}}{\text{tp} + \text{fn}}$$

The  $F_1$  score is the harmonic mean of the precision and recall:

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{tp}}{\text{tp} + \frac{1}{2}(\text{fp} + \text{fn})}$$

**Ethics** Our experiments do not raise ethical issues because:

- We only capture traffic that we created ourselves (through browser automation).
- We do not use residential proxies because they might be running without the device owner’s consent.
- We described our project to the Bright Initiative and they approved us for using their proxy infrastructure.

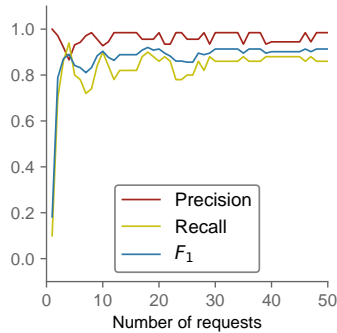


FIGURE 5.7: Accuracy depending on the number of requests. Proxies can be detected reliably after only about 30 requests in our datasets.

### 5.7.2 Detection time

In this experiment, we evaluate how the accuracy depends on the number of observed requests. We run 12 normal clients in New York City and 48 proxy clients at random locations. These proxy clients use one of 10 proxies in New York City at random.

In Figure 5.7, we show the precision, recall and  $F_1$  score depending on the number of analyzed requests. Each point in the plot shows the average over 5 runs of the experiment. The results show that the detection works well ( $F_1$  score  $\geq 90\%$ ) after 30 observed requests. Ideally, this would mean that our system can identify proxies after only 30 user requests. However, as we discuss in Section 5.8, our datasets are not fully representative for real ISP traffic, which might have a positive impact on our results.

### 5.7.3 Impact of the bandwidth

In this experiment, we evaluate the impact of the clients' bandwidth on the detection accuracy. To do this, we limit the bandwidth of each client randomly to one of the following values: 1 Gbps, 500 Mbps, 300 Mbps, 100 Mbps, 50 Mbps, 25 Mbps, 10 Mbps, 5 Mbps, 1 Mbps, 0.5 Mbps. The six highest values correspond to the rates offered by AT&T, one of the world's largest ISPs [285]. We added the smaller values to simulate devices that cannot fully utilize the high bandwidth.

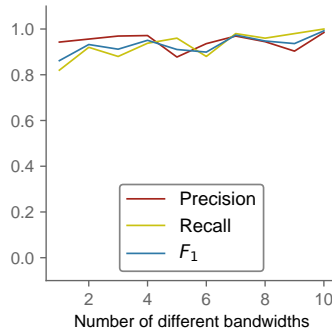


FIGURE 5.8: Accuracy depending on the bandwidth distribution. The bandwidth has a small impact on the accuracy because the website is small.

In Figure 5.8, we show how the  $F_1$  score changes depending on the number of different bandwidths of the clients. The value on the x-axis specifies that the clients used the top- $x$  bandwidths randomly (e.g., for  $x = 3$ , each client was configured with a bandwidth of either 1 Gbps, 500 Mbps, or 300 Mbps). To limit the bandwidth, we used `iproute-tc` [286]. In each case, there were 12 normal clients in New York City, 48 proxy clients at random locations, and 10 proxy servers in New York City. Each point in the plot shows the average over 5 runs of the experiment.

The results show that the bandwidth distribution has a small impact on the accuracy of our system. The explanation for this is that the website that we load is small (about 400 kB in total, and the HTML code is only 1 kB), and therefore the impact of the bandwidth on the CRT is small. Since we compute the CRT between the first two client hello packets, only the HTML code and handshake packets are transmitted during this time. Since these packets are small, the bandwidth does not have a significant impact on the CRT (e.g., the time difference between transmitting 1 kB at 1 Gbps and at 25 Mbps is only 312  $\mu$ s while  $CRT_m$  is around 0.8 s in our topology).

#### 5.7.4 Impact of the geographical distribution

In this experiment, we evaluate the impact of the client's location on the  $F_1$  score. To do this, we randomly distribute the 48 proxy clients across an increasing number of locations. The available locations are the ones offered by DigitalOcean as listed above.

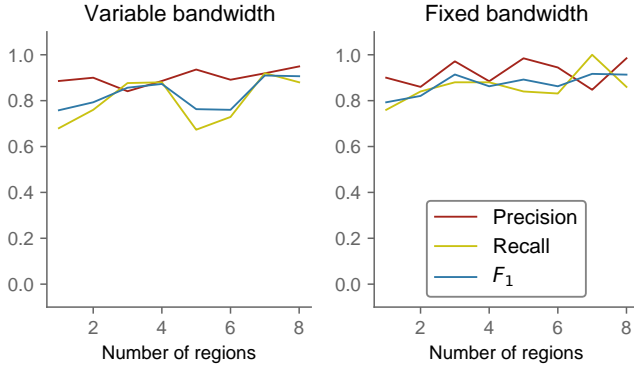


FIGURE 5.9: Accuracy depending on the number of regions where proxy clients are placed. The accuracy increases if the proxy clients are distributed in multiple regions.

In Figure 5.9, we show how the  $F_1$  score changes depending on the number of different locations of the clients. The value on the x-axis specifies that the top- $x$  locations sorted by their distance to New York City are used (e.g., for  $x = 3$ , we distributed the clients across New York City, Toronto and San Francisco).

We use 12 normal clients in New York City and 48 proxy clients at random locations. We evaluate the case where the bandwidth of each client is the same (1 Gbps) and the case where each client has a bandwidth randomly selected among 1 Gbps, 500 Mbps, 300 Mbps, 100 Mbps, 50 Mbps, 25 Mbps (the bandwidths offered by AT&T [285]). Again, each point in the plot shows the average over 5 runs of the experiment.

The results show that the accuracy generally increases slightly with the number of regions. This is expected because if the proxy clients are in other regions than the normal clients,  $CRT_m$  increases. And if the proxy clients are distributed over many regions,  $CRT_{std}$  increases too. The results vary more if the bandwidth is randomly chosen because of our small sample size (only 5 runs of the experiment).

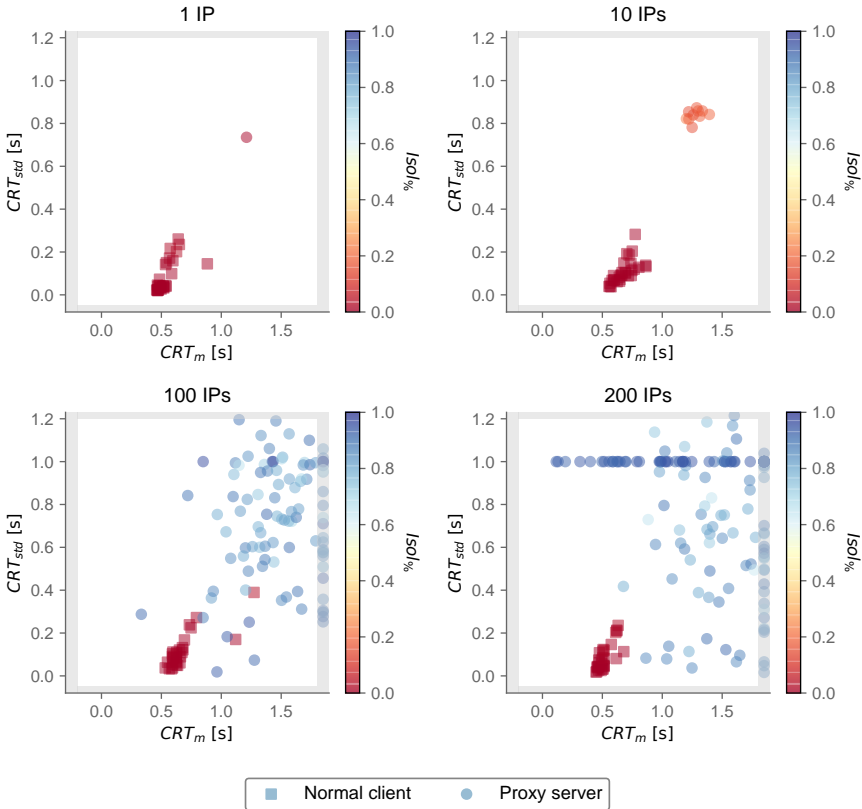


FIGURE 5.10: Examples of feature values for proxies with rotating IPs. Datapoints outside of the shown range are mapped to the grey border area.

### 5.7.5 Impact of rotating IP addresses

In this experiment, we evaluate the impact of rotating IP addresses on the F1 score. For this experiment, we use 30 normal clients in New York City and 30 proxy clients with a randomly chosen bandwidth at randomly chosen locations. All proxy clients use the same proxy server (in New York City), but depending on the scenario, the proxy server distributes its requests over 1, 10, 100 or 200 IP addresses.<sup>2</sup>

<sup>2</sup> Higher numbers of IPs were not available.



	IPs per proxy			
	1	10	100	200
Precision	1.0	1.0	0.97	0.45
Recall	0.93	1.0	0.98	0.92
F1	0.96	1.0	0.97	0.53

TABLE 5.2: Accuracy for proxies with rotating IPs. Our system can detect proxies even if they use many source IPs.

In Figure 5.10, we illustrate the results of one measurement in each of these scenarios. The plots now show 1, 10, 100 or 200 proxy IP addresses and a varying degree of isolated requests. In this experiment, the feature  $Isol_{\%}$  becomes important because the CRT cannot be computed if the head and embedded requests come from different source IPs. The example with 200 proxy IPs shows many IPs with  $CRT_{std}$  equal to 1. This is because we initialize  $CRT_{std}$  to 1 if there are less than two CRT measurements for this IP. This often happens here because the requests are spread over many IPs.

In Table 5.2, we show the accuracy depending on the number of IP addresses per proxy. The results show that our system can detect proxies with up to 100 rotating IP addresses with the same accuracy as it detects proxies with a fixed IP. However, as the number of IPs per proxy increases to 200, the clustering does not work well anymore. While the recall is still high (i.e., the IPs classified as proxies are indeed proxies), the precision is low (i.e., many of the IPs classified as normal clients are in fact proxies). Presumably, a more sophisticated and better tuned clustering algorithm would achieve better results as our basic approach based on k-means and three equally weighted features.

## 5.8 DISCUSSION AND FUTURE WORK

The purpose of this case study is to sketch how programmable switches can extract data from network traffic with proxy server detection as an example. While the developed system shows promising results, it is far from perfect and provides many opportunities to improve it.

Next, we discuss the current limitations and outline approaches to fix them. In addition, we discuss how proxy providers could avoid detection.

### 5.8.1 Limitations

Below, we discuss the main limitations of our proxy detection system and describe solutions to improve them.

**Simple datasets** The datasets from our evaluation are not representative of traffic that an actual ISP would see for the following reasons:

- (i) the datasets only contain HTTPS traffic towards one webserver and one website
- (ii) the loaded website loads all elements immediately
- (iii) all clients and proxy clients have the same operating system and browser
- (iv) all clients and proxy clients run as VMs in datacenters of a cloud provider
- (v) each stub consists of only one device
- (vi) each client and proxy client performs all requests of the website sequentially
- (vii) each stub consists of only one device

Item (i) and Item (ii) could be achieved in a real dataset relatively easily by filtering only traffic of websites that are suitable for the analysis. Of course, this would slow down the detection because the system would need to wait until it has captured enough requests for these websites.

Item (iii), Item (iv), Item (v), and Item (vii) could be addressed through more extensive simulations. Instead of trying to simulate a realistic environment, one could collect the dataset with the help of volunteers who access websites for which one can record the traffic. This would also address Item (vi).

Item (vii) could be overcome by performing the experiments via multiple proxy providers.

**Overlapping requests** Our heuristic to compute the CRT is based on the assumption that a stub accesses only one website at a time. In practice, this assumption does not hold in many cases, especially if there are multiple active clients in one stub.

The simple solution here would be to discard these requests and wait until only one client is active simultaneously. However, this would add

delay to the decision. A more sophisticated approach would be to use other features for identifying head and embedded requests. For example, the server name indication (SNI) extension in TLS allows to extract the (unencrypted) domain name of the requested website [287]. Furthermore, existing works (e.g., [120, 288]) have also shown that the TLS handshake reveals enough information to fingerprint the client (e.g., with respect to its operating system and browser).

***Different versions of HTTP*** Over the years, HTTP has undergone several enhancements that affect how embedded elements are loaded:

- HTTP/1.0 establishes a separate TCP connection for every element [54]
- HTTP/1.1 allows reusing a TCP connection for loading multiple elements [289]
- HTTP/2 uses a single TCP connection for all elements [290]
- HTTP/3 uses QUIC and UDP instead of TCP connections [291]

We designed our proxy detection system for HTTP/1.0 and HTTP/1.1 (if HTTP/1.1 does not reuse the connection between the head request and the embedded requests). These versions of HTTP are still predominant today, but HTTP/2 and HTTP/3 are gaining more and more support (currently, about 45 % of the 10 M most popular websites support HTTP/2 [292] and 25 % support HTTP/3 [293]). Extracting the CRT from HTTP/2 and HTTP/3 would require a different approach, but it is still possible. Since all requests to the same server would be transmitted over the same connection, one could identify the head and embedded requests as the first two “bursts” of response data in this connection.

***Many packets sent to the controller*** In our current implementation, we send a digest message to the controller for every client hello packet. This could overload the controller in large networks. To reduce the load on the controller, one could improve the system in the following ways:

- Compute the CRT in the data plane: Using registers, it is possible to compute the CRT entirely in the data plane (we have developed a proof-of-concept in [277]). However, the relatively little available memory (tens of megabytes [294]) limits the number of parallel website requests, for which the timestamps can be stored in the switch.
- Adaptive sampling: As shown in the evaluation, a few requests are enough to identify proxies with high certainty. Therefore, constantly analyzing all traffic is not needed. Instead, one could only analyze a

small percentage of requests if they come from IPs that are already classified. This sampling could be performed directly in the data plane.

### 5.8.2 Countermeasures

In this section, we present some countermeasures that proxy providers or content providers could deploy in order to prevent detection by our system.

**Chaff traffic** To make the CRT computation more difficult, proxies could initiate additional requests simultaneously to the requests that they initiate for their clients. This would have a similar effect as simultaneous requests discussed above: It would disturb our current heuristic and lower the detection accuracy.

If the proxy provider uses a gateway server between the clients and the proxies, it could also inject chaff packets into the connection between the gateway and the proxy. However, since our analysis is based on the traffic between the proxy and the server, this would not have an impact on our system.

**Caching** Proxies can cache the contents they requested for their clients such that they can respond to future requests with the data that they already have in cache. In this case, our system could not extract a CRT because the proxy answers the embedded request without contacting the server. This would slow down the detection with our system because it takes more time to observe enough (non-cached) requests.

**Delaying requests** A proxy could tamper with the CRT by introducing additional (potentially random) delays before it relays a request from a client to the server. By doing this, the proxy could increase  $CRT_m$  and increase or decrease  $CRT_{std}$ . However, the higher  $CRT_m$  would likely make it easier to detect the proxy.

**Encoding elements in HTML** Instead of embedding the paths to elements in HTML, it is possible to encode the elements (e.g., images) directly in the HTML code using base64 encoding [295]. If a website does this for all elements, it does not trigger embedded requests. However, this would decrease the website performance and require cooperation from the website provider.

## 5.9 CONCLUSION

In this chapter, we have shown that programmable switches can help to de-obfuscate network users and their traffic.

We first summarized how programmable switches can *(i)* identify VoIP calls and determine the caller and callee at scale by recognizing the unique traffic pattern that occurs upon the establishment of a VoIP call; *(ii)* apply random forest models to network traffic at line rate by optimizing the models and features for the resources and operations available in programmable switches; and *(iii)* extract the features that are required for many traffic-analysis attacks at line rate and without sampling, thereby making these attacks more scalable.

Then, we presented a case-study that shows how programmable switches can help to scale the identification of proxy servers to ISP networks. Here, the main insight was that timing properties of web requests that come from proxies are different from requests that come from the client directly. We have shown that by measuring the so-called client reaction time it is possible to distinguish between regular clients and proxy servers.

In summary, we have shown that programmable networks are not only useful for implementing network-level defenses, but they can also perform powerful traffic analysis at line rate. This results in a new generation of possible attack vectors not only for de-obfuscation.

## CONCLUSION AND OUTLOOK

---

In this chapter, we conclude this dissertation and we describe future research problems in the area of network obfuscation.

### 6.1 CONCLUSION

In this dissertation, we presented two systems that increase network security through obfuscation. Both systems leverage recent advances in network programmability, which allow network switches to run sophisticated algorithms at line rate.

In Chapter 3, we demonstrated how programmable switches allow obfuscating topologies in order to prevent link-flooding attacks. We presented `NetHide`, a new, usable approach for obfuscating network topologies. The core idea behind `NetHide` is to phrase the obfuscation task as a multi-objective optimization problem. The security requirements are encoded as hard constraints and the usability ones as soft constraints using the notions of accuracy and utility. `NetHide` relies on an ILP solver and effective heuristics to compute obfuscated topologies and on programmable network devices to capture and modify path tracing traffic at line rate. Our evaluation on realistic topologies and simulated attacks shows that `NetHide` can obfuscate large topologies with marginal impact on usability.

In Chapter 4, we demonstrated how programmable switches allow obfuscating volume- and timing properties of wide area network (WAN) traffic in order to prevent traffic-analysis attacks. We presented `ditto`, a system that mixes real and chaff traffic and adds padding to packets such that they follow a predefined pattern with respect to packet size and timing. Two insights allow `ditto` to achieve high performance (up to 70 Gbps per 100 Gbps switch port for real Internet backbone traffic and interactive applications) and perfect security (observed traffic is independent of real traffic): (i) the traffic pattern is efficient because it fits the actual overall traffic distribution in the protected network; and (ii) programmable network devices offer the

features which are needed to perform packet padding and mixing with chaff traffic at line rate.

After presenting two systems that use programmable switches for obfuscation, we changed the perspective for the next chapter and discussed how programmable switches can also achieve the opposite – for benign and malicious purposes.

In Chapter 5, we demonstrated how programmable switches allow de-obfuscating network users and their traffic. We explained how programmable switches can be used to *(i)* identify VoIP calls and determine the caller and callee at scale; *(ii)* apply random forest models to network traffic at line rate; *(iii)* scale traffic-analysis attacks to high volumes; and *(iv)* detect devices that are acting as proxy servers in an ISP network.

## 6.2 OPEN RESEARCH PROBLEMS

In this section, we first describe opportunities for improving NetHide and ditto. Then, we sketch a more general obfuscation framework that could be developed in future work.

### 6.2.1 *Better topology obfuscation*

We see two main directions along which future work could improve NetHide.

***Handling changes in the topology*** If the physical topology changes permanently, NetHide needs to compute a new virtual topology. This virtual topology will be similar to the new physical topology, but not necessarily similar to the previous virtual topology. To change that, we suggest including the difference between the previous virtual topology and the new virtual topology in the accuracy metric. Then, the optimization problem would produce a new virtual topology similar to both the previous virtual topology and the new physical topology.

***Preventing router fingerprinting and timing side-channels*** NetHide takes two measures to prevent router fingerprinting based on timing information: First, it ensures that the physical path and the virtual path have similar lengths (to avoid information leakage through the propagation time). And second, it ensures that the response to a path tracing packet comes from the correct router according to the virtual topology (to avoid information

leakage through the processing time). However, NetHide measures the length of a path in terms of the number of hops. Since the physical length of two paths (and therefore their propagation time) can be very different even if two paths have the same number of hops, router fingerprinting could still be possible. To counteract this, NetHide could delay path tracing responses to obfuscate the true RTT (e.g., by recirculating a response packet several times to delay it) or it could take the physical link lengths into consideration when generating the virtual topology.

In addition, we see a research opportunity in combining topology obfuscation through modified path tracing responses (as done by NetHide) with topology obfuscation through modified end-to-end delays (as done by [167] and [171]). A combined system could prevent both types of topology inference while maintaining the utility of path tracing tools and minimizing the additional delay.

### 6.2.2 *Better traffic obfuscation*

We see two main directions along which future work could improve ditto.

***Reacting to changes in the traffic distribution*** ditto keeps using a pattern until the operator instructs it to compute a new one. In the future, ditto could automatically compute and deploy a new pattern when the traffic distributions changes significantly. To achieve this, ditto could measure the padding it adds to packets, the reordering it causes, and the recirculations it requires. At the same time, it could continuously measure the traffic distribution (e.g., by using similar techniques as presented in [296, 297]). When the current pattern requires too much padding, reordering, or recirculations, ditto could immediately compute and deploy a new pattern.

On the other hand, adapting the pattern too often can reveal too much about the current real traffic. Part of this future work should also be to define the trade-off between overhead and information leakage and to find the optimal parameter choice.

***Distributing padding over multiple switches*** Recirculations that are required because a switch cannot add enough padding in one pipeline pass constitute a significant part of ditto's overhead. Instead of recirculating the packets multiple times through the same switch, one could distribute the padding over multiple switches, especially since most traffic likely traverses multiple switches because it enters the WAN.



These “internal” switches could add padding to each packet until it has reached the size of one of the pattern states. Ideally, the edge switch then only needs to mix real and chaff packets. To do this, the internal switches need to be programmable and they need to know the pattern states (the edge switch could distribute this information).

The same applies to the de-obfuscation side, where the removal of padding could also happen over multiple switches.

### 6.2.3 *Towards a general obfuscation framework*

NetHide, ditto, and most related works are isolated systems that obfuscate one property of a network (e.g., the topology or the traffic) using one platform (e.g., programmable switches). For future work, we see an opportunity to combine many obfuscation techniques into one framework. This framework could then leverage multiple platforms (e.g., hosts, network interfaces and switches) to obfuscate various properties of networks (following the concept of “deep programmability” discussed in [298]).

Developing such a framework would involve research to answer the following questions:

- *Which properties can and should be obfuscated?* We have shown how to obfuscate the topology and traffic. However, these are not the only properties of a network that can be obfuscated. Other properties include, for example, the number and type of hosts or the configuration (such as firewall rules or load balancing).
- *How can we specify an obfuscation policy?* Depending on the network, there will be different requirements regarding what needs to be obfuscated. To specify these requirements, a policy language should be developed. The policies could be based on concepts such as  $k$ -anonymity [299] (e.g., to specify that the traffic of one host is indistinguishable of at least  $k$  other hosts).
- *How can we perform the obfuscation?* Given the properties that need to be obfuscated and the policies that need to hold, there should be a compiler that translates them into programs and configurations for hosts, network interfaces, switches, and other middleboxes such as firewalls.

## OWN PUBLICATIONS

---

- [1] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. “NetHide: Secure and Practical Network Topology Obfuscation”. In: *Proceedings of the USENIX Security Symposium*. 2018.
- [2] Roland Meier, Vincent Lenders, and Laurent Vanbever. “ditto: WAN Traffic Obfuscation at Line Rate”. In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2022.
- [3] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. *pForest: In-Network Inference with Random Forests*. <https://arxiv.org/abs/1909.05680>. arXiv preprint. 2019.
- [4] Ege Cem Kirci, Maria Apostolaki, Roland Meier, Ankit Singla, and Laurent Vanbever. “Mass Surveillance of VoIP Calls through Programmable Data Planes”. In: *Proceedings of the ACM Symposium on SDN Research (SOSR)*. 2022.
- [5] Roland Meier, David Gugelmann, and Laurent Vanbever. “iTAP: In-network Traffic Analysis Prevention using Software-Defined Networks”. In: *Proceedings of the ACM Symposium on SDN Research (SOSR)*. 2017.
- [6] Roland Meier, Cornelia Scherrer, David Gugelmann, Vincent Lenders, and Laurent Vanbever. “FeedRank: A Tamper-resistant Method for the Ranking of Cyber Threat Intelligence Feeds”. In: *Proceedings of the International Conference on Cyber Conflict (CyCon)*. 2018.
- [7] Nicolas Käzig, Roland Meier, Luca Gambazzi, Vincent Lenders, and Laurent Vanbever. “Machine Learning-based Detection of C&C Channels with a Focus on the Locked Shields Cyber Defense Exercise”. In: *Proceedings of the International Conference on Cyber Conflict (CyCon)*. 2019.
- [8] Pierre Dumont, Roland Meier, David Gugelmann, and Vincent Lenders. “Detection of Malicious Remote Shell Sessions”. In: *Proceedings of the International Conference on Cyber Conflict (CyCon)*. 2019.

- [9] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. “(Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2019.
- [10] Roland Meier, Arturs Lavrenovs, Kimmo Heinäaro, Luca Gambazzi, and Vincent Lenders. “Towards an AI-powered Player in Cyber Defence Exercises”. In: *Proceedings of the International Conference on Cyber Conflict (CyCon)*. 2021.
- [11] Lina Gehri, Roland Meier, Daniel Hulliger, and Vincent Lenders. *Generalizing Machine Learning Models to Detect Command and Control Attack Traffic*. Under submission. 2022.

## REFERENCES

---

- [12] The New York Times. *Strava Fitness App Can Reveal Military Sites, Analysts Say* (01/29/2018). <https://www.nytimes.com/2018/01/29/world/middleeast/strava-heat-map.html>. (Accessed on 07/25/2022).
- [13] TIME. *And Bomb The Anchovies* (08/13/1990). <http://content.time.com/time/subscriber/article/0,33009,970860,00.html>. (Accessed on 06/10/2021).
- [14] Washington Post. *With Capital in Panic, Pizza Deliveries Soar* (12/19/1998). <https://www.washingtonpost.com/wp-srv/politics/special/clinton/stories/pizza121998.htm>. (Accessed on 07/25/2022).
- [15] Lawrence G Roberts. "The evolution of packet switching". In: *Proceedings of the IEEE* 66.11 (1978).
- [16] Michael Hauben. *Behind the Net: The Untold History of the ARPANET and Computer Science*. <http://www.columbia.edu/~rh120/ch106.x07>. (Accessed on 06/11/2022). 1995.
- [17] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. "A Brief History of the Internet". In: *ACM SIGCOMM Computer Communication Review (CCR)* 39.5 (2009).
- [18] Juniper Research. *IoT Connected Devices to Triple to Over 38Bn Units* (07/28/2020). <https://www.juniperresearch.com/press/iot-connected-devices-to-triple-to-38-bn-by-2020>. (Accessed on 06/11/2022).
- [19] Cisco. *Cisco Annual Internet Report (2018-2023)*. <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>. (Accessed on 06/11/2022). 2020.
- [20] The Washington Post. *The real story of how the Internet became so vulnerable* (05/30/2015). <https://www.washingtonpost.com/sf/business/2015/05/30/net-of-insecurity-part-1/>. (Accessed on 07/27/2022).
- [21] Kenneth Ingham, Stephanie Forrest, et al. *A history and survey of network firewalls*. Tech. rep. University of New Mexico, 2002.

- [22] Google Transparency Report. *HTTPS encryption on the web*. <https://transparencyreport.google.com/https/overview>. (Accessed on 07/27/2022).
- [23] Bingdong Li, Esra Erdin, Mehmet Hadi Gunes, George Bebis, and Todd Shipley. "An overview of anonymity technology usage". In: *Computer Communications* 36.12 (2013).
- [24] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*. <https://twiki.cern.ch/twiki/pub/HEPIX/TechwatchNetwork/HtwNetworkDocuments/white-paper-c11-741490.pdf>. 2019.
- [25] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review (CCR)* 38.2 (2008).
- [26] PCWorld. *This startup may have built the world's fastest networking switch chip (06/14/2016)*. <https://www.pcworld.com/article/415231/this-startup-may-have-built-the-worlds-fastest-networking-switch-chip.html>. (Accessed on 06/23/2022).
- [27] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. "Network-wide heavy hitter detection with commodity switches". In: *Proceedings of the ACM Symposium on SDN Research (SOSR)*. 2018.
- [28] Diogo Barradas, Nuno Santos, Luis Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. "FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications." In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2021.
- [29] Qun Huang, Patrick PC Lee, and Yungang Bao. "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference". In: *Proceedings of the ACM SIGCOMM Conference*. 2018.
- [30] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics". In: *Proceedings of the ACM SIGCOMM Conference*. 2017.
- [31] Jiao Zhang, Shubo Wen, Jinsheng Zhang, Hua Chai, Tian Pan, Tao Huang, Linqun Zhang, Yunjie Liu, and F Richard Yu. "Fast switch-based load balancer considering application server states". In: *IEEE/ACM Transactions on Networking* 28.3 (2020).

- [32] Shie-Yuan Wang, Jun-Yi Li, and Yi-Bing Lin. "Aggregating and disaggregating packets with various sizes of payload in P4 switches at 100 Gbps line rate". In: *Journal of Network and Computer Applications* 165 (2020).
- [33] Jiamin Cao, Jun Bi, Yu Zhou, and Cheng Zhang. "Cofilter: A high-performance switch-assisted stateful packet filter". In: *Proceedings of the ACM SIGCOMM Conference (Posters)*. 2018.
- [34] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. "Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense". In: *Proceedings of the ACM SIGCOMM Conference*. 2022.
- [35] Guanyu Li, Menghao Zhang, Chang Liu, Xiao Kong, Ang Chen, Guofei Gu, and Haixin Duan. "Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering". In: *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*. 2019.
- [36] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends". In: *IEEE Access* 9 (2021).
- [37] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks (5th Edition)*. Morgan Kaufmann, 2011.
- [38] *CIDR Report*. <https://www.cidr-report.org/as2.0/>. (Accessed on 06/02/2022).
- [39] Cisco Press. *The Switched Environment*. <https://www.ciscopress.com/articles/article.asp?p=2181835&seqNum=5>. (Accessed on 07/26/2022).
- [40] *IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks*. IEEE Std 802.1Q-1998. 1999.
- [41] John Moy. *OSPF Version 2*. RFC 2328. <http://www.rfc-editor.org/rfc/rfc2328.txt>. 1998.
- [42] Yakov Rekhter and Tony Li. *A Border Gateway Protocol 4 (BGP-4)*. RFC 1654. <http://www.rfc-editor.org/rfc/rfc1654.txt>. 1994.
- [43] *IEEE Standard for Ethernet*. IEEE Std 802.3-2018. 2018.
- [44] Google. *IPv6 Statistics*. <https://www.google.com/intl/en/ipv6/statistics.html>. (Accessed on 06/23/2022).

- [45] IPv4 Market Group. *A Brief History of IPv4*. <https://ipv4marketgroup.com/a-brief-history-of-ipv4/>. (Accessed on 07/28/2022).
- [46] RIPE Network Coordination Centre. *What is IPv4 Run Out?* <https://www.ripe.net/manage-ips-and-asns/ipv4/ipv4-run-out>. (Accessed on 07/28/2022).
- [47] IANA. *ICMP Parameters*. <https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml>. (Accessed on 06/05/2022).
- [48] *ping(8) - Linux man page*. <https://linux.die.net/man/8/ping>. (Accessed on 08/08/2022).
- [49] *traceroute(8) - Linux manual page*. <http://man7.org/linux/man-pages/man8/traceroute.8.html>. (Accessed on 08/09/2022).
- [50] J. Postel. *User Datagram Protocol*. RFC 768. <http://www.rfc-editor.org/rfc/rfc768.txt>. 1980.
- [51] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*. RFC 6335. <http://www.rfc-editor.org/rfc/rfc6335.txt>. 2011.
- [52] Jon Postel. *Transmission Control Protocol*. RFC 793. <http://www.rfc-editor.org/rfc/rfc793.txt>. 1981.
- [53] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. <http://www.rfc-editor.org/rfc/rfc8446.txt>. 2018.
- [54] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. <http://www.rfc-editor.org/rfc/rfc1945.txt>. 1996.
- [55] W3Techs. *Usage Statistics of Default protocol https for Websites, July 2022*. <https://w3techs.com/technologies/details/ce-httpsdefault>. (Accessed on 07/26/2022).
- [56] Paul Göransson and Chuck Black. *Software Defined Networks, A Comprehensive Approach*. Morgan Kaufmann, 2014.
- [57] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review (CCR)* 44.3 (2014).
- [58] *P4-16 Portable Switch Architecture (PSA)*. <https://p4.org/p4-spec/docs/PSA-v1.1.0.html>. (Accessed on 08/09/2022).

- [59] *Intel Tofino Series Programmable Ethernet Switch ASIC*. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html>. (Accessed on 08/09/2022).
- [60] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. "Programmable Calendar Queues for High-speed Packet Scheduling". In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2020.
- [61] Péter Vörös and Attila Kiss. "Security middleware programming using P4". In: *Proceedings of the International Conference on Human Aspects of Information Security, Privacy, and Trust*. 2016.
- [62] Rakesh Datta, Sean Choi, Anurag Chowdhary, and Younghee Park. "P4guard: Designing p4 based firewall". In: *Proceedings of the IEEE Military Communications Conference (MILCOM)*. 2018.
- [63] Ali AlSabeH, Elie Kfoury, Jorge Crichigno, and Elias Bou-Harb. "P4DDPI: Securing P4-Programmable Data Plane Networks via DNS Deep Packet Inspection". In: *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*. 2022.
- [64] Amar Almaini, Ahmed Al-Dubai, Imed Romdhani, and Martin Schramm. "Delegation of authentication to the data plane in software-defined networks". In: *Proceedings of the IEEE International Conference on Smart Computing, Networking and Services (SmartCNS)*. 2019.
- [65] Eder Ollora Zaballa, David Franco, Zifan Zhou, and Michael S Berger. "P4Knocking: Offloading host-based firewall functionalities to the network". In: *Proceedings of the IEEE Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 2020.
- [66] Amar Almaini, Ahmed Al-Dubai, Imed Romdhani, Martin Schramm, and Ayoub Alsarhan. "Lightweight edge authentication for software defined networks". In: *Computing* 103.2 (2021).
- [67] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. "Programmable In-Network Security for Context-aware BYOD Policies". In: *Proceedings of the USENIX Security Symposium*. 2020.



- [68] Harsh Gondaliya, Ganesh C Sankaran, and Krishna M Sivalingam. "Comparative evaluation of IP address anti-spoofing mechanisms using a P4/NetFPGA-based switch". In: *Proceedings of the P4 Workshop in Europe*. 2020.
- [69] Peng Kuang, Ying Liu, and Lin He. "P4DAD: securing duplicate address detection using P4". In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2020.
- [70] Aldo Febro, Hannan Xiao, and Joseph Spring. "Telephony Denial of Service defense at data plane (TDoSD@ DP)". In: *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 2018.
- [71] Goksel Simsek, Hakan Bostan, Alper Kaan Sarica, Egemen Sarikaya, Alperen Keles, Pelin Angin, Hande Alemdar, and Ertan Onur. "Dropppp: a P4 approach to mitigating dos attacks in SDN". In: *Proceedings of the International Workshop on Information Security Applications*. 2019.
- [72] Damu Ding, Marco Savi, Federico Pederzoli, Mauro Campanella, and Domenico Siracusa. "In-Network Volumetric DDoS Victim Identification Using Programmable Commodity Switches". In: *IEEE Transactions on Network and Service Management* 18.2 (2021).
- [73] Francesco Musumeci, Valentina Ionata, Francesco Paolucci, Filippo Cugini, and Massimo Tornatore. "Machine-learning-assisted DDoS attack detection with P4 language". In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2020.
- [74] Jiarong Xing, Wenqing Wu, and Ang Chen. "Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries". In: *Proceedings of the USENIX Security Symposium*. 2021.
- [75] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. "Poseidon: Mitigating volumetric ddos attacks with programmable switches". In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2020.
- [76] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. "Jaquen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches". In: *Proceedings of the USENIX Security Symposium*. 2021.

- [77] Jiarong Xing, Wenqing Wu, and Ang Chen. “Architecting programmable data plane defenses into the network with FastFlex”. In: *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*. 2019.
- [78] Garegin Grigoryan and Yaoqing Liu. “Lamp: Prompt layer 7 attack mitigation with programmable data planes”. In: *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA)*. 2018.
- [79] Aldo Febro, Hannan Xiao, and Joseph Spring. “Distributed SIP DDoS defense with P4”. In: *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*. 2019.
- [80] Yu Mi and An Wang. “ML-pushback: Machine learning based pushback defense against DDoS”. In: *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2019.
- [81] Libardo Andrey Quintero González, Lucas Castanheira, Jonatas Adilson Marques, Alberto Schaeffer-Filho, and Luciano Paschoal Gaspar. “BUNGEE: An Adaptive Pushback Mechanism for DDoS Detection and Mitigation in P4 Data Planes”. In: *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2021.
- [82] Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspar. “Offloading real-time DDoS attack detection to programmable data planes”. In: *Proceedings of the IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2019.
- [83] Xin Zhe Khooi, Levente Csikor, Dinil Mon Divakaran, and Min Suk Kang. “DIDA: Distributed in-network defense architecture against amplified reflection DDoS attacks”. In: *Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*. 2020.
- [84] Marinos Dimolianis, Adam Pavlidis, and Vasilis Maglaris. “A multi-feature ddos detection schema on p4 network hardware”. In: *Proceedings of the IEEE Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 2020.
- [85] Kurt Friday, Elie Kfoury, Elias Bou-Harb, and Jorge Crichigno. “Towards a unified in-network DDoS detection and mitigation strategy”. In: *Proceedings of the IEEE Conference on Network Softwarization (NetSoft)*. 2020.

- [86] Ya Gao and Zhenling Wang. “A Review of P4 Programmable Data Planes for Network Security”. In: *Mobile Information Systems 2021* (2021).
- [87] TechTarget. *What is obfuscation and how does it work?* <https://www.techtarget.com/searchsecurity/definition/obfuscation>. (Accessed on 07/20/2022).
- [88] Tor Project. *Pluggable Transports*. <https://2019.www.torproject.org/docs/pluggable-transports>. (Accessed on 07/20/2022).
- [89] Philipp Winter, Tobias Pulls, and Juergen Fuss. “ScrambleSuit: A polymorphic network protocol to circumvent censorship”. In: *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*. 2013.
- [90] Brandon Wiley. *Dust: A blocking-resistant internet transport protocol*. Tech. rep. 2011.
- [91] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briese-meister, Steven Cheung, Frank Wang, and Dan Boneh. “Stegotorus: a camouflage proxy for the tor anonymity system”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2012.
- [92] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. “Skypemorph: Protocol obfuscation for tor bridges”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2012.
- [93] Amir Houmansadr, Thomas J Riedl, Nikita Borisov, and Andrew C Singer. “I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention.” In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2013.
- [94] Shuai Li, Mike Schliep, and Nick Hopper. “Facet: Streaming over videoconferencing for censorship circumvention”. In: *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*. 2014.
- [95] Jeroen Massar, Ian Mason, Linda Briese-meister, and Vinod Yegneswaran. “Jumpbox—a seamless browser proxy for tor pluggable transports”. In: *Proceedings of the International Conference on Security and Privacy in Communication Networks*. 2014.
- [96] Amir Houmansadr, Wenxuan Zhou, Matthew Caesar, and Nikita Borisov. “Sweet: Serving the web by exploiting email tunnels”. In: *IEEE/ACM Transactions on Networking* 25.3 (2017).

- [97] Qiyan Wang, Xun Gong, Giang TK Nguyen, Amir Houmansadr, and Nikita Borisov. "Censorspoof: asymmetric communication using ip spoofing for censorship-resistant web browsing". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2012.
- [98] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. "Cloud-transport: Using cloud storage for censorship-resistant networking". In: *Proceedings of the International Symposium on Privacy Enhancing Technologies (PoPETs)*. 2014.
- [99] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. "Protocol misidentification made easy with format-transforming encryption". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013.
- [100] Kevin P Dyer, Scott E Coull, and Thomas Shrimpton. "Marionette: A programmable network traffic obfuscation system". In: *Proceedings of the USENIX Security Symposium*. 2015.
- [101] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. Naval Research Lab Washington DC, 2004.
- [102] Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Adrian Perrig, Akira Yamada, Samuel C Nelson, Marco Gruteser, and Wei Meng. "LAP: Lightweight anonymity and privacy". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2012.
- [103] Jody Sankey and Matthew Wright. "Dovetail: Stronger anonymity in next-generation internet routing". In: *Proceedings of the International Symposium on Privacy Enhancing Technologies (PoPETs)*. 2014.
- [104] Chen Chen, Daniele E Asoni, David Barrera, George Danezis, and Adrain Perrig. "HORNET: High-speed onion routing at the network layer". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.
- [105] Chen Chen and Adrian Perrig. "Phi: Path-hidden lightweight anonymity protocol at network layer". In: *Proceedings on Privacy Enhancing Technologies* 2017.1 (2017).
- [106] Chen Chen, Daniele E Asoni, Adrian Perrig, David Barrera, George Danezis, and Carmela Troncoso. "TARANET: Traffic-Analysis Resistant Anonymity at the Network Layer". In: *Proceedings of the IEEE European Symposium on Security and Privacy (Euro S&P)* (2018).

- [107] Taeho Lee, Christos Pappas, David Barrera, Pawel Szalachowski, and Adrian Perrig. "Source accountability with domain-brokered privacy". In: *Proceedings of the ACM Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 2016.
- [108] Taeho Lee, Christos Pappas, Pawel Szalachowski, and Adrian Perrig. "Communication based on per-packet One-Time Addresses". In: *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*. 2016.
- [109] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. "spine: Surveillance protection in the network elements". In: *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*. 2019.
- [110] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. "Programmable In-Network Obfuscation of Traffic". In: *arXiv preprint arXiv:2006.00097* (2020).
- [111] Yashodhar Govil, Liang Wang, and Jennifer Rexford. "MIMIQ: Masking IPs with Migration in QUIC". In: *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*. 2020.
- [112] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. <http://www.rfc-editor.org/rfc/rfc9000.txt>. 2021.
- [113] Ahren Studer and Adrian Perrig. "The core melt attack". In: *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. 2009.
- [114] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. "The crossfire attack". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2013.
- [115] Jinwoo Kim and Seungwon Shin. "Software-Defined HoneyNet: Towards Mitigating Link Flooding Attacks". In: *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2017).
- [116] Qian Wang, Feng Xiao, Man Zhou, Zhibo Wang, Qi Li, and Zhetao Li. "Linkbait: Active Link Obfuscation to Thwart Link-flooding attacks". In: *arXiv preprint arXiv:1703.09521* (2017).

- [117] Samuel T Trassare, Robert Beverly, and David Alderson. "A technique for network topology deception". In: *Proceedings of the IEEE Military Communications Conference (MILCOM)*. 2013.
- [118] Jinwoo Kim, Eduard Marin, Mauro Conti, and Seungwon Shin. "EqualNet: A Secure and Practical Defense for Long-term Network Topology Obfuscation". In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2022.
- [119] B. Coskun and N. Memon. "Tracking encrypted VoIP calls via robust hashing of network flows". In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. 2010.
- [120] J. Muehlstein, Y. Zion, M. Bahumi, I. Kirshenboim, R. Dubin, A. Dvir, and O. Pele. "Analyzing HTTPS encrypted traffic to identify user's operating system, browser and application". In: *Proceedings of the IEEE Consumer Communications Networking Conference (CCNC)*. 2017.
- [121] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Ulugac. "Peek-a-Boo: I See Your Smart Home Activities, Even Encrypted!" In: *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. 2020.
- [122] Brendan Saltaformaggio, Hongjun Choi, Kristen Johnson, Yonghui Kwon, Qi Zhang, Xiangyu Zhang, Dongyan Xu, and John Qian. "Eavesdropping on Fine-Grained User Activities Within Smartphone Apps Over Encrypted Network Traffic". In: *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*. 2016.
- [123] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. "Website fingerprinting in onion routing based anonymization networks". In: *Proceedings of the ACM workshop on Privacy in the electronic society*. 2011.
- [124] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. "Robust smartphone app identification via encrypted network traffic analysis". In: *IEEE Transactions on Information Forensics and Security* 13.1 (2018).
- [125] Mikhail Andreev, Avi Klausner, Trishita Tiwari, Ari Trachtenberg, and Arkady Yerukhimovich. "Nothing But Net: Invading Android User Privacy Using Only Network Access Patterns". In: *arXiv preprint arXiv:1807.02719* (2018).

- [126] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. “Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018.
- [127] Michael Backes, Goran Doychev, Markus Dürmuth, and Boris Köpf. “Speaker recognition in encrypted voice streams”. In: *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. 2010.
- [128] R. Dubin, A. Dvir, O. Pele, and O. Hadar. “I Know What You Saw Last Minute—Encrypted HTTP Adaptive Video Streaming Title Classification”. In: *IEEE Transactions on Information Forensics and Security* 12.12 (2017).
- [129] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. “Beauty and the Burst: Remote Identification of Encrypted Video Streams”. In: *Proceedings of the USENIX Security Symposium*. 2017.
- [130] Y. Shi and S. Biswas. “Website fingerprinting using traffic analysis of dynamic webpages”. In: *Proceedings of the IEEE Global Communications Conference*. 2014.
- [131] Se Eun Oh, Shuai Li, and Nicholas Hopper. “Fingerprinting keywords in search queries over tor”. In: *Proceedings on Privacy Enhancing Technologies* 2017.4 (2017).
- [132] Fan Zhang, Wenbo He, Xue Liu, and Patrick G Bridges. “Inferring users’ online activities through traffic analysis”. In: *Proceedings of the ACM conference on Wireless network security*. 2011.
- [133] Jamie Hayes and George Danezis. “K-Fingerprinting: A Robust Scalable Website Fingerprinting Technique”. In: *Proceedings of the USENIX Security Symposium*. 2016.
- [134] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. “Touching from a distance: Website fingerprinting attacks and defenses”. In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2012.
- [135] Yong Guan, Xinwen Fu, Dong Xuan, P.U. Shenoy, R. Bettati, and Wei Zhao. “NetCamo: camouflaging network traffic for QoS-guaranteed mission critical applications”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 31.4 (2001).

- [136] Wei Wang, Mehul Motani, and Vikram Srinivasan. "Dependent link padding algorithms for low latency anonymity systems". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2008.
- [137] Xiang Cai, Rishab Nithyanand, and Rob Johnson. "CS-BuFlo: A congestion sensitive website fingerprinting defense". In: *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*. 2014.
- [138] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. "Toward an Efficient Website Fingerprinting Defense". In: *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. 2016.
- [139] Data Center Knowledge. *How to Fight the New Breed of DDoS Attacks on Data Centers*. <http://www.datacenterknowledge.com/security/how-fight-new-breed-ddos-attacks-data-centers>. (Accessed on 08/09/2022).
- [140] Akamai. *Q2 2017 State of the Internet*. <https://www.akamai.com/newsroom/press-release/akamai-releases-second-quarter-2017-state-of-the-internet-security-report>. (Accessed on 08/09/2022).
- [141] Dyn. *Dyn Statement on 10/21/2016 DDoS Attack*. <https://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/>. (Accessed on 08/09/2022).
- [142] WIRED. *Github survived the biggest DDoS attack ever recorded (03/01/2018)*. <https://www.wired.com/story/github-ddos-memcached/>. (Accessed on 08/09/2022).
- [143] Vasileios Giotsas, Georgios Smaragdakis, Christoph Dietzel, Philipp Richter, Anja Feldmann, and Arthur Berger. "Inferring BGP Black-holing Activity in the Internet". In: *Proceedings of the ACM Internet Measurement Conference (IMC)*. 2017.
- [144] Cloudflare. *Unmetered Mitigation: DDoS Protection Without Limits*. <https://blog.cloudflare.com/unmetered-mitigation/>. (Accessed on 08/09/2022).
- [145] Soo Bum Lee, Min Suk Kang, and Virgil D. Gligor. "CoDef: Collaborative Defense Against Large-scale Link-flooding Attacks". In: *Proceedings of the ACM Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 2013.



- [146] Ars Technica. *Can a DDoS break the Internet? Sure... just not all of it (04/02/2013)*. <https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/>. (Accessed on 08/09/2022).
- [147] ProtonMail. *Message Regarding the ProtonMail DDoS Attacks (11/10/2015)*. <https://protonmail.com/blog/protonmail-ddos-attacks/>. (Accessed on 08/09/2022).
- [148] TechRepublic. *Exclusive: Inside the ProtonMail siege: how two small companies fought off one of Europe's largest DDoS attacks (11/13/2015)*. <http://www.techrepublic.com/article/exclusive-inside-the-protonmail-siege-how-two-small-companies-fought-off-one-of-europes-largest-ddos/>. (Accessed on 08/09/2022).
- [149] Neil Spring, Ratul Mahajan, and David Wetherall. "Measuring ISP topologies with Rocketfuel". In: *ACM SIGCOMM Computer Communication Review (CCR)* 32.4 (2002).
- [150] RIPE Atlas. <https://atlas.ripe.net/>.
- [151] Min Suk Kang, Virgil D Gligor, and Vyas Sekar. "SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks". In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2015.
- [152] Christos Liaskos, Vasileios Kotronis, and Xenofontas Dimitropoulos. "A novel framework for modeling and mitigating distributed link flooding attacks". In: *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 2016.
- [153] Thomas Holterbach, Cristel Pelsser, Randy Bush, and Laurent Vanbever. "Quantifying Interference between Measurements on the RIPE Atlas Platform". In: *Proceedings of the ACM Internet Measurement Conference (IMC)*. 2015.
- [154] Ethan Katz-Bassett, John P John, Arvind Krishnamurthy, David Wetherall, Thomas E Anderson, and Yatin Chawathe. "Towards IP geolocation using delay and topology measurements." In: *Proceedings of the ACM Internet Measurement Conference (IMC)*. 2006.
- [155] C. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. <http://www.rfc-editor.org/rfc/rfc2992.txt>. 2000.
- [156] *The P4 Language Specification - Version 1.0.4*. <https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf>. (Accessed on 08/09/2022).

- [157] Lei Xue, Xiapu Luo, Edmond WW Chan, and Xian Zhan. "Towards Detecting Target Link Flooding Attack". In: *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*. 2014.
- [158] Fabien A. P. Petitcolas. "Kerckhoffs' Principle". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Springer US, 2011.
- [159] *Gurobi Mathematical Programming Solver*. <http://www.gurobi.com/products/gurobi-optimizer>.
- [160] VI Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions and Reversals". In: *Soviet Physics Doklady* 10 (1966).
- [161] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959).
- [162] Daniel Schoepe and Andrei Sabelfeld. "Understanding and Enforcing Opacity". In: *Proceedings of the IEEE Computer Security Foundations Symposium*. 2015.
- [163] *The Internet Topology Zoo*. <http://topology-zoo.org/>.
- [164] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. "Semi-Oblivious Traffic Engineering: The Road Not Taken". In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018.
- [165] J. M. Smith and M. Schuchard. "Routing Around Congestion: Defeating DDoS Attacks and Adverse Network Conditions via Reactive BGP Routing". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2018.
- [166] Muoi Tran, Min Suk Kang, Hsu-Chun Hsiao, Wei-Hsuan Chiang, Shu-Po Tung, and Yu-Su Wang. "On the feasibility of rerouting-based DDoS defenses". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2019.
- [167] Tao Hou, Zhe Qu, Tao Wang, Zhuo Lu, and Yao Liu. "ProTO: Proactive topology obfuscation against adversarial network topology inference". In: *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 2020.
- [168] Tao Hou, Tao Wang, Zhuo Lu, and Yao Liu. "Combating adversarial network topology inference by proactive topology obfuscation". In: *IEEE/ACM Transactions on Networking* 29.6 (2021).

- [169] Yaqun Liu, Jinlong Zhao, Guomin Zhang, and Changyou Xing. "NetObfu: A lightweight and efficient network topology obfuscation defense scheme". In: *Computers & Security* 110 (2021).
- [170] Jinwoo Kim, Jaehyun Nam, Suyeol Lee, Vinod Yegneswaran, Phillip Porras, and Seungwon Shin. "BottleNet: Hiding Network Bottlenecks Using SDN-Based Topology Deception". In: *IEEE Transactions on Information Forensics and Security* 16 (2021).
- [171] Yaqun Liu, Changyou Xing, Guomin Zhang, Lihua Song, and Hongxiu Lin. "AntiTomo: Network topology obfuscation against adversarial tomography-based topology inference". In: *Computers & Security* 113 (2022), 102570.
- [172] Fida Gillani, Ehab Al-Shaer, Samantha Lo, Qi Duan, Mostafa Ammar, and Ellen Zegura. "Agile virtualized infrastructure to proactively defend against cyber attacks". In: *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 2015.
- [173] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. "Lightweight Source Authentication and Path Validation". In: *Proceedings of the ACM SIGCOMM Conference*. 2014.
- [174] Cristina Basescu, Raphael M Reischuk, Pawel Szalachowski, Adrian Perrig, Yao Zhang, Hsu-Chun Hsiao, Ayumu Kubota, and Jumpei Urakawa. "SIBRA - Scalable Internet Bandwidth Reservation Architecture". In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2016.
- [175] Giacomo Giuliani, Dominik Roos, Marc Wyss, Juan Angel García-Pardo, Markus Legner, and Adrian Perrig. "Colibri: a cooperative lightweight inter-domain bandwidth-reservation infrastructure". In: *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. 2021.
- [176] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. "B4: Experience with a globally-deployed software defined WAN". In: *ACM SIGCOMM Computer Communication Review (CCR)* 43.4 (2013).
- [177] Amazon. *Global Infrastructure*. <https://aws.amazon.com/about-aws/global-infrastructure>. (Accessed on 06/15/2021).

- [178] Microsoft Azure. *Backbone Networking Infrastructure*. <https://azure.microsoft.com/en-us/global-infrastructure/global-network>. (Accessed on 06/15/2021).
- [179] Schweizer Armee. *IKT-Systeme der Armee*. <https://www.vtg.admin.ch/de/aktuell/themen/programme-projekte/ikt-systeme-der-armee.html>. (Accessed on 06/10/2021).
- [180] SWAN *Scottish Wide Area Network*. <https://www.scottishwan.com>.
- [181] North Dakota ITD. *Wide Area Network (WAN)*. <https://www.nd.gov/itd/services/wide-area-network-wan>. (Accessed on 06/10/2021).
- [182] Department of Administrative Services Connecticut. *Wide Area Networks*. <https://portal.ct.gov/DAS/BEST/Network-Services/Wide-Area-Networks>. (Accessed on 06/10/2021).
- [183] Georgia Technology Authority. *WAN Service*. <https://gta.georgia.gov/gta-services/georgia-enterprise-technology-services-gets/managing-your-gets-services/wan-service>. (Accessed on 06/15/2021).
- [184] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving high utilization with software-driven WAN". In: *Proceedings of the ACM SIGCOMM Conference*. 2013.
- [185] The Atlantic. *The Creepy, Long-Standing Practice of Undersea Cable Tapping (06/13/2013)*. <https://www.theatlantic.com/international/archive/2013/07/the-creepy-long-standing-practice-of-undersea-cable-tapping/277855/>. (Accessed on 04/06/2021).
- [186] The Guardian. *GCHQ taps fibre-optic cables for secret access to world's communications (06/21/2013)*. <https://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa>. (Accessed on 06/08/2021).
- [187] Sandra Kay Miller. *Hacking at the Speed of Light*. Securitysolutions.com. 2006.
- [188] Network Critical. *Passive Fiber TAPs*. <https://www.networkcritical.com/fiber-taps>. (Accessed on 06/08/2021).
- [189] Gigamon. *Passive Fiber Optic Network Tap*. <https://www.gigamon.com/products/access-traffic/network-taps/g-tap-m-series.html>. (Accessed on 06/08/2021).

- [190] Keysight. *Flex Tap Passive Fiber Optical Taps*. <https://www.keysight.com/ch/de/products/network-visibility/network-taps/flex-tap-fiber-optical.html>. (Accessed on 06/08/2021).
- [191] APCON. *Passive Optical Tap*. <https://www.apcon.com/hardware/network-taps/apcon-tap>. (Accessed on 06/08/2021).
- [192] Profitap. *Fiber TAPs*. <https://www.profitap.com/fiber-taps/>. (Accessed on 06/08/2021).
- [193] *IEEE Standard for Local and Metropolitan Area Networks: Media Access Control (MAC) Security*. IEEE Std 802.1AE-2006. 2006.
- [194] Microsoft. *Azure encryption overview*. <https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-overview>. (Accessed on 08/09/2022). 2022.
- [195] Aviatrix. *Is Amazon inter-region peering encrypted?* <https://aviatrix.com/learn-center/answered-access/is-amazon-inter-region-peering-encrypted/>. (Accessed on 08/09/2022).
- [196] Arturo Cabanas. *Managing Security on AWS*. [https://d1.awsstatic.com/events/Summits/PublicSector2020/Managing\\_Security\\_on\\_AWS\\_MGMT104\\_ENGLISH.pdf](https://d1.awsstatic.com/events/Summits/PublicSector2020/Managing_Security_on_AWS_MGMT104_ENGLISH.pdf). (Accessed on 08/09/2022). 2020.
- [197] OVH. *OVH Tasks*. <http://travaux.ovh.net/?do=details&id=10705&>. (Accessed on 08/09/2022). 2014.
- [198] C. Hopps. *IP-TFS: IP Traffic Flow Security Using Aggregation and Fragmentation*. <https://datatracker.ietf.org/doc/draft-ietf-ipsecme-iptfs/>. (Accessed on 08/09/2022). 2021.
- [199] Don Fedyk. *Ethernet-Traffic Flow Security*. <https://www.ieee802.org/1/files/public/docs2019/new-fedyk-traffic-flow-security-0219.pdf>. (Accessed on 08/09/2022). 2019.
- [200] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. "Characterization of encrypted and vpn traffic using time-related". In: *Proceedings of the international conference on information systems security and privacy (ICISSP)*. 2016.
- [201] Hong-Yen Chen and Tsung-Nan Lin. "The Challenge of Only One Flow Problem for Traffic Classification in Identity Obfuscation Environments". In: *IEEE Access* 9 (2021).
- [202] Tal Shapira and Yuval Shavitt. "Flowpic: Encrypted internet traffic classification is as easy as image recognition". In: *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPs)*. 2019.

- [203] Ania Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. "The Loopix Anonymity System". In: *Proceedings of the USENIX Security Symposium*. 2017.
- [204] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Bryan Ford, Jean-Pierre Hubaux, and Joan Feigenbaum. "PriFi: A Low-Latency Local-Area Anonymous Communication Network". In: *arXiv preprint arXiv:1710.10237* (2017).
- [205] Yahoo. *Barefoot Networks, Google Cloud, ONF and P4.org to Showcase P4 Runtime-based Control of Network Switches (10/03/2017)*. <https://finance.yahoo.com/news/barefoot-networks-google-cloud-onf-120000850.html>. (Accessed on 04/15/2021).
- [206] The Fast Mode. *Barefoot Networks Wins Deals from AT&T, Tencent, Alibaba and Baidu for Programmable Switches (06/08/2017)*. <https://www.thefastmode.com/technology-solutions/10724-barefoot-networks-wins-deals-from-at-t-tencent-alibaba-and-baidu-for-programmable-switches>. (Accessed on 04/15/2021).
- [207] S. Kent and K. Seo. *Security Architecture for the Internet Protocol*. RFC 4301. <http://www.rfc-editor.org/rfc/rfc4301.txt>. 2005.
- [208] Jonas Bushart and Christian Rossow. "Padding Ain't Enough: Assessing the Privacy Guarantees of Encrypted DNS". In: *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI)*. 2020.
- [209] X. Wang, S. Chen, and S. Jajodia. "Network Flow Watermarking Attack on Low-Latency Anonymous Communication Systems". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2007.
- [210] S. Feghhi and D. J. Leith. "A Web Traffic Analysis Attack Using Only Timing Information". In: *IEEE Transactions on Information Forensics and Security* 11.8 (2016).
- [211] Craig Hill and Stephen Orr. *Innovations in Ethernet Encryption (802.1AE - MACsec) for Securing High Speed (1-100GE) WAN Deployments*. <https://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Security/MACsec/WP-High-Speed-WAN-Encrypt-MACsec.pdf>. (Accessed on 08/09/2022). 2016.
- [212] CAIDA. *Trace Statistics for CAIDA Passive OC48 and OC192 Traces*. [https://www.caida.org/data/passive/trace\\_stats/](https://www.caida.org/data/passive/trace_stats/). (Accessed on 08/09/2022).

- [213] CAIDA. *The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019)*. <https://www.caida.org/data/passive/passive-dataset.xml>. (Accessed on 08/09/2022).
- [214] Cisco. *System Security Configuration Guide for Cisco 8000 Series Routers, IOS XR Release 7.0.x - Implementing Trustworthy Systems*. <https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/security/70x/b-system-security-cg-cisco8000-70x/implementing-trustworthy-systems.html>. (Accessed on 08/09/2022).
- [215] Arista. *Arista 7170 Series*. <https://www.arista.com/en/products/7170-series>. (Accessed on 08/09/2022).
- [216] Netberg. *Aurora 710*. <https://netbergtw.com/products/aurora-710>. (Accessed on 08/09/2022).
- [217] Edgecore Networks. *WEDGE 100BF-65X*. <https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=334>. (Accessed on 08/09/2022).
- [218] Scapy. <https://scapy.net/>.
- [219] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. “MoonGen: A Scriptable High-Speed Packet Generator”. In: *Proceedings of the ACM Internet Measurement Conference (IMC)*. Tokyo, Japan, 2015.
- [220] *iPerf*. <https://iperf.fr/>.
- [221] Catapult Project. *Web Page Replay*. [https://github.com/catapult-project/catapult/tree/master/web\\_page\\_replay\\_go](https://github.com/catapult-project/catapult/tree/master/web_page_replay_go). (Accessed on 04/08/2021).
- [222] *PJSIP*. <https://www.pjsip.org/>.
- [223] Alexa. *Top Sites in United States*. <https://www.alexa.com/topsites/countries/US>. (Accessed on 07/21/2020).
- [224] FMAD.IO. *100% Line Rate 100G packet capture*. <https://www.fmad.io/products-100G-packet-capture.html>. (Accessed on 04/08/2021).
- [225] *TCPDUMP & LIBPCAP*. <https://www.tcpcdump.org/>.
- [226] *Keras EarlyStopping*. [https://keras.io/api/callbacks/early\\_stopping/](https://keras.io/api/callbacks/early_stopping/). (Accessed on 08/09/2022).
- [227] David Hancock and Jacobus Van der Merwe. “Hyper4: Using p4 to virtualize the programmable data plane”. In: *Proceedings of the ACM Conference on emerging Networking Experiments and Technologies (CoNEXT)*. 2016.

- [228] Wen Ming Liu, Lingyu Wang, Pengsu Cheng, Kui Ren, Shunzhi Zhu, and Mourad Debbabi. "PPTP: Privacy-Preserving Traffic Padding in Web-Based Applications". In: *IEEE Transactions on Dependable and Secure Computing* 11.6 (2014).
- [229] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. "Effective Attacks and Provable Defenses for Website Fingerprinting". In: *Proceedings of the USENIX Security Symposium*. 2014.
- [230] Tao Wang and Ian Goldberg. "Walkie-talkie: An efficient defense against passive website fingerprinting attacks". In: *Proceedings of the USENIX Security Symposium*. 2017.
- [231] *GnuTLS*. <https://gnutls.org>.
- [232] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. "Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems". In: *ACM SIGCOMM Computer Communication Review (CCR)* 45.4 (2015).
- [233] Noah Apthorpe, Danny Yuxing Huang, Dillon Reisman, Arvind Narayanan, and Nick Feamster. "Keeping the Smart Home Private with Smart(er) IoT Traffic Shaping". In: *Proceedings on Privacy Enhancing Technologies* 2019.3 (2019).
- [234] *Tor Project*. <https://www.torproject.org/>.
- [235] T. Ylonen and C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. <http://www.rfc-editor.org/rfc/rfc4253.txt>. 2006.
- [236] S. Kent. *IP Encapsulating Security Payload (ESP)*. RFC 4303. <http://www.rfc-editor.org/rfc/rfc4303.txt>. 2005.
- [237] Michael G Reed, Paul F Syverson, and David M Goldschlag. "Anonymous connections and onion routing". In: *IEEE Journal on Selected areas in Communications* 16.4 (1998).
- [238] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. "Scalable onion routing with torsk". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2009.
- [239] David Goldschlag, Michael Reed, and Paul Syverson. "Onion routing". In: *Communications of the ACM* 42.2 (1999).



- [240] David Chaum. "The dining cryptographers problem: Unconditional sender and recipient untraceability". In: *Journal of cryptology* 1.1 (1988).
- [241] Charles V Wright, Scott E Coull, and Fabian Monrose. "Traffic Morphing: An Efficient Defense Against Statistical Traffic Analysis." In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2009.
- [242] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2012.
- [243] Vladimir Gurevich and Andy Fingerhut. *P416 Programming for Intel Tofino Using Intel P4 Studio*. Tech. rep. Intel, 2021.
- [244] AMS-IX Amsterdam. <https://www.ams-ix.net/ams>.
- [245] AMS-IX. *Total Stats AMS-IX Amsterdam*. <https://www.ams-ix.net/ams/documentation/total-stats>. (Accessed on 06/28/2022).
- [246] Techcrunch. *WhatsApp Hits 100 Million Calls per Day (06/24/2016)*. <https://social.techcrunch.com/2016/06/24/whatsapp-hits-100-million-calls-per-day/>. (Accessed on 05/30/2022).
- [247] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. "Flow length and size distributions in campus Internet traffic". In: *Computer Communications* 167 (2021).
- [248] Bright Data. *Data Collection Use Cases*. <https://brightdata.com/use-cases>. (Accessed on 06/15/2022).
- [249] FraudLabs. *10 Measures to Reduce Credit Card Fraud for Internet Merchants*. <https://www.fraudlabspro.com/resources/tutorials/10-measures-to-reduce-credit-card-fraud-for-internet-merchants/>. (Accessed on 06/15/2022).
- [250] IPQualityScore. *How Residential Proxies Enable Fraud*. <https://www.ipqualityscore.com/articles/view/13/how-residential-proxies-enable-fraud>. (Accessed on 06/16/2022).
- [251] Xianghang Mi, Xuan Feng, Xiaojing Liao, Baojun Liu, XiaoFeng Wang, Feng Qian, Zhou Li, Sumayah Alrwais, Limin Sun, and Ying Liu. "Resident evil: Understanding residential IP proxy as a dark service". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2019.

- [252] Akihiro Hanzawa and Hiroaki Kikuchi. "Analysis on malicious residential hosts activities exploited by residential IP proxy services". In: *Proceedings of the International Conference on Information Security Applications*. 2020.
- [253] Manos Antonakakis Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, et al. "Understanding the Mirai Botnet". In: *Proceedings of the USENIX Security Symposium*. 2017.
- [254] ProxyNova. *List of Free Public Proxy Servers*. <https://www.proxynova.com/proxy-server-list/>. (Accessed on 06/16/2022).
- [255] Didsoft. *Free Proxy List*. <https://free-proxy-list.net/>. (Accessed on 06/16/2022).
- [256] *Bright Data*. <https://brightdata.com/>.
- [257] *Storm Proxies*. <https://stormproxies.com/>.
- [258] *GeoSurf*. <https://www.geosurf.com/>.
- [259] *EarnApp*. <https://earnapp.com/>.
- [260] ZDNet. *New Windows malware sets up proxies on your PC to relay malicious traffic (08/01/2019)*. <https://www.zdnet.com/article/new-windows-malware-sets-up-proxies-on-your-pc-to-relay-malicious-traffic/>. (Accessed on 08/05/2022).
- [261] Ronan O'Flaherty and Kevin Curran. "Detecting anonymising proxy usage on the internet". In: *Wireless personal communications* 75:4 (2014).
- [262] Shane Miller, Kevin Curran, and Tom Lunney. "Securing the internet through the detection of anonymous proxy usage". In: *Proceedings of the World Congress on Internet Security (WorldCIS)*. 2015.
- [263] Rwei-Min Lin, Yi-Chun Chou, and Kuan-Ta Chen. "Stepping stone detection at the server side". In: *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2011.
- [264] Yanjie He and Wei Li. "A Novel Lightweight Anonymous Proxy Traffic Detection Method Based on Spatio-Temporal Features". In: *Sensors* 22:11 (2022).
- [265] Vahid Aghaei-Foroushani and A. Nur Zincir-Heywood. "A Proxy Identifier Based on Patterns in Traffic Flows". In: *Proceedings of the International Symposium on High Assurance Systems Engineering*. 2015.

- [266] Ziyue Deng, Zihan Liu, Zhouguo Chen, and Yubin Guo. "The random forest based detection of shadowsock's traffic". In: *Proceedings of the IEEE International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*. 2017.
- [267] Xuemei Zeng, Xingshu Chen, Guolin Shao, Tao He, Zhenhui Han, Yi Wen, and Qixu Wang. "Flow Context and Host Behavior Based Shadowsocks's Traffic Identification". In: *IEEE Access* 7 (2019).
- [268] Shane Miller, Kevin Curran, and Tom Lunney. "Detection of Anonymising Proxies Using Machine Learning". In: *International Journal of Digital Crime and Forensics (IJDCF)* 13.6 (2021).
- [269] Nan Zhiang, Tiantian Wu, Yuening Zhang, and Mingzhong Xiao. "Shadowsocks Traffic Identification Based on Convolutional Neural Network". In: *Proceedings of the International Conference on Information Science and Education (ICISE-IE)*. 2020.
- [270] Zhen-Hui Han, Xing-Shu Chen, Xue-Mei Zeng, Yi Zhu, and Ming-Yong Yin. "Detecting Proxy User Based on Communication Behavior Portrait". In: *The Computer Journal* 62.12 (2019).
- [271] Altug Tosun, Michele De Donno, Nicola Dragoni, and Xenofon Fafoutis. "Resip host detection: Identification of malicious residential ip proxy flows". In: *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE)*. 2021.
- [272] Ari Luotonen and Kevin Altis. *World-Wide Web Proxies*. <https://courses.cs.vt.edu/~cs4244/spring.09/documents/Proxies.pdf>. (Accessed on 07/18/2022). 1994.
- [273] Oxylabs. *SOCKS vs HTTP Proxy: What Is the Difference?* <https://oxylabs.io/blog/socks-vs-http-proxy>. (Accessed on 07/18/2022).
- [274] CompTIA. *Network Address Translation*. <https://www.comptia.org/content/guides/what-is-network-address-translation>. (Accessed on 07/18/2022).
- [275] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. "Resurf: Reconstructing web-surfing activity from network traffic". In: *Proceedings of the IFIP Networking Conference*. 2013.
- [276] David Gugelmann, Fabian Gasser, Bernhard Ager, and Vincent Lenders. "Hviz: HTTP (S) traffic aggregation and visualization for network forensics". In: *Digital Investigation* 12 (2015).

- [277] Sebastian Reidy, Roland Meier, and Laurent Vanbever. *In-Network Detection of Proxy Servers*. Tech. rep. ETH Zürich (Semester thesis), 2022.
- [278] P4.org API Working Group. *P4Runtime Specification: Digests*. <https://p4.org/p4-spec/p4runtime/v1.3.0/P4Runtime-Spec.html#sec-digestentry>. (Accessed on 07/23/2022). 2020.
- [279] Scikit-learn Developers. *Clustering*. <https://scikit-learn.org/stable/modules/clustering.html#k-means>. (Accessed on 07/15/2022).
- [280] *DigitalOcean*. <https://www.digitalocean.com/>.
- [281] *Selenium*. <https://www.selenium.dev/>.
- [282] *Bright Initiative*. <https://brightinitiative.com/>.
- [283] *The Apache HTTP Server Project*. <https://httpd.apache.org/>.
- [284] *Let's Encrypt*. <https://letsencrypt.org/>.
- [285] AT&T. *Internet Plans & Services*. <https://www.att.com/internet/internet-service-plans/>. (Accessed on 07/21/2022).
- [286] ManKier. *Package iproute-tc*. <https://www.mankier.com/package/iproute-tc>. (Accessed on 07/21/2022).
- [287] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. *Transport Layer Security (TLS) Extensions*. RFC 3546. <http://www.rfc-editor.org/rfc/rfc3546.txt>. 2003.
- [288] Martin Husak, Milan Cermak, Tomas Jirsik, and Pavel Celeda. "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting". In: *EURASIP Journal on Information Security* 2016.1 (2016).
- [289] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. <http://www.rfc-editor.org/rfc/rfc7230.txt>. 2014.
- [290] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. <http://www.rfc-editor.org/rfc/rfc7540.txt>. 2015.
- [291] M. Bishop. *HTTP/3*. RFC 9114. <http://www.rfc-editor.org/rfc/rfc9114.txt>. 2022.
- [292] W3Techs. *Usage Statistics of HTTP/2 for Websites, July 2022*. <https://w3techs.com/technologies/details/ce-http2>. (Accessed on 07/23/2022).

- [293] W3Techs. *Usage Statistics of HTTP/3 for Websites, July 2022*. <https://w3techs.com/technologies/details/ce-http3>. (Accessed on 07/23/2022).
- [294] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching". In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2017.
- [295] MDN. *Base64*. <https://developer.mozilla.org/en-US/docs/Glossary/Base64>. (Accessed on 07/23/2022).
- [296] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. "Beaucoup: Answering many network traffic queries, one memory update at a time". In: *Proceedings of the ACM SIGCOMM Conference*. 2020.
- [297] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. "Sonata: Query-driven streaming network telemetry". In: *Proceedings of the ACM SIGCOMM Conference*. 2018.
- [298] Nate Foster, Nick McKeown, Jennifer Rexford, Guru Parulkar, Larry Peterson, and Oguz Sunay. "Using deep programmability to put network owners in control". In: *ACM SIGCOMM Computer Communication Review (CCR)* 50.4 (2020).
- [299] Pierangela Samarati and Latanya Sweeney. *Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression*. Tech. rep. SRI International, 1998.