A SECURITY FRAMEWORK FOR SOFTWARE-DEFINED NETWORKS

by

Beytüllah Yiğit

B.S., Computer Engineering, Boğaziçi University, 2012

M.S., Computer Engineering, Boğaziçi University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2024

ACKNOWLEDGEMENTS

I would like to thank to my thesis supervisors Prof. Fatih Alagöz and Ph.D. Gürkan Gür for their support and guidance during my thesis. I am really grateful to my advisors for all their continuous support, invaluable feedback and contributions. They spent a lot of time to review and improve my studies. I would like to express my deep and sincere gratitude to them. This thesis would certainly not come to an end without them.

I also would like to thank to Prof. Bernhard Tellenbach, Prof. Tuna Tuğcu, and Prof. Şerif Bahtiyar for their valuable evaluation during each stage of progress and final defense presentations. Their insightful comments and suggestions have significantly contributed to refining the quality and rigor of this work. Special thanks are also to Prof. Berk Canberk and Prof. Birkan Yılmaz for their valuable contributions as members of my thesis committee. Their expertise and feedback have played a pivotal role in enhancing the scholarly merit of this thesis.

I thank to Meltem for giving me the strength to live. Without you, I would be alone.

To my beloved son, Halil Alp, you are the true joy of my life and your love has been a constant source of motivation and inspiration.

Finally, I dedicate this thesis to my dear parents, Kenan and Esme, to thank them for their unconditional love, trust, appreciation and support.

ABSTRACT

A SECURITY FRAMEWORK FOR SOFTWARE-DEFINED NETWORKS

Software-Defined Networking (SDN) emerges as a transformative technology, offering the flexibility and scalability crucial for modern digital services. While SDN presents opportunities to address security shortcomings in traditional networks, it also introduces new vulnerabilities, particularly in data and control plane communications. This thesis proposes a security framework to mitigate these risks, focusing on securing data exchange among SDN entities (controllers, switches, and applications). In our model, Transport Layer Security (TLS) can be activated between SDN nodes to ensure confidentiality, integrity, authentication, and authorization, leveraging specialized certificate fields. Additionally, an integrated security module enhances communication security by enforcing Access Control Lists (ACLs), hardening TLS configuration, and mitigating the risk of private key hijacking. From an availability perspective, the limited flow table capacity of switches leads to table saturation attacks. Besides, disaggregated and layered architecture of SDN is susceptible to time-based fingerprinting attacks. To address this, we introduce an automated attacker tool that utilizes probing-based measurements for fingerprinting. This tool can discern network types (SDN or traditional), extract flow rule timeout values (hard and idle), determine flow table utilization rate, size, and replacement algorithm. Armed with this information, the attacker can execute intelligent saturation attacks. Furthermore, we propose a lightweight defense mechanism that randomizes rule timeouts and proactively deletes flow rules based on network status. Through comprehensive simulations under various network conditions, we demonstrate the efficacy of our proposed techniques, achieving superior success rates across diverse scenarios.

ÖZET

YAZILIM TANIMLI AĞLAR İÇİN BİR GÜVENLİK ÇERÇEVESİ

Yazılım Tanımlı Ağ (SDN), modern dijital hizmetler için çok önemli olan esneklik ve ölçeklenebilirliği sunan dönüştürücü bir teknoloji olarak ortaya çıkmaktadır. SDN, geleneksel ağlardaki güvenlik eksikliklerini gidermek için fırsatlar sunarken, özellikle veri ve kontrol düzlemi iletişimlerinde yeni güvenlik açıklarını da beraberinde getirmektedir. Bu tez, SDN varlıkları (kontrolcüler, anahtarlayıcılar ve uygulamalar) arasındaki veri alışverişini güvence altına almaya odaklanarak bu riskleri azaltmak için bir güvenlik çerçevesi önermektedir. Modelimizde, özel sertifika alanlarından yararlanarak gizlilik, bütünlük, kimlik doğrulama ve yetkilendirme sağlamak için SDN düğümleri arasında Taşıma Katmanı Güvenliği (TLS) etkinleştirilebilir. Ek olarak, entegre bir güvenlik modülü Erişim Kontrol Listelerini (ACL'ler) zorlayarak, TLS yapılandırmasını sertleştirerek ve özel anahtar ele geçirme riskini azaltarak iletişim güvenliğini artırır. Kullanılabilirlik açısından bakıldığında, anahtarların sınırlı akış tablosu kapasitesi tablo doygunluğu saldırılarına yol açmaktadır. Ayrıca, SDN'nin ayrıştırılmış ve katmanlı mimarisi zaman tabanlı parmak izi saldırılarına karşı hassastır. Bunu ele almak için, parmak izi öğrenmek için problama tabanlı ölçümleri kullanan otomatik bir saldırgan aracı sunuyoruz. Bu araç ağ türlerini (SDN veya geleneksel) ayırt edebilir, akış kuralı zaman aşımı değerlerini (sabit ve avare) çıkarabilir, akış tablosu kullanım oranını, boyutunu ve değiştirme algoritmasını belirleyebilir. Bu bilgilerle donanmış olan saldırgan, akıllı tablo doygunluk saldırıları gerçekleştirebilir. Ayrıca, kural zaman aşımlarını rastgele hale getiren ve ağ durumuna göre akış kurallarını proaktif olarak silen hafif bir savunma mekanizması öneriyoruz. Çeşitli ağ koşulları altında gerçekleştirdiğimiz kapsamlı simülasyonlar sayesinde, önerdiğimiz tekniklerin etkinliğini ve farklı senaryolarda üstün başarı oranları elde ettiğimizi gösteriyoruz.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES ix
LIST OF SYMBOLS xi
LIST OF ACRONYMS/ABBREVIATIONS xiii
1. INTRODUCTION 1
2. SECURED COMMUNICATION CHANNELS IN SOFTWARE-DEFINED NETWORKS 8
2.1. Introduction 8
2.2. Related Work 9
2.3. SECAS: A Secure Communication Architecture for Software-Defined Networks 12
2.3.1. SECAS Key Generation Model for SDN 12
2.3.2. SECAS Security Module (*SecMod*) 15
2.3.2.1. TLS Hardening 16
2.3.2.2. Security Management Functions 16
2.3.2.3. TLS-less Operation 17
2.3.2.4. Access Control 18
2.3.2.5. Protection against Private Key Hijacking 18
2.3.2.6. Security Monitoring and Situation Awareness 19
2.3.2.7. How They All Fit Together – SecMod Operation Flow 19
2.4. Discussion 21
2.5. Conclusions 24
3. NETWORK FINGERPRINTING VIA TIMING ATTACKS AND DEFENSE IN SOFTWARE DEFINED NETWORKS 25
3.1. Introduction 25

3.2. SDN Fundamentals . . . 27
3.3. Related Work . . . 28
3.4. Robust and Automated Fingerprinting Agent for SDN (*RAFA*) . . . 34
3.4.1. Network Type Fingerprinting . . . 35
3.4.2. Timeout Fingerprinting . . . 36
3.5. Lightweight Defense against SDN Timeout Fingerprinting . . . 42
3.6. Simulations and Performance Evaluation . . . 45
3.6.1. Network Type Fingerprinting Experiments . . . 46
3.6.2. Timeout Fingerprinting Simulations . . . 48
3.6.3. Fingerprinting Countermeasure Simulations . . . 51
3.7. Discussion . . . 52
3.8. Conclusions . . . 56
4. UNMASKING SDN FLOW TABLE SATURATION: FINGERPRINTING, ATTACKS AND DEFENSES . . . 57
4.1. Introduction . . . 57
4.2. SDN Fundamentals . . . 58
4.3. Related Work . . . 59
4.4. Table Analyzer and Saturation Orchestrator for SDN (*TASOS*) . . . 66
4.5. Lightweight Defense against Flow Table Fingerprinting and Saturation Attacks (*LIDISA*) . . . 73
4.5.1. Design and Operation . . . 75
4.6. Simulations and Performance Evaluation . . . 77
4.6.1. Fingerprinting and Saturation Experiments . . . 77
4.6.2. Fingerprinting Countermeasure Simulations . . . 82
4.7. Discussion . . . 83
4.8. Conclusions . . . 86
5. CONCLUSION . . . 87
REFERENCES . . . 90

LIST OF FIGURES

Figure 1.1. SDN architecture and threat vector (TV) map [10]. 2

Figure 2.1. Certificate and key pair (KP) generation in SECAS key generation model. 14

Figure 2.2. Operation flow of *SecMod*. 20

Figure 3.1. SDN packet processing flowchart. 28

Figure 3.2. Hard timeout calculation algorithm. 38

Figure 3.3. Idle timeout calculation algorithm. 40

Figure 3.4. Simulation topology for *RAFA* and *DAFTR*. 45

Figure 4.1. Flow table fingerprinting algorithm. 70

Figure 4.2. Replacement algorithm identifier. 71

Figure 4.3. Simulation topology for *TASOS* and *LIDISA*. 77

LIST OF TABLES

Table 2.1. Related literature on SDN communication channel security. 9

Table 2.2. TLS support in various commercial switches. 17

Table 2.3. Certificate generation time in SECAS. 23

Table 3.1. Related literature on SDN fingerprinting. 30

Table 3.2. Parameters for timeout fingerprinting algorithms. 36

Table 3.3. Parameters for timeout defense mechanism. 44

Table 3.4. Network parameters for network type simulations. 46

Table 3.5. Network fingerprinting results for hop count 1. 47

Table 3.6. Network parameters for timeout simulations. 49

Table 3.7. Idle timeout simulation results. 49

Table 3.8. Hard timeout simulation results for [17]. 50

Table 3.9. Idle timeout simulation results for [17]. 50

Table 3.10. Hard timeout simulation results for [41]. 50

Table 3.11. Idle timeout simulation results for [41]. 51

Table 4.1. Related work of flow table fingerprinting and saturation attack. . . 65

Table 4.2. Parameters for flow table fingerprinting algorithm. 68

Table 4.3. Parameters for *LIDISA* defense mechanism. 76

Table 4.4. Network parameters. 78

Table 4.5. Algorithm parameters. 78

Table 4.6. FT_{size} accuracy for different network parameters. 79

Table 4.7. Algorithm parameters. 80

Table 4.8. Flow table size fingerprinting accuracy results. 82

LIST OF SYMBOLS

$\mathcal{C}_a$	Application certificate
$\mathcal{C}_c$	Controller certificate
$\mathcal{C}_s$	Switch certificate
$FD_{\text{threshold}}$	Flow duration threshold for the flow removal messages (in seconds)
$flowCountPerSecond$	Number of new flows per second
$flowUpdatePeriod$	Period of the flow update mechanism (in seconds)
FRM_{Count}	Flow removal messages count
FTR_{Deletion}	Ratio of flow rules for deletion
$FTUB_{\text{threshold}}$	Flow table utilization base threshold
$FTU_{\text{threshold}}$	Flow table utilization threshold
FT_{residual}	Residual flow size of the flow table
FT_{size}	Flow table size
Nsf	Number of the selected flows
$PacketIn_{\text{Count}}$	Packet-in count
$PC_{\text{threshold}}$	Packet count threshold for the flow removal messages
$Ratio_{\text{threshold}}$	Ratio threshold between the packet-in and the flow removed messages
RTT_{mean}	RTT mean
RTT_{std}	RTT standard deviation
T_0	Current time
T_1	RTT required for the Fast Path
T_2	RTT required for the Slow Path
T_3	RTT required for the Slowest Path
$T_{\text{hardActual}}$	Actual hard timeout
T_{hard}	Hard timeout
$T_{\text{idleActual}}$	Actual idle timeout
T_{idle}	Idle timeout

T_{residual}	Residual idle timeout time used for flow update
T_{sleep}	The time between probing packets used for the idle timeout fingerprinting (in seconds)
T_{step}	Sleep interval time (in seconds) which can be between 0 and 1
$T_{\text{threshold}}$	RTT threshold
α	Significance level used to compute the confidence level
β	Randomization coefficient
ρ	Majority coefficient
τ	iterationCount

LIST OF ACRONYMS/ABBREVIATIONS

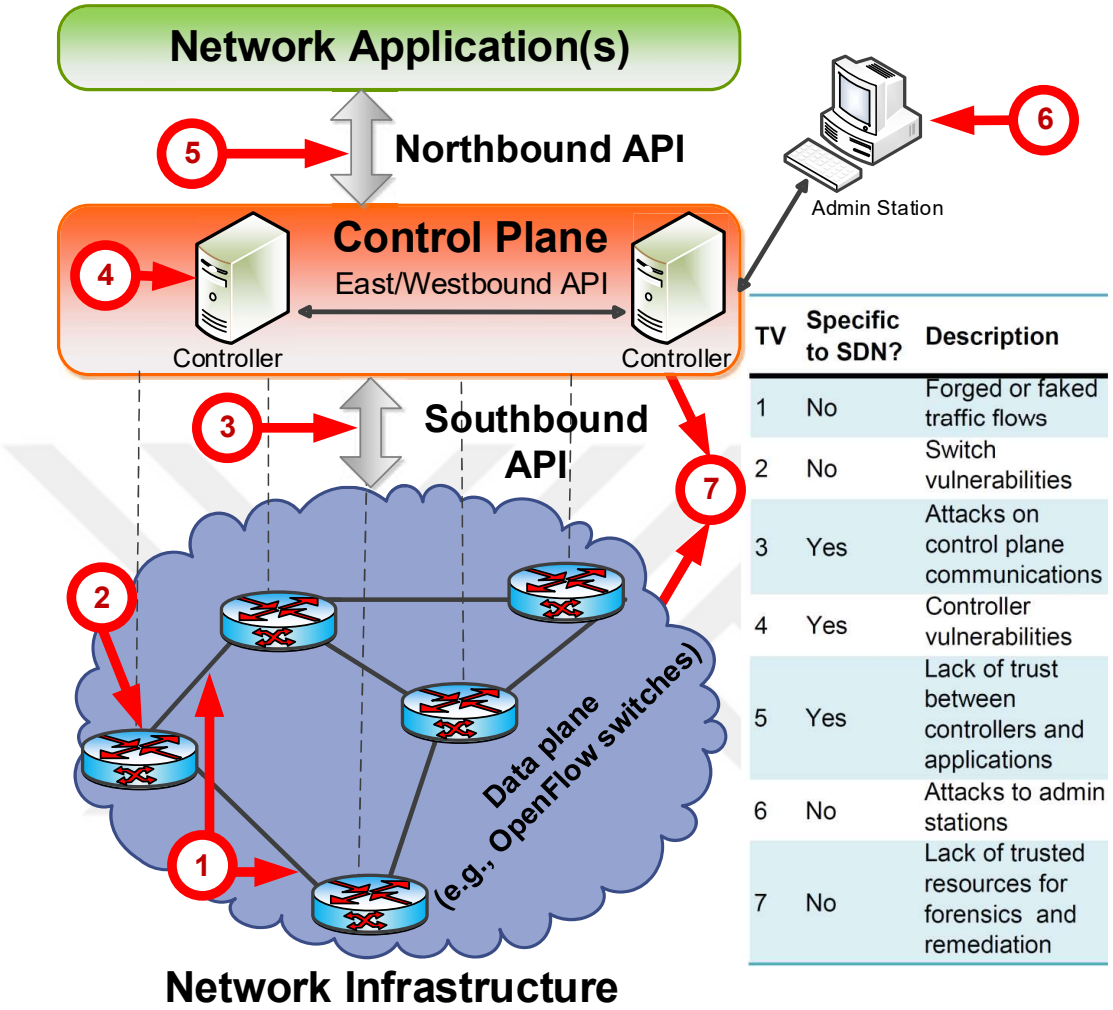
ACL	Access Control List
ARP	Address Resolution Protocol
BT	Background Traffic
BW	Bandwidth
CA	Certificate Authority
CRL	Certificate Revocation List
CCOR	Control Channel Occupation Rate
DAFTR	Dynamic Adaptive Flow Timeout Randomization
DPID	Data Plane Identifier
DTLS	Datagram Transport Layer Security
DoS	Denial of Service
DDoS	Distributed Denial of Service
EWB	East-westbound
ECC	Elliptic Curve Cryptography
FPPT	First Packet Processing Time
HIP	Host Identity Protocol
IBC	Identity-based Cryptography
JKS	Java Key Store
LIDISA	Lightweight Defense against Flow Table Fingerprinting and Saturation Attacks
LDA	Linear Discriminant Analysis
LLDP	Link Layer Discovery Protocol
MTD	Moving Target Defense
NB	Northbound
NoAF	Number of Active Flows
PPS	Packet-Per-Second
PKG	Private Key Generator
PKI	Public Key Infrastructure

PKCS	Public Key Cryptography Standards
RA	Replacement Algorithm
RAFA	Robust and Automated Fingerprinting Agent for SDN
RTT	Round-Trip Time
SECAS	Secure Communication Architecture for Software-Defined Networks
SecMod	Security Module
SDN	Software-Defined Networking
SB	Southbound
TASOS	Table Analyzer and Saturation Orchestrator for SDN
TCAM	Ternary Content-Addressable Memory
TV	Threat Vector
TLS	Transport Layer Security

1. INTRODUCTION

Next-generation networks are envisioned to cultivate the future of a universally connected, data-intensive, and intelligent society. Serving this pivotal goal, Software-Defined Networking (SDN) integrates programmability as an inherent attribute of networks, ushering in automation and enhanced flexibility in network management by decoupling the control plane from the data plane. In contrast to traditional networks, SDN empowers a central controller to dynamically program the data plane and execute applications and network policies with a precise granularity of information, enabling the efficient steering and control of network traffic. Nonetheless, the improved flexibility and control achieved via SDN are accompanied by new security vulnerabilities, such as control plane saturation attacks [1]. Thus, the realization of SDN's full potential is hampered by security concerns, challenging its widespread practical implementation. Consequently, the security of SDN constitutes a dynamic and multifaceted domain of research [2–8].

SDN relies on the main idea of decoupling the control and the data planes: network intelligence and network states are logically centralized while the underlying network infrastructure is abstracted by the controller for different applications [9]. The software-based controller determines the network decisions (*control plane*), while switches handle the actual forwarding of data packets (*data plane*). A simplified view of a generic SDN architecture and threats against SDN are shown in Figure 1.1 [10]. In this setting, the controller can be used to enhance network security by exploiting the global network view. Security applications at the controller can analyze and correlate network data to infer the network status spatially and temporally. Furthermore, they can run attack detection algorithms and help to understand the nature of various network threats and security incidents. The information collected and analyzed by the control plane can then be used to enforce appropriate security policies in the data plane. Overall, these characteristics provide potent capabilities to improve the security level of the communication infrastructure.



TV	Specific to SDN?	Description
1	No	Forged or faked traffic flows
2	No	Switch vulnerabilities
3	Yes	Attacks on control plane communications
4	Yes	Controller vulnerabilities
5	Yes	Lack of trust between controllers and applications
6	No	Attacks to admin stations
7	No	Lack of trusted resources for forensics and remediation

Figure 1.1. SDN architecture and threat vector (TV) map [10].

Although SDN offers opportunities to address security related network issues, it also involves various comparable challenges which cast worthwhile technical questions such as control plane saturation attacks and detection of malicious application. Threat vectors against SDN are also shown in Figure 1.1 [10]. As seen in the figure, threat vectors ①, ②, ⑥ and ⑦ already exist in traditional networks. On the contrary, threat vectors ③, ④ and ⑤ are specific to SDN and stem from the separation of the control and data planes and the addition of a new component to the network, namely the controller. Threat ① considers generating malicious flows for Denial of Service (DoS) attacks. System and software vulnerabilities in the switch, controller and admin stations result in Threat ②, ④ and ⑥, respectively. Threat ③ includes attacks on

the confidentiality, integrity and availability of the communication channel between the control and data planes. For example, they can be carried out by exploiting a (configuration) flaw in the TLS protocol or by carrying out a DoS attack on a network link or communication endpoint. Threat ⑤ arises from lack of trust between the controllers and applications while Threat ⑦ points lack of reliable and trustworthy information to understand the cause of a detected problem. Threats ⑦ also includes absence of secure and stable system snapshots for remediation. Therefore, security in SDN is a vibrant research field with a wide range of topics. Most of them can be assigned to the body of work either on using SDN as a security enabler or on how to implement SDN architectures so that they are themselves secure. In this thesis, we focus on the latter and consider the security of SDN communication channels.

Specifically, there exist three key communication interfaces in SDN: SB (southbound), EWB (east-westbound) and NB (northbound) as shown in Figure 1.1. SB interface is located between the controller and switches. OpenFlow has emerged as the *de facto* SB protocol. OpenFlow specification recommends the use of TLS with mutual authentication between the controllers and their switches. (The specification also mentions auxiliary OpenFlow connection which can use Datagram Transport Layer Security (DTLS)). However, this feature is optional starting with OpenFlow version 1.1.0. In the control plane, the EWB interface covers controller communications in a cluster while the NB interface provides communication between network applications and the controller. Although EWB and NB interfaces do not have agreed-upon protocols, a REST API, which is TLS compliant, is typically used for them.

In essence, TLS is imperative for protecting communication in SDN interfaces. Nevertheless, relying on pure TLS usage alone is not sufficient, albeit being instrumental. First, TLS must be hardened in the whole network since it contains some security flaws due to obsolete or vulnerable versions. Furthermore, TLS must be used with mutual authentication to limit the possibility of adding a bogus device. Hence, the communicating entities should have proper certificates. In that situation, one needs to consider the security of these certificates and handle possible outcomes of private key

hijacking (stealing). Additionally, an Access Control List (ACL) for network elements should regulate access since a switch or a controller is authorized to connect to only a subset of controllers. From the network management perspective, a security architecture must handle network devices which lack TLS support, generate alarms about security incidents and allow per-channel activation/deactivation of TLS protocol.

For that purpose, we propose *Secure Communication Architecture for Software-Defined Networks (SECAS)* to overcome the aforementioned security problems. We design a cryptographic key generation model and implement it with an SDN application that can generate and manage key material for communicating entities and then issue and manage corresponding X.509 certificates. Special certificate fields are designed to limit the impact of private key hijacking. Moreover, TLS configuration is hardened for all interfaces through features such as removal of TLS version and removal of support for weak cipher suites. From the operational perspective, real-time security alarms are dispatched for security breaches like unauthorized access requests. TLS usage in each interface can be managed or monitored separately via a user interface.

TLS can handle confidentiality, integrity, authentication and authorization but not availability. One of the primary threat vectors within the data plane is the flow table of SDN switches. Due to constraints such as high power consumption and cost, commercial SDN/OpenFlow switches possess restricted flow table capacities, typically ranging from hundreds to thousands [11]. This capacity is often inadequate for managing the millions of flows commonly encountered in enterprise and data center networks [12]. It is evident that future switches are expected to support larger flow tables. However, the efficiency of flow table management remains a significant concern. This is particularly important because TCAMs (Ternary Content-Addressable Memory) consume much more power and are costly, e.g., reported as approximately 100 times more power-hungry and expensive compared to conventional RAM in [13].

In SDN, given that switches are relatively simplistic devices, they rely on communication with the controller to handle packets that do not align with any existing flow

rule. This communication method can be exploited by attackers in various manners. A notable instance of such an attack involves targeting an SDN switch with the aim of overwhelming its flow table by generating traffic that leads to numerous new flow table entries. Additionally, if the attacker can ascertain the TCAM size of the switch and flow table replacement algorithm, they can fine-tune the flow generation rate to overwhelm the switch with a minimal number of packets efficiently. This restricted packet generation benefits the attacker in terms of performance and reduces the risk of detection before the attack accomplishes its objective.

To assess whether a network is vulnerable to these attacks, an attacker needs to establish whether it operates as an SDN and gather relevant information. One method for identifying an SDN is to exploit the concept of First Packet Processing Time (FPPT), which is defined as the variation in Round-Trip Time (RTT) between the first and second packets of a novel unmatched flow. These techniques are commonly referred to as SDN fingerprinting and are often followed by methods aimed at determining pertinent SDN characteristics, such as flow rule timeout values, flow match fields, active flow rules and flow table capacity [11, 12, 14–24].

Following SDN fingerprinting, attackers can transition from the reconnaissance phase to the subsequent stages of the cyber kill chain and execute attacks that, in the most dire scenarios, have the potential to result in the complete shutdown of the entire network by utilizing the fingerprinting information. Nonetheless, it is essential to recognize that SDN fingerprinting research remains in its nascent stages with the integration of this technology into diverse network infrastructures and predominantly centers on the attacker's perspective, specifically on methods to extract increasingly reliable data from networks. Strategies for preventing attackers from gaining access to such information in the first place are seldom explored. In this thesis, both attack and defense methods are proposed and their performance is investigated by using simulation-based experiments. The main contributions of our work can be summarized as follows:

- *SECAS* — a secure communication architecture including a key generation application to eliminate security threats to the communication channels of SDN.
- A fully automated, robust and reliable SDN fingerprinting module to infer network type (SDN or traditional). Then it effectively fingerprints flow timeout values (hard and idle) and flow table-related details, including size, utilization rate and replacement policy, enabling the execution of efficient saturation attacks using this information.
- Development of an agile and dynamic defense mechanisms that enhances the complexity and expenses for attackers by adjusting flow timeout values and flow table attributes in response to changes in network status.
- Extensive simulations are conducted to demonstrate the high accuracy and adaptability of these proposed methods across various network conditions.

The structure of this thesis is as follows: In Chapter 2, the *SECAS* is described. This is followed by an introduction to a fully automated, robust, and reliable SDN fingerprinting module, *Robust and Automated Fingerprinting Agent for SDN (RAFA)*, designed to infer network type (SDN or traditional) and flow timeout values (hard and idle). Additionally, in that chapter, a lightweight and dynamic defense mechanism is introduced to increase complexity and costs for attackers by adjusting flow timeout values according to network status. We also demonstrate the high accuracy of the proposed methods for different network conditions through extensive simulations. In Chapter 4, a comprehensive and automated module for conducting flow table saturation attacks, named *Table Analyzer and Saturation Orchestrator for SDN (TASOS)*, is described. Furthermore, the exploration delves into the development of an agile and dynamic defense mechanism that adapts flow table attributes in response to changes in network status. Finally, in Chapter 5, we explore potential avenues for future research and conclude the thesis.

Parts of this thesis were previously published in [3,8]. The images created within the scope of this thesis and copyrighted to the publishers have been used in the thesis in accordance with the 'publication policy regarding the reuse of the author's own texts

and graphics' as specified on the publisher's website.

2. SECURED COMMUNICATION CHANNELS IN SOFTWARE-DEFINED NETWORKS

2.1. Introduction

SDN brings new opportunities to alleviate the existing security deficiencies of traditional networks. However, it also introduces new issues, a primary one being the vulnerabilities related to data and control plane communications. This chapter presents a security architecture to address security problems regarding data exchange in software-defined networks.

In this chapter, we propose *Secure Communication Architecture for Software-Defined Networks (SECAS)* to overcome the aforementioned security problems. We design a cryptographic key generation model and implement it with an SDN application that can generate and manage key material for communicating entities (controller, switch and application) and then issue and manage corresponding X.509 certificates. Special certificate fields are designed to limit the impact of private key hijacking. Hence TLS can be activated between SDN nodes to provide confidentiality, integrity, authentication and authorization with special certificate fields.

Besides, an integrated security module further strengthens the communication security by applying Access Control List (ACL). Moreover, TLS configuration is hardened for all interfaces through features such as removal of TLS version and removal of support for weak cipher suites. From the operational perspective, real-time security alarms are dispatched for security breaches like unauthorized access requests. TLS usage in each interface can be managed or monitored separately via a user interface.

This chapter is structured as follows: In the next section, we present the related work on SDN communication security and authentication of the network elements. Our proposal *SECAS* is described in Section 2.3. In Section 2.4, *SECAS* and its adoption

in SDN are discussed from a cost-benefit perspective. Finally, Section 2.5 concludes the chapter.

2.2. Related Work

In this section, we present the literature on communication channel security and authentication of network elements in SDN. Table 2.1 shows some related technical works summarizing their main methods and key generation capability. The papers [25] and [26] do not entail a proposal for SDN security. Samociuk compares security protocols, TLS, SSH and IPSec in [25]. He also mentions about certificate generation via Public Key Infrastructure (PKI). Lam et al. focuses on EWB interface and implements TLS support in that channel for ONOS controller in [26]. However, only intra-cluster communication is secured and inter-cluster communication remains unprotected. Their experiments show that TLS overhead in terms of number of messages is about 5%.

Table 2.1. Related Literature on SDN communication channel security.

Work	**Contributions and employed key method(s)**	**Key Generation**
Samociuk [25]	Advantages and drawbacks of TLS, SSH and IPSec protocols are compared for SB interface.	Not Applicable (N/A)
Lam et al. [26]	Implementation of TLS support for cluster communications (EWB) in ONOS controller	N/A
Lam et al. [27]	Identity-based cryptography to secure SB interfaces	Partially
Kang et al. [28]	Pre-shared secrets for authentication of switches by extending OpenFlow to solve Data Path ID duplication problem.	No

Table 2.1. Related Literature on SDN communication channel security. (cont.)

Agborubere et al. [29]	Review of TLS security flaws and proposal of a TLS extension for OpenFlow	No
Liyanage et al. [30]	Host Identity Protocol to authenticate switches	No
Othman et al. [31]	Not only controller but also switches can install flow rules to other switches by using digital signatures.	No
Cui et al. [32]	A mutual authentication mechanism provided by certificates for NB interface	No
Paladi et al. [33]	A framework to establish a secure communication channel for OpenFlow via certificates is developed. It uses Intel SGX which offers strong isolation of cryptographic data by reserving a dedicated private area in the memory.	Partially
Qin et al. [34]	Authentication for controllers and their messages by using hash chains	Partially
Peng et al. [35]	Usage of Quantum Key Distribution technology to secure SB interface	Partially
Toseef et al. [36]	Certificate-based AAA services for SDN experimental testbeds (e.g. OFELIA)	Partially
Banse et al. [37]	Authentication and authorization of SDN applications by controlling certificate fields	No

Lam et al. propose to apply identity-based cryptography (IBC) to secure SB communications [27]. They modify TLS protocols to use them with IBC. In IBC, public keys are generated according to the identity of the user corresponding to, for

instance, MAC address. The private keys are generated in Private Key Generator (PKG) which is a separate security module. Hence, an entity first creates its public key according to its identity and then communicates with PKG to obtain its private key.

Kang et al. propose a lightweight data plane authentication scheme to solve data plane identifier (DPID) duplication problem where switches have the same DPID causing the controller to malfunction [28]. Liyanage et al. use the Host Identity Protocol (HIP) to authenticate switches in [30]. In HIP, similar to IBC, public keys are used as Host Identity which are then used to mutually authenticate end nodes. Their proposal changes the SDN architecture and switch design by adding security gateways between the controller and switches to use IPSec tunneling.

In [27] and [28], communicating entities need to possess a secret key beforehand which means that they must employ symmetric key distribution. Furthermore, TLS and OpenFlow protocols need to be modified for [27,29] and [28], respectively. Likewise, SDN architecture must include IPSec capable security gateway in [30]. Modification of a widely deployed protocol and extra component integration to SDN limit the usability of these proposals.

The works in the technical literature reveals that incumbent research generally does not address key generation or distribution. Hence, a key generation application for SDN can be beneficial and necessary since it can provide public/private keys and certificates regarding all mentioned works. Moreover, these proposals only focus on a part of the overall problem such as switch authentication or SB interface security. Thus, a holistic approach is crucial for securing SDN communication channels in an efficient and robust way. To the best of our knowledge, our architecture *SECAS* is the first work which provides a complete set of features like certificate generation and protection, ACL, security alarms and TLS hardening for communication security in SDN.

2.3. SECAS: A Secure Communication Architecture for Software-Defined Networks

Software-defined networks entail control plane channels (i.e. SB, NB, EWB links) which must be secured to implement a robust SDN architecture. If the data exchange is not protected, an attacker can impersonate a controller or switch and carry out malicious activities which can have serious consequences, depending on the traffic flowing over this infrastructure.

In this overall setting, a powerful and motivated attacker/adversary can control the network infrastructure; hence block, save, change, drop or replay all communication in the SDN. Moreover, it can deploy fake network entities to initiate a Sybil attack and subsequently disrupt the network communication. Therefore, authentication and authorization are crucial functions for this system.

In this section, we propose *SECAS* for secure communication of controller, switch, and application entities in the SDN and aim to overcome the SDN-specific threats ③ and ⑤. To this end, the attacker/adversary is restricted primarily by the deployed cryptographic methods. *SECAS* consists of two main components: a key generation application to generate required cryptographic material explained in Section 2.3.1 and a security module (*SecMod*) to control TLS usage described in Section 2.3.2.

2.3.1. SECAS Key Generation Model for SDN

The employment of TLS in SDN involves some extra operational steps for facilitating and maintaining secure communications. In particular, certificate generation for network entities, signing of certificates with private keys and distribution of cryptographic material to the network devices are required. To generate certificates and public-private key pairs, a key generation application is instrumental. In the application, a Certificate Authority (CA) is generated as the first step. All other certificates are trusted to CA. Hence, each entity only needs the CA certificate and its own cer-

tificate. Typically, self-signed certificates generated outside of the software-defined network are used to enable TLS in SDN. However, self-signed certificate generation is cumbersome and has scalability issues since these certificates must be transferred to each communicating entity which leads to the common key distribution challenge.

SECAS key generation application contains only one CA and no intermediate subordinate CA. The system administrator can choose cryptographic algorithms and key length according to the security requirements. *SECAS* model supports two widely used asymmetric algorithms: RSA and Elliptic Curve Cryptography (ECC). Although all key lengths are supported, the security administrator needs to be aware that larger key length substantially increases the time required for key generation and exchange. ECC support is sensible since it offers the same level of security with relatively small key sizes. For instance, security level of 224-bit ECC is equivalent to 2048-bit RSA. Besides, smaller key size enables faster authentication.

In *SECAS*, an SDN application takes over the job of creating and managing the certificates and key material needed to protect the communication. The sequence diagram of certificate generation is depicted in Figure 2.1. First, that application generates a key pair and issues a CA certificate by using the generated key pair according to designated security parameters, e.g. the chosen asymmetric algorithm and key length. The CA certificate and key pair can be stored in password-protected Java Key Store (JKS) or Public-Key Cryptography Standards (PKCS) data format. The password for the key stores are dynamically generated according to password generation parameters (password length and character complexity such as lower letters, upper letters, numbers and special characters).

After the CA generation, the network administrator can make certificate request via the user interface or a REST call. There are three certificate types: switch ($\mathcal{C}_s$), controller ($\mathcal{C}_c$) and application ($\mathcal{C}_a$). $\mathcal{C}_s$ and $\mathcal{C}_a$ certificates are used for SB and NB interface, respectively. $\mathcal{C}_c$ certificates are used for SB, EWB and NB interfaces.

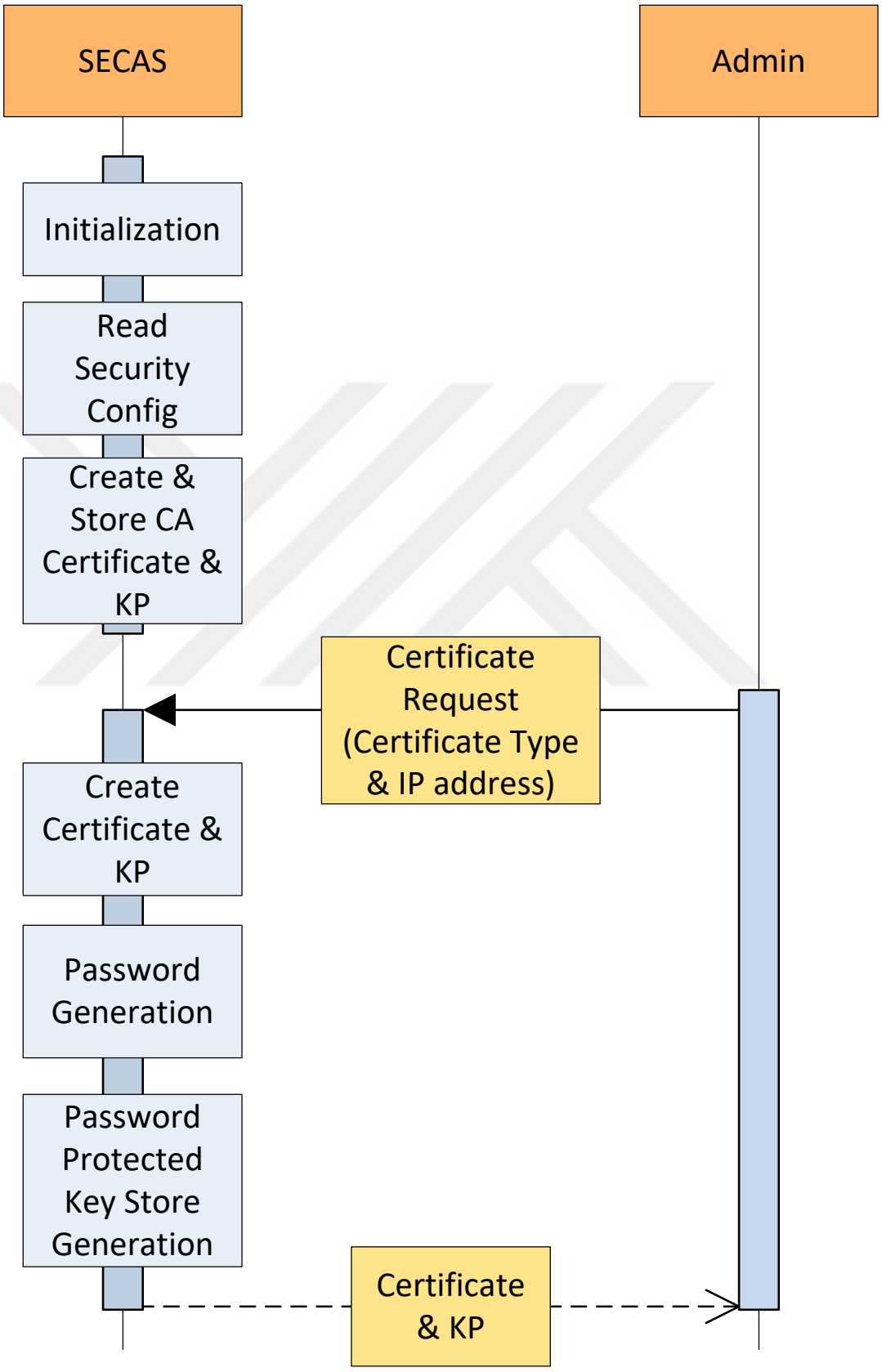


Figure 2.1. Certificate and key pair (KP) generation in SECAS key generation model.

SECAS utilizes only one CA in its key generation model. However, one of the biggest concerns in using a single trusted CA is the private key compromise (key hijacking). Plain password protection of keys is usually ineffective since passwords are written somewhere in the device where keys are also located. Thus, if an attacker can somehow obtain a certificate or key with its corresponding password, he/she can connect to any element in the network. To alleviate this threat, our certificates contain two internal information fields about communicating entity: type and IP address of the entity. In that way, the impact of stealing the private key from a device other than those operating the CA is limited since an attacker can only impersonate one entity from the specific type written in the certificate and can only communicate from a static IP address recorded in the certificate. Accordingly, a certificate request shown during the certificate generation in Figure 2.1 contains two parameters: `certificate_type` and `IP_address`.

When a certificate request is received from the network administrator, the certificate and key pair are generated according to the security requirements and input parameters. Besides, the administrators can make the request to revoke a certain certificate of a device. Accordingly, *SECAS* generates and publishes the Certificate Revocation List (CRL) to announce the devices which should no longer be trusted.

In our model, certificates and key pairs which are requested by the administrator must be transferred to the network devices manually. Developing secure key distribution mechanism is out of scope of this thesis. However, this is an important research topic. For in-depth information, please refer to [38, 39].

2.3.2. SECAS Security Module (*SecMod*)

Certificate generation is not solely adequate for secure communication in a TLS environment. We need to handle mutual authentication with special certificate fields, notify the network administrator about security incidents, harden TLS configuration and comply with the requirements in realistic settings like ACL. Hence, a security

module in the controller is beneficial to secure the communication infrastructure of SDN in an integrated manner. The best location for the security module is the controller since controllers are involved in all communication interfaces. Besides, it is difficult to interfere with switch software or network applications as they are produced by different vendors and a large developer base.

2.3.2.1. TLS Hardening. TLS protocol has some vulnerabilities leading to some significant attacks such as BEAST, CRIME and BREACH [25]. Despite such security incidents, TLS is the de facto standard on the Internet to secure communications between clients and servers, and more than half of the network traffic is protected by TLS. Thus, a proper implementation of TLS is prevailingly accepted as secure [25].

TLS protocol has evolved over years and has different versions deployed across connected devices. It employs different cipher suites which are set of of cryptographic algorithms and used for key exchange, signature generation, encryption and integrity. For secure SDN communication channels, we need to harden TLS configuration by disabling older and vulnerable TLS versions, and removing weak and vulnerable cipher suites.

In SECAS, the network administrator can determine the TLS configuration to deploy in the whole software-defined network. That configuration contains allowed TLS versions and cipher suites in preferred order. Thus, only secure TLS versions and cipher suites are maintained in the controller and TLS attacks like CRIME and BEAST can be prevented.

2.3.2.2. Security Management Functions. In most of the controller software implementations, TLS communication must be activated before the controller is started and it cannot be turned on or off without taking the controller offline. However, this situation is impractical since the controller must be restarted and then preserve all its operational context to enable or disable merely a security feature. Additionally, the

security administrator is required to start or stop TLS for different reasons such as changing security policies. As an example, let us consider that all network applications are running in a compute node near to the controller and TLS is deactivated in NB interface since it is assumed to be physically secure in a data center environment. The administrator needs to activate TLS in NB interface when a remote application is deployed. Nevertheless, intermittent restart of the network controller is not a viable requirement due to changing security policies.

To address this management deficiency, *SecMod* allows activation/deactivation of TLS communication on-the-fly in every interface separately via user controls. Thus, the administrator can manage the network in an agile manner as its requirements are constantly changing.

Table 2.2. TLS support in various commercial switches.

Switch Model	**TLS Support**
HP 3500	No
Brocade MLXe	No
Alcatel-Lucent OmniSwitch	No
Open vSwitch	Yes

2.3.2.3. TLS-less Operation. A network consists of a diverse set of elements (switches and network applications). Particularly, switches can have different vendors and capabilities. Table 2.2 shows TLS support of various switch vendors [40]. Although more capable switches are being released alleviating this deficiency, SDN network elements in the field may still lack TLS support. Thus, *SECAS* also supports network elements whose TLS support is absent via a fall-back scenario. The administrators can mark these elements in the configuration to permit plain-TCP connections to the controller while other elements still connect via TLS. The controller can not be marked since the selected controller software must support TLS anyway. Despite the fact that this is

not a desired and recommended situation, it may occur due to practical constraints in operational networks.

2.3.2.4. Access Control. A distributed SDN control plane can have more than one controller cluster. Controllers have two types of communication through EWB interface: intra-cluster and inter-cluster. In this setting, each controller manages a subset of switches and communicates with only a set of controllers. Hence, switches must connect to the controllers assigned to them and a controller can only connect to allowed peers. To this end, *SecMod* supports ACL for each controller and can further restrict access.

An ACL includes IP addresses of valid switches and controllers which can connect to a controller. When a device sends a connection request to a controller, *SecMod* controls whether the ACL contains the IP address of the device. If it does not, the connection request is rejected by the security module. Hence, no unlisted device can access the controller.

2.3.2.5. Protection against Private Key Hijacking. Private keys can be hijacked by hacking an SDN communicating entity or physically compromising it. As mentioned earlier, two special fields, `certificate_type` and (device) `IP_address`, are added to the certificates to cope with the hijacking issue. *SecMod* overrides the certificate verification to control these two fields. Besides, client authentication is enforced since the default mode of TLS considers only server authentication.

After certificate generation, the client and server already have necessary certificates and key pairs which are used as key-stores to authenticate themselves. They also have the CA certificate to use as trust-store. *SecMod* takes over the certificate verification when there is a connection attempt to the controller. It performs the usual certificate verification using the device's trust-store. *SecMod* also checks the CRL to control whether the certificate is revoked or not. Additionally, the module checks the

certificate type and controls whether IP address of the connecting party is identical with the respective field in the certificate. For instance, if a switch certificate is tried to be used in EWB interface, that connection is rejected because of the certificate type mismatch.

2.3.2.6. Security Monitoring and Situation Awareness. Network administrators operate large networks and they can not control every network element and their states in real time. SDN provides powerful mechanisms to monitor network status such as global network view and flow statistics. *SecMod* leverages the SDN infrastructure to generate real-time alarms for the following security incidents:

- Connection requests from IP addresses not in the ACL.
- The use of invalid or untrusted certificates in the TLS handshake.
- A mismatch between the IP address or the type of the connecting device and the corresponding information in the certificate shown during the TLS connection setup.

These alarms specifically indicate which devices are affected from which security incidents. For further analysis by the administrator(s), they can be seen from a graphical user interface. These alarms can also be used to determine the certificates which need to be revoked and can be seen from a graphical user interface. Moreover, they can be generated to feed a third-party security incident and event tool.

2.3.2.7. How They All Fit Together – SecMod Operation Flow. Figure 2.2 depicts the operation flow of the *SECAS SecMod* when a switch initiates a connection to the controller. The other cases such as controller-to-controller communication are handled in a similar way. When the connection request is received by the controller, *SecMod* checks whether the IP address of the connecting switch is in the ACL. If it is not, the connection is rejected and a security alarm is generated.

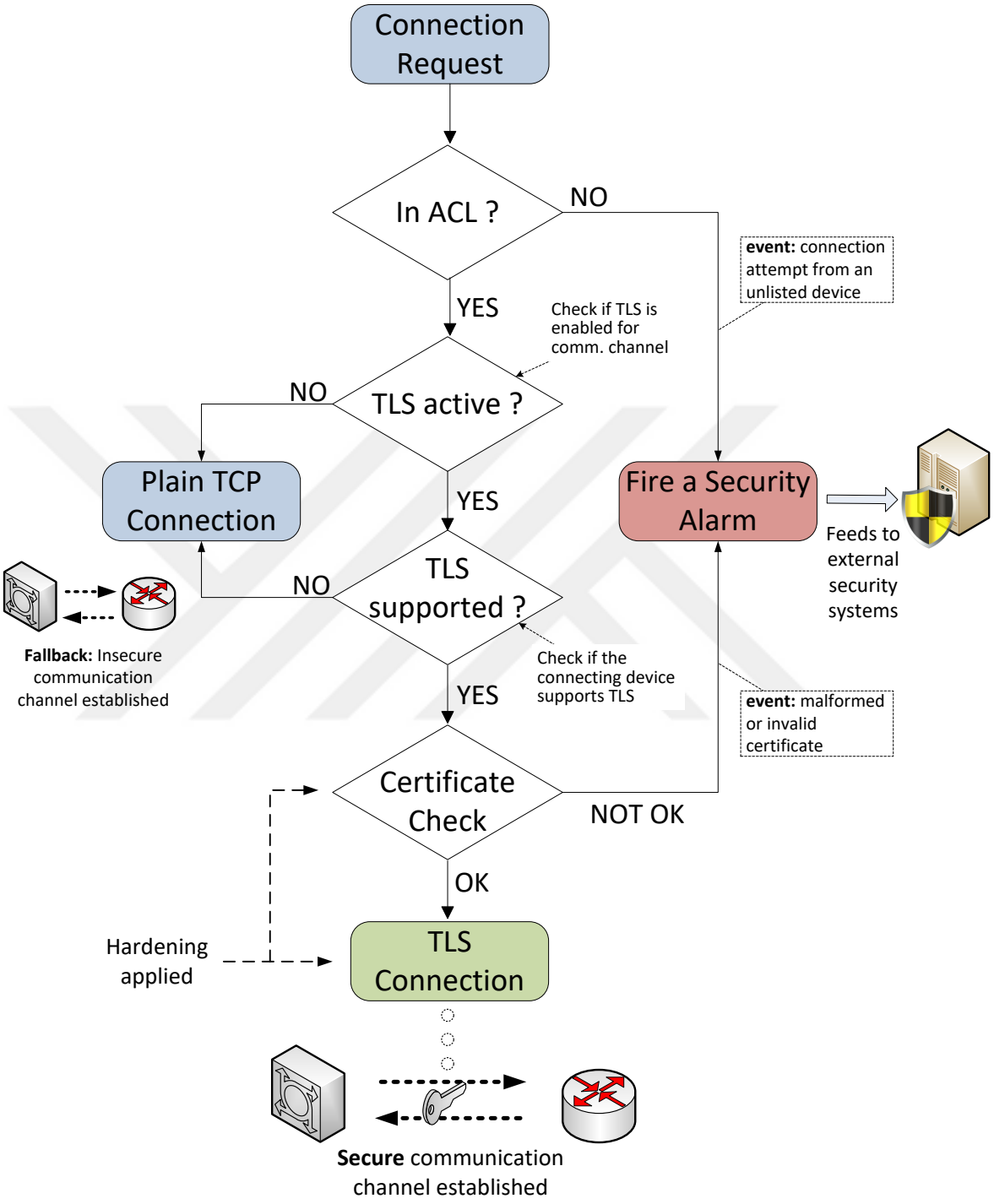


Figure 2.2. Operation flow of *SecMod*.

Otherwise, the module checks if TLS is active and the switch supports TLS. If that is not the case, the connection request is accepted but the controller starts managing the switch via plain OpenFlow. Otherwise, a TLS handshake is started. When the certificate of the switch is received, its verification is started. The certificate must be issued by the CA trusted by the controller. Additionally, its type must be "switch" and the IP address of the connecting switch must be the same IP address embedded in the certificate.

If any of these checks fails, a security alarm is triggered. According to the alarm, the administrator(s) may decide to revoke the certificate of the connecting switch. If all controls are passed, the TLS handshake is completed and communication between the switch and the controller over the TLS channel can start.

2.4. Discussion

In this section, we elaborate on the performance of *SECAS* through a discussion of its overhead and some empirical results due to key generation. Based on this analysis, system administrators can deduce the impact of their security preferences on the network if *SECAS* is employed. Additionally, they can decide on which values for security parameters are to be selected according to their requirements and use-cases.

Persistently, there is a performance concern when applying any security mechanism to a communication channel in data networks. Such concerns are common excuses to disable/ignore security functions based on a cost-benefit perspective. Thus, communication security in SDN using TLS mechanisms has not gained adequate traction. Besides, data centers, touted as leading execution points of SDN implementations, have relatively controlled and restricted environments. Therefore, security hardening and cyberattacks on the communication channels in these domains are regarded as relatively far-fetched issues. These factors have led to the current situation where securing communication channels via TLS is generally overlooked or comes later in the list of security priorities [25].

There are different elements of overhead due to adoption of TLS solutions for SDN security. First, encryption and certificate usage naturally leads to computational costs. However, modern hardware minimizes these costs with perpetual improvements. Processors utilize built-in instruction sets to efficiently handle encryption and mitigate performance issues. Hence, the performance overhead of TLS practically becomes manageable nowadays [26].

SECAS supports RSA and ECC as asymmetrical algorithms along with arbitrary key sizes. To render the related trade-offs of the *SECAS* key generation application in a tangible way, a performance simulation was conducted by generating certificates with different key sizes of RSA and ECC algorithms.

We used BouncyCastle cryptography library and Java 1.8 on a 64-bit Ubuntu machine with 2.5GHz CPU, 8GB RAM. We ran each simulation for 100 times and investigated the average generation time. Table 2.3 shows the simulation results.

The results indicated that ECC has faster certificate generation speed and that generation time does not change significantly as the key size increases. In contrary, as the key size of RSA increases, the required time for certificate generation increases drastically. For instance, NIST allows 2048-bit or longer RSA keys to be used in TLS and *SECAS* key generation application can generate and pack 2048-bit certificate in less than 0.5 second on average while 224 bit ECC which is equivalent to 2048 bit RSA, took only 8 milliseconds. Hence, ECC should be preferred especially for large key sizes.

If a network administrator chooses 3072-bit RSA, it takes two seconds to complete. However, keep in mind that this is an one-time process per device, thus acceptable. Additionally, the required time for certificate generation can be reduced significantly by using a dedicated device such as a hardware security module. In addition to this investigation of key generation performance, as future work, we plan to perform system-level overhead analysis of *SECAS*.

Table 2.3. Certificate generation time in SECAS.

RSA			**ECC**		
Key size	**Mean time (ms)**	σ **(ms)**	**Key size**	**Mean time (ms)**	σ **(ms)**
2048	485.0	330.6	224	8.0	8.6
3072	2,032.0	1,187.0	256	5.0	7.9
7680	64,071.0	40,876.8	384	9.0	8.0
15360	814,798.0	550,120.0	521	21.0	13.4

Another considerable drawback is the certificate and key distribution since it may require manual steps like transferring generated certificates to network elements. However, key distribution is an inherent problem which can be addressed according to different requirements. Efficient mechanisms can be devised based on existing solutions to address different security policies, system attributes and use-cases. Besides, network administrators already deal with such operational challenges in the field and are used to address this kind of deployment tasks thanks to practices such as scripting. Essentially, SDN usage cannot be expanded without securing it by accepting some bearable overhead.

Since key generation must be done before TLS handshake, the key generation application does not work in real time. Thus, failure or saturation of the application do not impose a major risk and can be mitigated using various techniques (e.g. additional hardware resources). Besides, according to our simulation, 2048-bit RSA certificate generation of a large network with 1000 switches, 10 controllers and 100 applications would take less than 10 minutes. This run-time can be shortened by using a dedicated hardware module, deploying multiple instances of the key generation application by cloning the CA key pair on different machines.

From an architectural viewpoint, *SECAS* can be used with intermediate CAs to achieve better scalability. Additionally, if one uses at least two sub-CAs with a root CA where the private key is only offline available, he significantly reduces the

risk of single-point-of-failure and the system might have a recovery path if only one sub-CA certificate is revoked and no secure connection is possible anymore because of this situation. However, the security administrators should keep in mind that each additional layer significantly increases the time required for certificate verification.

2.5. Conclusions

In this section, we first present a concise overview of SDN strengths and weaknesses from the security perspective. There are many works which use SDN as a security enabler. However, the security of SDN is also a crucial topic. Therefore, we propose *SECAS* — a secure communication architecture including a key generation application to eliminate security threats to the communication channels of SDN. *SECAS* provides mutual authentication between SDN entities, end-to-end TLS encryption for all communication interfaces, certificate verification and real-time notifications/alarms about security violations. It also hardens the TLS configurations by removing vulnerable TLS versions and weak cipher suites while applying ACL to further restrict access. *SECAS* considers real-world requirements of SDN such as on-the-fly TLS activation/deactivation and support for TLS-incapable devices. We also present a discussion on how *SECAS* provides secured communication channels in SDN.

As a future research direction, the security module can be further extended to detect malicious network devices and then automatically revoke their certificates. For the key management aspect, a more automated key distribution and regeneration method for SDN could be another direction to fully automatize *SECAS* flow. For the key management aspect, a more automated key distribution and regeneration method for SDN could be another direction to fully automatize *SECAS* flow. Besides, overhead of TLS to the SDN system in terms of bandwidth, latency and throughput with respect to different security parameters (e.g., key length and deployed algorithms) can be investigated. Accordingly, particular impact of TLS to the network operation according to pertinent security requirements can be explicitly conveyed.

3. NETWORK FINGERPRINTING VIA TIMING ATTACKS AND DEFENSE IN SOFTWARE DEFINED NETWORKS

3.1. Introduction

SDN is becoming a native networking model for next generation networks. However, with its decoupled architecture, SDN is susceptible to reconnaissance through time inference attacks. Attackers can use probing based measurements and gather information such as network type and flow table size. In this section, an automated attacker tool called *RAFA* is proposed to infer network type (SDN or traditional) and flow rule timeout values (hard and idle). Moreover, a lightweight defense mechanism to randomize rule timeouts with respect to network status is described. A comprehensive simulation setup with different network conditions shows that the proposed methods achieve superior success rate in diverse settings.

In SDN, since switches are relatively dumb devices, they need to communicate with the controller to process a packet that does not match any flow rule. This communication mechanism can be used by the attacker in various ways. The attacker can generate traffic that exhausts the network resources like switch/controller CPU or the bandwidth between switch and controller which makes SDNs vulnerable to (distributed) denial-of-service ((D)DoS) attacks in all planes (data, control and management). One prominent example of such an attack is an attack on an SDN switch that saturates its flow table by creating traffic causing many new flow table entries. Furthermore, if the attacker can also extract the TCAM (Ternary Content-Addressable Memory) size of a switch, they can adapt the flow generation rate to effectively choke the switch with a minimal number of packets. Such restrained packet generation is beneficial on the attacker side for both performance reasons and the reduced risk of being detected before the attack succeeds.

To understand whether a network is susceptible to such attacks, an attacker must determine whether it is an SDN and learn about relevant properties. One way determine whether a network is a software-defined network is to exploit First Packet Processing Time (FPPT) which is defined as the Round-Trip Time (RTT) deviation of the first and second packets of a new unmatched flow. Such techniques are called SDN fingerprinting and are usually followed up by techniques to determine relevant SDN properties such as flow rule timeout values, flow match fields, active flow rules and flow table capacity [14–18, 20, 41–45].

With this information, attackers can then move from the reconnaissance phase to the next phases of the cyber kill chain and conduct attacks that, in the worst case, can lead to the take-down of the entire network. However, it is important to note that SDN fingerprinting research is still at an early stage and focuses mostly on the attacker side, namely on how to extract more and more reliable information from the networks. How to prevent the attacker from obtaining such information in the first place is rarely looked into. In this section, both attack and defense methods are proposed and their performance is investigated by using simulation-based experiments. The main contributions of our work can be summarized as follows:

- A fully automated, robust and reliable SDN fingerprinting module, *Robust and Automated Fingerprinting Agent for SDN (RAFA)*, to infer network type (SDN or traditional), and flow timeout values (hard and idle). We show that the proposed methods are of high accuracy for different network conditions via extensive simulations.
- A lightweight and dynamic defense mechanism to increase complexity and costs for attackers by adjusting flow timeout values according to network status.

This chapter is structured as follows: First, we present some SDN fundamentals relevant for this chapter in Section 3.2 and continue with a discussion of related work in Section 3.3. This is followed by a presentation of *RAFA* in Section 3.4 and our defense method, *DAFTR*, in Section 3.5. We then use Section 3.6 to describe how we

conducted our extensive simulations and to document the results with focus on the performance of *RAFA* and our defense method. The limitations and relevant open questions are discussed in Section 3.7 to provide a critical view on what we achieved. Finally, Section 3.8 discusses future research directions and concludes the chapter.

3.2. SDN Fundamentals

Figure 3.1 illustrates the packet processing logic within SDN. It delineates three distinct paths for a packet, demarcated by dashed rectangular regions in the diagram. Upon packet arrival in the data plane, the header information undergoes checks against the match fields of pre-existing flow rule entries. When a match is found, the action field of the corresponding entry is executed (*Fast Path*). In cases where no match is identified, the data plane triggers a `packet-in` message, which is subsequently processed by the controller. The controller then establishes a new flow rule to manage this novel flow (*Slow Path*). However, the switch must ascertain whether adequate space remains within the flow table for this new entry. In situations where the flow table is at full capacity, the controller or switch is compelled to undertake flow table replacement operations. This involves the removal of certain flow rules according to the table replacement algorithm, potentially leading to further performance degradation across the network (*Slowest Path*).

An essential aspect of the processing logic involves comprehending how and when flow rules are purged when the flow table is not at full capacity. OpenFlow [46], having established itself as the primary southbound protocol, incorporates two fundamental mechanisms for flow rule deletion in the majority of SDN environments: deletion orchestrated by the controller and deletion predicated on timeouts. OpenFlow also defines two distinct timeout values: hard and idle [46]. In the context of idle timeouts, flow rules are removed following a designated period of inactivity. Conversely, hard timeouts dictate the removal of flow rules after a specified period, irrespective of their activity status. These timeout values hold critical significance for attackers, as they can directly impact their fingerprinting efforts and the performance of their attacks.

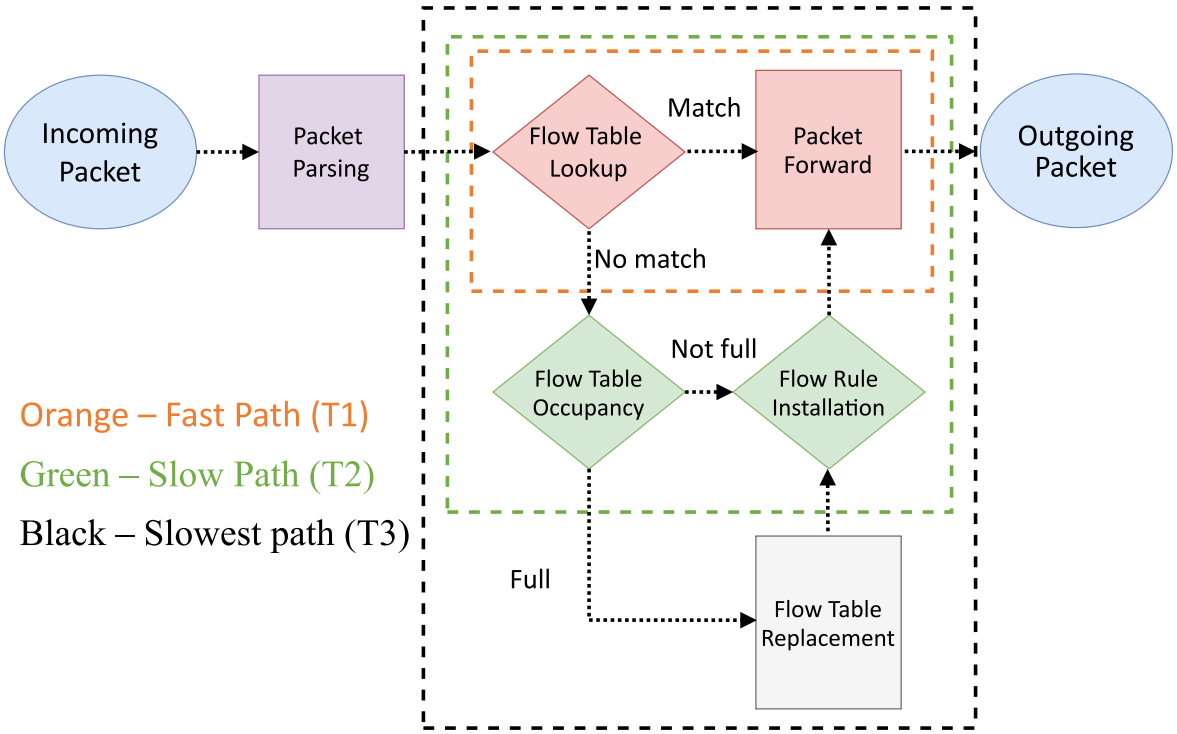


Figure 3.1. SDN packet processing flowchart.

Two significant consequences can be deducted from the packet processing and flow rule logic described above. Firstly, an attacker can deduce valuable insights into the switch's state by comparing the RTT values of packets. This is due to the fact that each path taken by a packet involves distinct interactions and iterations. By analyzing the path information of a packet, which is contingent on flow rule timeouts, the attacker can gather data concerning the state and size of the flow table, the timeout configurations in use and even details about the flow rules themselves, including the match fields. Secondly, armed with this acquired information, an attacker could conduct more efficient and tailored attacks, exploiting these insights to optimize their malicious activities.

3.3. Related Work

Fingerprinting encompasses the process of identifying and gathering information regarding a specific computer system or network, with the aim of discerning its unique characteristics and potential vulnerabilities. This involves the collection of various

data points, including details on open ports, the operating system in use, software applications, network topology and configuration settings [47]. The information acquired through fingerprinting can serve multiple purposes, ranging from launching targeted attacks to enhancing security measures designed to safeguard against potential threats [48]. There are lots of works in different areas such as radio frequency, indoor localization, browser and TLS parameter fingerprinting [49–52].

In this section, we present the literature on SDN fingerprinting and on methods to make fingerprinting more difficult. Table 3.1 shows some related technical works summarizing their main methods and inferred SDN information via fingerprinting.

[14] is the first theoretical work discussing the possibility of SDN fingerprinting. The authors proposed that flow rule timeout randomization can be used as a defense mechanism. [15] is the first practical paper which uses fingerprinting via analysis of RTT differences between packets.

In [16], the impact of hop count and bandwidth on RTT fingerprinting accuracy is investigated. The authors use the student t-test and found that fingerprinting accuracy is only marginally affected by hop count and data link bandwidth. As a defense against RTT based fingerprinting, they propose to add a delay to a few packets of new inactive flows. A flow is deemed inactive if no packets associated with it have been received by the switch within a predetermined period of time. However, dumb switches cannot handle this type of operations and it requires some modifications in switches and OpenFlow.

In [17], several challenges faced when doing SDN fingerprinting in real-world settings like unknown background traffic, rule granularity and rule overlapping are considered. The authors found that if there is a considerable amount of traffic besides the fingerprinting traffic, this greatly affects the fingerprinting accuracy.

Table 3.1. Related literature on SDN fingerprinting.

Work	**Mechanism and Fingerprint Info**
[14]	- First theoretic work on SDN fingerprinting. - Flow timeout randomization for defense.
[15]	- SDN identification via time inference using t-test.
[16]	- Infer active flows. - Delay a few packets of inactive flows according to a fixed distribution as defense.
[17]	- Show the challenges of SDN fingerprinting. - Background traffic, rule granularity and rule overlapping are considered. - Proposed solutions: flow timeout randomization, proactive rules and TCP proxy.
[20]	- Infer flow table capacity and flow table usage. - Determine flow replacement algorithm (LRU or FIFO).
[41]	- Fingerprint OpenFlow Controllers via default parameters and packet header information.
[18]	- Infer SDN policy parameters such as hard and idle timeouts, match-fields, controller reaction at `table full` event.
[42]	- Add delay to the packets of every flow with decreasing probability.
[43, 53]	- Send timing probes and test streams together to infer match fields of flow rules and active flows. - Proposed defense is timeout proxy in the switch.
[44]	- Infer `match field` of flow rules and controller type.
[45]	- Infer `match fields` and from that information infer controller type. - For defense, `packet-out` usage is considered.

In [17], the authors proposed two defense methods: flow timeout randomization and traffic-aware mitigation. The latter suggests to use proactive rules for ICMP protocol and rate limiting for UDP protocol to reduce the attack surface. These methods cannot prevent SDN fingerprinting although they can degrade network performance drastically. As a third method, the authors discussed a TCP proxy for defense. This method depends on additional equipment since a simple SDN switch cannot implement this kind of proxy.

In [20], by sending an initial packet, a new flow is installed and the corresponding RTT (T_1) is calculated. Afterwards, a second packet is sent and its RTT (T_2) is calculated. After that, packets are sent that are designed to lead to the installation of a new flow for each of them until the RTT for such a packet is significantly bigger than T_1 because the flow table is now full and the slowest path becomes relevant. This RTT is denoted as T_3. By using T_1, T_2 and T_3, they try to find the size of the flow table and the number of present flow rules in the flow table. Besides, they use measurements to infer which of the two flow table replacement algorithms LRU or FIFO is used.

[41] aims to determine the controller type (ONOS [54], RYU and so on) by using packet header information like Link Layer Discovery Protocol (LLDP) messages. Furthermore, they aim to infer hard and idle timeout values and compare these values with the default values of the controller software to make an assumption about its type.

In [18], the authors aim to find a diverse set of SDN policy parameters such as hard/idle timeouts, match-fields of the flow rules, and controller reaction at table_full event (do not react or delete flow rules). Furthermore, they exploit the fact that fat-tree topology is (often) used in SDN deployed data centers, they can infer some information (in-rack, in-pod and inter pod groups) about the topology.

In [42], a defense mechanism against a smart attacker grasping that delays are added to the second packet of a flow only to throw off traditional fingerprinting attempts is proposed. Their solution applies delays to the packets with decreasing prob-

ability as the number of packets in the flow increases. The authors also proposed that the controller type should be changed in some interval to achieve substantial uncertainty for the network since different types of controllers can have different network parameters such as flow timeout values and match fields. The interval of controller change is calculated according to some network parameters such as the number of switches in the target network and the number of users connected to a single switch. Both probabilistic delay addition and controller type change can have huge impact on the network, leading to unacceptable performance penalty. Since the switches cannot handle random delay addition to the flows, the controller must add the delay which requires lots of packet-in usage between switch and controller. Besides, when the controller type changes, all the flow rule information in the controller is reset since the new controller does not have it.

To calculate timeout values, incremental packet interval can be used. As the interval hits a timeout, the flow of the packet is removed and a surge in RTT is measured. However, hard timeout should be higher than idle timeout [18]. Therefore, hard timeout should be found first since while increasing the intervals to infer idle timeout, hard timeout can be triggered if the sum of the time interval exceeds the hard timeout value.

Another issue about timeout calculation is how to determine the RTT surge. In that regard, statistical methods should be used to determine the threshold which will be used to control when the flow is expired. Besides, sometimes a RTT surge can be caused by unfavorable network conditions. Therefore, timeout fingerprinting should be executed a number of times to eliminate network-based errors. Also, RTT should be considered while calculating the timeout since if RTT is large enough, it can greatly affect accuracy of the calculation.

Timeout fingerprinting is investigated in [17, 18, 20, 41, 42]. [20, 42] do not provide an algorithm for timeout measurement and mainly present their procedure of the measurement. [17, 18, 20, 41, 42] try to find idle timeout first which can cause erroneous

measurements as stated before. Besides, it is not clear whether their simulations are executed when both timeouts are set or it is set separately. Additionally, statistical methods are not employed in threshold calculation in [17,20,41,42]. All these works do not consider RTT while calculating timeouts and they do not cover networks with high end-to-end latency in their experiments. Furthermore, network-based errors are not taken into account in those works. As a consequence of these deficiencies or missing features, there is room for improvement for timeout fingerprinting works.

In [43] and [53], two types of flows (timing probes and test streams) are sent together. Timing probes are specially crafted to be processed in the controller like ARP messages. In SDN, MAC learning is often implemented in the control plane. If an attacker sends a spoofed ARP request, the request and/or reply will be routed through the control plane. Hence the RTTs of ARP request/reply pairs can be used to measure the control plane timings. Same is true for OpenFlow Echo and Hello messages. SDN can also be configured to forward at the IP level. This allows an adversary to use IP packets with low TTL values as timing probes. OpenFlow switch cannot generate ICMP packets and must send packets with expiring TTL values to the controller. The controller will generate the appropriate ICMP response and send it back.

Test streams are used as new flows with different header fields. Therefore, if timing probes take longer while a test stream is sent, the newly changed header field is considered as a match field for the flow rules. In terms of defensive mechanisms against fingerprinting, they proposed the usage of a timeout proxy. The authors suggested that a proxy in the switch installs a default rule for a new flow if the controller does not install a rule for that flow in a predetermined time. The main drawback of this method is that with current switches, the implementation of such a proxy is not possible without architectural changes to the switch. Also the communication overhead between switch and the controller plays a huge role in the fingerprinting mechanism anyway since the predetermined time should include this overhead.

In [44], match fields of flow rules are found by changing packet header information and comparing RTT deviation. According to match fields, the controller type is guessed with the default match field of the controller software. In [45], similar to [44], match fields are used to infer the controller type. Besides, the authors suggested usage of `packet-out` for defense. `Packet-out` message is used to inform switches what to do with the packet without installing a new rule. Hence, packets of new flows experience similar RTTs as existing flows. This defense action increases both switch and controller load in terms of CPU, memory and bandwidth since the number of `packet-in` messages multiplies.

In summary, the review of the existing literature revealed that there are numerous approaches to do SDN fingerprinting, but they are not very robust and usually only work reliably under laboratory conditions. Furthermore, we found that in the area of anti-fingerprinting techniques there is a lack of solutions without significant impact on SDN architecture or network performance. Hence, to improve on the attacker side, more robust approaches are definitely advantageous. And on the defender side, we believe that a more lightweight defense approach is crucial to support a more widespread adoption of anti-fingerprinting techniques.

3.4. Robust and Automated Fingerprinting Agent for SDN (*RAFA*)

In this section, we propose *RAFA* to infer the network type (traditional or SDN) and flow rule timeouts (hard and idle). *RAFA* is more robust and provides better accuracy than previous methods, especially under different and difficult network conditions. To achieve this, we make use of statistical methods adapted for such conditions. *RAFA* is denoted as an automated agent since once the attacker decides on the targeted network, the fingerprinting operates fully automated. First, it determines the network type. If it is SDN, it starts timeout fingerprinting and infers the hard timeout. Next, the fingerprinting of the idle timeout is started. Idle timeout fingerprinting makes use of the hard timeout and RTT threshold, $T_{\text{threshold}}$, which is determined while measuring hard timeout.

RAFA relies on three key assumptions. First, the attacker possesses knowledge of at least one flow match-field. This information can be obtained through SDN fingerprinting, as demonstrated in [23], but it is beyond the scope of this thesis. Second, the attacker utilizes a host capable of sending packets to a server in the target network. Lastly, there exists a single SDN network on the path to the target network. The attacker sends packets to measure RTTs and infers information based on RTT deviations. To better cope with network jitter, the attacker sends two packets almost consecutively in back-to-back mode. Increased intervals between packets could result in significant network jitter affecting the results.

In our threat model, we assume an adversary who has access to one or multiple hosts to send packets to a server within the targeted network without needing to compromise any other devices on the network, including switches, controllers, and applications. We assume that a defense mechanism, as mentioned in Chapter 2, is deployed. Consequently, the attacker cannot block, save, alter, drop, or replay any communications within the SDN. Additionally, the attacker is incapable of deploying fake network entities to initiate a Sybil attack.

3.4.1. Network Type Fingerprinting

To understand whether a network is an SDN, the first two packets of a new flow are sent to a server in the target network, and the first RTT (T_1) and second RTT (T_2) values are measured. To make this statistically meaningful, this measurement has to be carried out multiple times. These measurements can be done in parallel or sequentially. Nonetheless, the parallel execution can increase the risk of detection for the attacker.

After a group of T_1 and T_2 measurements are collected, we use the student t-test [55] to determine whether these two groups have the same mean. If they are, the network is traditional.

Table 3.2. Parameters for timeout fingerprinting algorithms.

Parameters	**Explanation**
iterationCount (τ)	Number of times the timeout measurement is run.
$T_{\textbf{step}}$	Sleep interval time (in seconds) which can be between 0 and 1.
Majority coefficient (ρ)	Percentage of the majority where the frequency of a timeout value must exceed it for a successful fingerprinting.
$T_{\textbf{sleep}}$	The time between probing packets used for the idle timeout fingerprinting (in seconds).

The t-test is used when two independent groups are compared. It simply checks whether two sets are significantly different from each other for a given confidence interval. The test just needs sample mean and sample standard deviation values of the groups, and accordingly rejects or confirms the hypothesis that the two groups have the same (unknown) mean. The t-test can be two-tailed and one-tailed. Two-tailed one just checks whether the means are equivalent while one-tailed or one-sided also considers whether one mean is larger or smaller than the other. The confidence interval shows the statistical significance of the experiments. For instance, an α-value of 0.01 indicates that there is only a 1% probability where the results from an experiment happened by chance. We use one-tailed t-test since typically T_2 should not be bigger than T_1.

3.4.2. Timeout Fingerprinting

OpenFlow defines two timeouts for flow rules: hard and idle. According to the OpenFlow specification, timeouts can be assigned to any values between 0 to 65535 seconds [46]. The value 0 means that timeout is not set hence the flow rules will not be deleted unless the controller wants to.

First, hard timeout is searched in *RAFA*. Hence we can use it in idle timeout measurement. Also sample mean and variance of RTT values are utilized to determine

threshold which is used for flow deletion control. *RAFA* executes timeout fingerprinting multiple times to get rid of variance which can stem from network fluctuations. Besides, RTT is integrated into timeout measurement considering that it can be long enough to interfere.

After fingerprinting the network type, if a SDN is inferred, *RAFA* automatically starts to measure flow rule timeouts. The parameters used in the timeout fingerprinting algorithms are shown in Table 3.2. To eliminate inaccuracies caused by delay variance due to the network conditions, we measure the timeout values `iterationCount (`τ`)` times. `Majority Coefficient (`ρ`)` determines the required percentage of a timeout value for fingerprinting. It should be at least 0.5 or bigger to achieve majority. For instance, if the τ and ρ values are 5 and 0.5 respectively, the same timeout value must be found at least 3 times (precisely stated, it should be bigger than $5 * 0.5 = 2.5$).

Interval step times (T_{step}) for hard and idle timeout fingerprinting are 0.5 and 0.3, respectively. T_{step} can be at most 1. The frequency $1/T_{\text{step}}$ shows the number of times which a timeout value is checked. Hence as the step becomes smaller, the attacker needs to send more packets and invest more time for fingerprinting. However, if the step becomes too large, the accuracy of the fingerprinting is likely to suffer, especially if the delay is also high. Therefore, there is an implicit trade-off between accuracy and overhead for the attacker.

T_{sleep} is only used in the idle timeout algorithm and captures the duration between consecutive packets. It is basically a sleep time where sleeping means that nothing is done. Furthermore, in the same algorithm, T_{step} is used to increase T_{sleep} so that it tests for longer timeouts.

After the network is inferred as SDN, *RAFA* starts hard timeout calculation. For hard timeout calculation, we send packets with fixed intervals since the hard timeout does not depend on packet arrivals. When the RTT of a packet is significantly higher than the average, we conclude that the flow associated with the packets is deleted. The

hard timeout calculation algorithm is shown in Figure 3.2.

Data: iterationCount (τ)
Result: T_{hard}

1 Get response times and calculate RTT_{mean} and RTT_{std}
2 Calculate $T_{\text{threshold}} = RTT_{\text{mean}} + 3 * RTT_{\text{std}}$
3 $T_{\text{step}} \leftarrow 0.5$
4 $T_{\text{hardArray}} \leftarrow []$
5 $counter \leftarrow 0$
6 **while** $counter < \tau$ **do**
7 Send first packet
8 $T_0 \leftarrow currentTime$
9 Sleep for T_{step}
10 Send packet and measure RTT
11 **if** $RTT < T_{threshold}$ **then**
12 go to 9
13 **else**
14 $T_1 \leftarrow currentTime$
15 $T_{\text{hard}} \leftarrow T_1 - T_0 - RTT$
16 $T_{\text{hardArray}}$.append(T_{hard})
17 Find the most common T_{hard}, $T_{\text{hardMostCommon}}$, in $T_{\text{hardArray}}$
18 Find the number of occurrence of $T_{\text{hardMostCommon}}$, $T_{\text{hardOccurence}}$
19 **if** $T_{hardOccurence} > \rho * \tau$ **then**
20 return $T_{\text{hardMostCommon}}$
21 **else**
22 return 0

Figure 3.2. Hard timeout calculation algorithm.

To understand a significant RTT surge, we need a threshold. Hence initially we calculate the sample mean (RTT_{mean}) and sample standard deviation (RTT_{std}) of RTT values where these RTT values are from the second packet of new flows (Line 1). Therefore, we do not consider FPPT while calculating the threshold. To find a

statistically valid threshold, $T_{\text{threshold}} = RTT_{\text{mean}} + 3 * RTT_{\text{std}}$ formula is used (Line 2). Thus, 99.7% of the data falls within three sample standard deviations of the mean. We set sleep interval as 0.5 (Line 3).

To eliminate network based errors which can occur with delay deviation due to changing network conditions, we calculate the hard timeout value τ times. Therefore, we initialize an empty hard timeout value array (Line 4) and set `counter` to zero (Line 5).

Then first packet is sent (Line 7) and current time is set to T_0 (Line 8). Then the process sleeps for T_{step} second(s) (Line 9). In Line 10, another packet is sent and the new RTT is measured. If that value is less than the threshold, we go back to Line 9. If it is bigger, then we subtract T_0 and RTT from the current time (Line 15).

RTT is subtracted since the timer started when the first packet was sent and it can be long enough to affect the timeout value. However, it approximately takes $RTT/2$ for the first packet to arrive at the switch. After that, the hard timeout counter is started. Further, the flow rule is deleted after the last packet but again we see it approximately $RTT/2$ time later. Therefore, the RTT is subtracted.

The hard timeout is found and added to the hard timeout array (Line 16). Hard timeouts are searched as before, until the array is full. When the search is done, the most common hard timeout value (Line 17) and its occurrence count (Line 18) are found. If the occurrence count is bigger than $\rho \times \tau$, the most common hard timeout value is returned. If not, 0 is returned. The value 0 means that the hard timeout is not set or is not deducible.

After the calculation of the hard timeout, *RAFA* automatically starts to measure the idle timeout. To do so, we send packets with gradually increasing intervals. The idle timeout calculation algorithm is shown in Figure 3.3.

Data: iterationCount (τ), T_{hard}, $T_{threshold}$

Result: T_{idle}

1 **if** $T_{hard} = 0$ **then**
2 $\quad T_{hard} \leftarrow 65536$
3 $T_{idleArray} \leftarrow []$ and $counter \leftarrow 0$
4 **while** $counter < \tau$ **do**
5 $\quad totalSleepTime \leftarrow 0$, $T_{sleep} \leftarrow 1$ and $T_{step} \leftarrow 0.3$
6 $\quad$ Send first packet
7 $\quad T_{sleep} \leftarrow T_{sleep} + T_{step}$
8 $\quad totalSleepTime \leftarrow totalSleepTime + T_{sleep}$
9 $\quad$ Sleep for T_{sleep}
10 $\quad$ Send packet and measure RTT
11 $\quad$ **if** $RTT < T_{threshold}$ **then**
12 $\quad\quad$ go to 7
13 $\quad$ **else**
14 $\quad\quad$ **if** $T_{sleep} \geq T_{hard}$ **then**
15 $\quad\quad\quad$ $counter \leftarrow counter + 1$ and **break**
16 $\quad\quad$ **if** $totalSleepTime \geq T_{hard}$ **then**
17 $\quad\quad\quad$ $totalSleepTime \leftarrow 0$ and $T_{sleep} \leftarrow T_{sleep} - T_{step}$
18 $\quad\quad\quad$ go to 6
19 $\quad\quad$ $T_{idle} \leftarrow T_{sleep} + RTT$ and $counter \leftarrow counter + 1$
20 $\quad\quad$ $T_{idleArray}$.append(T_{idle})
21 Find the most common T_{idle}, $T_{idleMostCommon}$, in $T_{idleArray}$
22 Find the number of occurrence of $T_{idleMostCommon}$, $T_{idleOccurence}$
23 **if** $T_{idleOccurence} > \rho * \tau$ **then**
24 $\quad$ return $T_{idleMostCommon}$
25 **else**
26 $\quad$ return 0

Figure 3.3. Idle timeout calculation algorithm.

First, we check the hard timeout (Line 1). If the hard timeout equals 0, we set the hard timeout as 65536 which is the maximum value for the flow rule timeouts (Line 2). Then we initialize an empty idle timeout value array (Line 3) and set `counter` to zero (Line 4). Then we set the total sleep time to zero (Line 6). Total sleep time is considered since while calculating the idle timeout, the hard timeout can be activated.

The initial sleep time T_{sleep} and the step T_{step} are set to 1 and 0.3, respectively (Line 7). After that, the first packet is sent (Line 8). T_{sleep} is incremented with T_{step} to achieve gradually increasing sleep time (Line 9). It is added to the total sleep time (Line 10) and then we wait for T_{sleep} seconds (Line 11). In Line 12, another packet is sent and RTT is measured. If the RTT is less than the threshold, we go back to Line 9. Hence we can increase T_{sleep} to test longer timeouts.

If the RTT is bigger, we check whether T_{sleep} is bigger than the hard timeout (Line 16). If it is, we increment the counter (Line 17) and go for another idle timeout calculation (Line 18). We check the sleep interval against the hard timeout since the idle timeout cannot be bigger. After that, we check whether the total sleep time is bigger than the hard timeout (Line 19). If it is, we subtract T_{step} from T_{sleep} (Line 20) and go to Line 8 (Line 21). Total sleep time can be bigger than the hard timeout since it is cumulative. As an example, consider that the hard timeout and idle timeout are 5 and 4, respectively. Also, the initial sleep interval is 1 and the interval step is 0.5. If we calculate $1.5 + 2.0 + 2.5 = 6$, it is bigger than 5, hence the hard timeout deletes the flow when T_{sleep} equals to 2.5. Hence the RTT becomes bigger than the threshold because of this hard timeout induced deletion. Thus we need to try again for 2.5. Therefore we subtract T_{step} from T_{sleep}.

If all checks were fine, we found the idle timeout to be the sleep time plus the RTT (Line 22). The RTT is added since it can be long enough to affect the timeout value. It is added since T_{sleep} only covers the sleep time. However, it approximately took $RTT/2$ to the arrival of the sent packet to the attacker and before that the idle timeout counter is started. Furthermore, the flow rule is deleted after the last packet

but again we see it approximately $RTT/2$ time later. Therefore, the RTT is added.

The idle timeout is found and added to the idle timeout array (Line 23). The measurement of the idle timeout continues as before, until the array is full. When the search is done, the most common idle timeout value (Line 24) and its occurrence count (Line 25) are determined. If the occurrence count is bigger than $\rho * \tau$, the most common idle timeout value is returned. If not, zero is returned. Similar to the hard timeout, 0 means that the idle timeout is not set or is not deducible.

3.5. Lightweight Defense against SDN Timeout Fingerprinting

Timeout fingerprinting is essential to perform efficient DDoS attacks against an SDN controller. By inferring flow rule timeouts, the attacker can maintain flow rules in the flow table with minimal effort. It is also crucial to infer flow table information like flow table usage and size.

The idea of timeout randomization to prevent timeout fingerprinting has been explored in prior research, notably in [14, 17]. [14] provides theoretical recommendations for the randomization of flow timeouts but does not offer any implementation or practical insights. In [17], fixed flow time randomization is used when the network address is not known or trusted without considering network status. It is a challenging task to know which hosts are trustable. Nevertheless, the authors do not explain any method how to obtain this trust information. Moreover, when hosts are external, it can be impossible to determine credibility of the hosts thus, utilizing the proposed approach can be problematic against outsider attacks.

Our method, on the other hand, utilizes network statistics while applying randomization to the flow. As a result, attackers encounter greater levels of randomness, while the impact of randomization on legitimate hosts remains minimal. With this method, success likelihood of an attacker becomes lower and lower as the attack continues.

The works [14,16,17,42,45,53] proposed defense mechanism against fingerprinting. [14] is a theoretical work and the authors just give recommendations about flow timeout randomization. [16] suggests adding delay a few packets of the flows which cannot be done without modifications to the SDN .

[17] suggested flow timeout randomization, proactive rules, rate limiting and TCP proxy. Proactive rules and rate limiting can reduce attacker success chance, however they can degrade the performance of the network. TCP proxy also requires modifications in SDN. In [17], flow time randomization is used when network address is not known or trusted without considering network status. [42] suggested adding delay to the packet of all flows with probabilistic scrambling. [53] suggested a proxy in the switches to install default rules when the controller does not respond within a time frame. Again [42] and [53] require some changes on SDN components. [45] suggested `packet-out` usage for the ports whose source address is changed. Fingerprinting can be done without spoofing by using, for instance, ARP messages.

To prevent timeout fingerprinting, a lightweight randomization mechanism is suggested to make the network more complex and uncertain in this work. By this way, attackers need to invest more for fingerprinting and the network becomes harder to explore. Network parameter randomization is a well-researched topic under Moving Target Defense (MTD) domain [47, 56]. With randomization, timeout values become a moving target. Thus attackers have difficulties to predict operations of the network, and need to send more packets and spend more time. These additional packets and time can also increase the chance of detection by the defender.

Our defense proposal, *Dynamic Adaptive Flow Timeout Randomization (DAFTR)*, in this work is a lightweight defense mechanism since it can be implemented without any changes in the SDN components or protocols. Since it randomizes the flow rule timeout according to basic statistics, its effect on performance is negligible. The defense parameters are shown in Table 3.3.

Table 3.3. Parameters for timeout defense mechanism.

Parameters	**Explanation**
$\mathbf{PacketIn_{Count}}$	`Packet-in` count.
$\mathbf{FRM_{Count}}$	Flow removal messages count.
$\mathbf{PC_{threshold}}$	Packet count threshold for the flow removal messages.
$\mathbf{FD_{threshold}}$	Flow duration threshold for the flow removal messages (in seconds).
$\mathbf{Ratio_{threshold}}$	Ratio threshold between the `packet-in` and the `flow removed` messages.
Randomization Coefficient (β)	Randomization coefficient for the flow rule timeouts.

In our mechanism, initially, we randomize the timeouts of all flows with a randomization coefficient, β. Then a time window is started. In each time window, the number of `packet-in`, $\text{PacketIn}_{\text{Count}}$, and `flow removed` messages, $\text{FRM}_{\text{Count}}$, for each switch port is calculated. `Flow removed` messages contain flow information such as duration, packet count, and delete reason (hard/idle timeout, controller delete, etc.). Therefore, while counting `flow removed` messages, we only consider the flows whose duration or packet count are below a respective threshold. Hence, the logic for counting `flow removed` messages can be expressed as

$$duration < FD_{\text{threshold}} \wedge packetCount < PC_{\text{threshold}}. \tag{3.1}$$

Equation 3.1 shows the counting of `flow removed` messages according to the packet count, $\text{PC}_{\text{threshold}}$, and duration, $\text{FD}_{\text{threshold}}$, thresholds. Both `packet-in` and `flow removed` messages are considered because otherwise a user with a high number of flows can be marked as an attacker. Furthermore, there are packet count and duration thresholds to exclude, for instance, a server which processes numerous flows hence both `packet-in` and `flow removed` messages can be high. In this way, we aim to exclude normal flows by considering only short and mice flows. When the time window is completed, the ratio between $\text{PacketIn}_{\text{Count}}$ and $\text{FRM}_{\text{Count}}$ is checked. The adjustment

in the randomization coefficient can be formulated as

$$(PacketIn_{\text{Count}}/FRM_{\text{Count}}) < Ratio_{\text{threshold}}. \tag{3.2}$$

Hence if the ratio is above a predetermined threshold for a port, we will increase the randomization coefficient for that port's flows. To reduce the coefficient, that ratio should be below the threshold for three consecutive periods. The value of both increment and decrement step size is β. For instance, if β is 0.1 and the ratio is also above the threshold for a port in a switch, the new threshold becomes 0.2 for that port. If this happens again, it then becomes 0.3.

3.6. Simulations and Performance Evaluation

The topology used in the experiments is shown in Figure 3.4. Our topology includes two hosts (a traffic generator and a server), a controller, and one or more switch according to the simulation configuration. To emulate hosts and switches, Mininet [57] is chosen since it can allow to flexibly configure link parameters such as bandwidth and delay. RYU [58] is used as the SDN controller. Our proposed algorithms are implemented in Python. *RAFA* uses `Impacket` [59] Python library for packet generation and capture. For RTT measurement, most common network tool, `ping`, was used. Simulations are conducted on a computer with Intel i7-6500U 2.5 GHz (2 cores) processor and 16GB RAM.

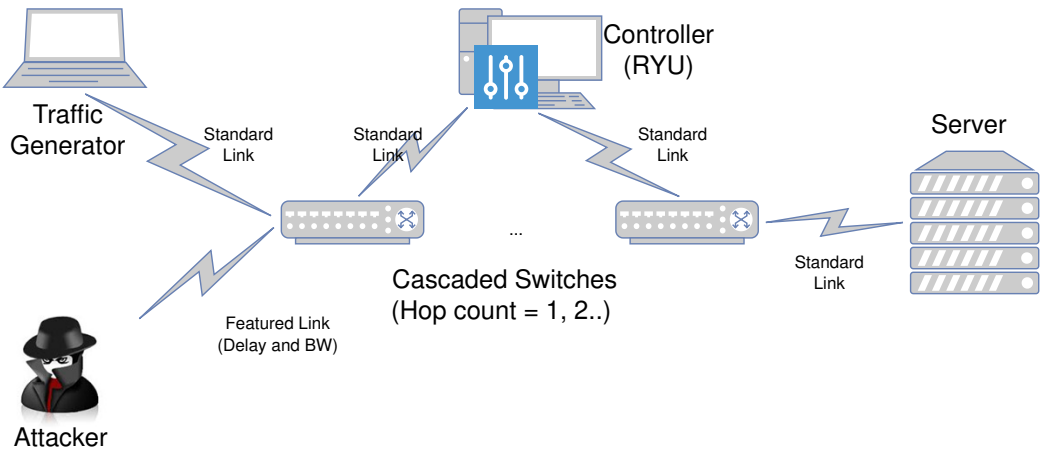


Figure 3.4. Simulation topology for *RAFA* and *DAFTR*.

Table 3.4. Network parameters for network type simulations.

SDN Simulation Network Parameters	
Hop Count	1, 2
Link Delay (ms)	0, 100, 200
Bandwidth (Mbps)	10, 100, 1000
BT (Mbps)	10, 100, 1000

3.6.1. Network Type Fingerprinting Experiments

In this section, we run some experiments to infer the network type. One tailed t-test is used for network type fingerprinting. For simulations, α is set to 0.01. To test accuracy of our algorithm, both traditional network and simulated SDN network are used. Firstly, *RAFA* is checked against traditional networks. Hence 30 public servers across the globe (from different countries and continents such as Austria or Argentina) are selected. These servers are used as traditional network components to test success likelihood of the *RAFA* in traditional network settings. There is no guarantee that the servers are not in SDN domain. Nevertheless, this situation is very unlikely.

To better understand the effect of delay, we aim for delay diversity in our selection: RTTs of the servers varies between 5ms to 350 ms. In addition to delay, to consider changing network conditions like background traffic, the simulation is executed in every hour for 24 hours. Hence there are $24 * 30 = 720$ simulations. *RAFA* correctly identifies 714 of them. Therefore, the success ratio is 0.99 (714 out of 720).

To test an SDN network, we use our simulated topology. To better understand the impact of the network parameters and investigate the robustness of our solution, the same topology is generated with different characteristics. Hence the fingerprinting success rate with respect to hop count, link delay, bandwidth and background traffic (BT) is investigated. Network parameters for the SDN fingerprinting simulation are listed in Table 3.4.

Table 3.5. Network fingerprinting results for hop count 1.

(BW, BT)	**LD = 0**	**LD = 100**	**LD = 200**
(10,10)	1	1	0.98
(10,100)	1	1	1
(10,1000)	1	1	1
(100,10)	1	0.99	0.99
(100,100)	1	1	1
(100,1000)	1	0.99	1
(1000,10)	1	1	0.98
(1000,100)	1	1	1
(1000,1000)	1	0.98	0.98

There are 54 distinct network conditions and each condition is simulated 100 times to analyze the accuracy. The network parameters (link delay and bandwidth) are applied to the link of the attackers. The hop count increases the number of switches in a cascaded manner in the topology. For background traffic, a traffic generator generates traffic in the switches.

The accuracy of the *RAFA* is shown in Table 3.5 for hop count equal to 1. Highlighted cells show the results which are not equal to 1. The success ratio is 1.0 when hop count is set to 2. Hence overall success ratios of the simulations is between 0.98 and 1.0. It seems that as the hop count increases, the success probability is affected positively while link delay has an opposite effect.

These results are meaningful since as the hop count increases, flow setup delay also increases and as the link delay becomes larger, the effect of the flow setup delay becomes limited. Because RTT also raises in this case but the flow setup time for the network does not increase and remains invariable. Bandwidth and background traffic seem to have negligible impact on the success ratio. However, BT can affect the ratio positively. As BT increases, the number of new flows, i.e., `packet-in` count, also increases since the flow setup delay is affected by the load of the controller. The

bandwidth of the attacker can have a positive impact since as the bandwidth becomes larger, the transmission delay of the packet becomes smaller. However, this change is not significant.

RAFA has better accuracy with more challenging network conditions compared to the related work ([15–17]). For scalability analysis, we conduct 54 different simulations by examining the effects of background traffic, link delay, bandwidth and hop count in experiments. [16] only considers hop count and bandwidth while [17] looks at background traffic. In [15], only a traditional network is tested. The authors used predefined flow setup time and added it to the response time of the second packet to simulate an SDN network. The success ratio of the solution is 0.85.

In [16], 20 clients across the globe are involved to test an SDN network. Their success ratio is between 0.92 and 1.0. The effectiveness of their ratio exhibits a marked decline when the network includes soft switches. The solution in [17] fails completely in accurately inferring the network type in presence of heavy background traffic. Besides, there are no simulation details like the number of simulations or the bandwidth of the background traffic.

3.6.2. Timeout Fingerprinting Simulations

In this section, we conduct some experiments to infer flow rule timeouts. For simulations, the hard and idle timeouts are set to 10 and 5 seconds, respectively. Parameters `iterationCount` (τ) and `majority coefficient` (ρ) are 5 and 0.5, respectively. Hence a timeout value should be observed at least 3 times. Network parameters for the simulation are shown in Table 3.6. Since the hop count and the link delay have the most noticeable effect on accuracy, we test them extensively. Besides as we see from the previous simulations, bandwidth and BT do not have noticeable effect on the results, the maximum values are used for them. Thus, there are 9 different network conditions and each condition is simulated 50 times to evaluate the accuracy performance.

Table 3.6. Network parameters for timeout simulations.

Timeout Simulation Network Parameters	
Hop Count	1, 2, 3
Link Delay (ms)	0, 100, 200
Bandwidth (Mbps)	1000
BT (Mbps)	1000

Table 3.7. Idle timeout simulation results.

		Hop Count		
		1	**2**	**3**
Delay	**0**	1.0	1.0	1.0
	100 ms	1.0	1.0	1.0
	200 ms	0.92	1.0	1.0

The success ratio of hard timeout simulations are 1.0. The accuracy of the *RAFA* for idle timeout fingerprinting is shown in Table 3.7. Similar to the network type experiments, it seems that as the hop count increases, the success probability is affected positively while link delay has opposite effect.

In [17], the presence of background traffic significantly diminishes the accuracy of timeout inference attacks. In case of background traffic, the accuracy is less than 40% and becomes nearly zero for some type of scan traffic. Without background traffic, the success ratio is 53% and 66% for hard and idle timeouts, respectively. In [20], the exact value of the timeouts cannot be found. Timeouts are found in plus-or-minus 10% error margin. The success ratio of the timeout simulations is 96% in [41]. In [18], similar to [20], the timeout values are measured in a plus-or-minus 3% error margin. There is no timeout simulation result in [42]. For scalability analysis, we conduct nine different simulations with different link delay and hop count parameter values. In that regard, all the related work does not consider any network parameters, except [17] which only considers background traffic. Again, *RAFA* has better accuracy for more challenging network conditions compared to the related work in [17, 18, 20, 41, 42].

Table 3.8. Hard timeout simulation results for [17].

		Hop Count		
		1	**2**	**3**
Delay	**0**	0.36	0.46	0.32
	100 ms	0	0	0
	200 ms	0	0	0

Table 3.9. Idle timeout simulation results for [17].

		Hop Count		
		1	**2**	**3**
Delay	**0**	0.06	0.08	0.14
	100 ms	0	0.1	0.1
	200 ms	0.2	0.14	0.08

However, these comparisons are naturally based on the best of our knowledge (i.e., what is revealed in those published works) regarding the design and the performance of those algorithms by the respective authors and for different (simulated or real) network environments. Please note that, for instance, *RAFA* might have a "more favorable" network environment. Essentially, we implemented the proposed algorithms in [17, 41] from these available alternative techniques in the literature. As we point out earlier, there is only procedural explanation in [20, 42] which has missing parts for a specific implementation such as RTT surge detection method. Besides, [20, 42] employ a very similar approach with [17, 41].

Table 3.10. Hard timeout simulation results for [41].

		Hop Count		
		1	**2**	**3**
Delay	**0**	0.88	0.86	0.64
	100 ms	0	0	0
	200 ms	0	0	0

Table 3.11. Idle timeout simulation results for [41].

		Hop Count		
		1	**2**	**3**
Delay	**0**	0	0	0
	100 ms	0	0	0
	200 ms	0	0	0.02

The simulation results of [17, 41] are shown in Table 3.8, 3.9, 3.10, 3.11. The experiments are executed in our simulated environment. The results show that *RAFA* outperforms in every scenario in terms of success likelihood of the probing attack. As expected, since the referenced works try to calculate idle timeout first, it leads to very low accuracy, especially when the link delay is high.

3.6.3. Fingerprinting Countermeasure Simulations

We test our lightweight defense solution which randomizes flow rule timeouts according to the network status. Initial randomization coefficient, which applies to all flows, is set to 0.1.

The randomization coefficient is used to find a range of possible timeout values. Consider that hard timeout is 10 seconds. If the randomization coefficient of a port equals to 0.2, hard timeout is randomly set between 8 and 12 for the flows of that port. Basically real hard timeout value is multiplied by the randomization coefficient and the result is added/subtracted to find the range of the hard timeout values. Similar to timeout fingerprinting simulations, hard and idle timeouts are set to 10 and 5 seconds, respectively. To better focus on probing flows, while counting `flow removed` messages, long and large flows are ignored. Duration threshold, $\text{FD}_{\text{threshold}}$, equals to $2 * (hardtimeout + idletimeout)$ for this simulation and the duration bigger than 30 seconds is assumed as long. The threshold for large flows, $\text{PC}_{\text{threshold}}$, are measured as packet count and the value 500 is considered large. The time window is 180 seconds while $\text{Ratio}_{\text{threshold}}$ is 0.5.

When our defense solution becomes active, hard timeouts are found as "not active or not deducible" as a result of our majority rule in timeout fingerprinting. Since hard timeouts are not found properly, idle timeouts are found incorrectly likewise. To measure the effect of our solution in terms of performance, *RAFA* attacks the network while the defense mechanism is on and off. We control average response time to the `packet-in`, CPU and memory usage metrics to understand the effect of the randomization on the network. The performance simulation is run 100 times with 50 of them having active defense solution. According to the simulations, the average response time for `packet-in` increases by 3.9% from 5.89 to 6.12 milliseconds. The CPU and memory usage figures are increased by 5.78% and 0.77%, respectively. The results show that the proposed method slightly impacts performance and effectively prevents fingerprinting attacks.

3.7. Discussion

SDN fingerprinting, which involves identifying the network type and timeout values, can be carried out both with and without the use of spoofing. Although it depends on the match fields of the flow rules, parallel execution of the fingerprinting algorithms typically needs IP spoofing. (If the match fields include layer 4 port information, flow rule generation can be triggered by just changing the port numbers.) Spoofing can be detected more easily if the attacker is inside the targeted network. However, it is a challenging task when outsider attackers are considered. Hence, detection or prevention of SDN fingerprinting is not a straightforward goal with traditional security approaches. In our simulations, the fingerprinting process is executed without spoofing.

The duration of the SDN fingerprinting process varies according to the network characteristics, fingerprinting algorithm parameters and also the execution method of the algorithms. In the case of network type fingerprinting, executing the algorithm in parallel can yield results in less than a second. Conversely, it can take more than a month if the attacker sends two packets (to calculate T_1 and T_2) in each day. Timeout fingerprinting has similar logic but it also highly depends on actual hard ($T_{\text{hardActual}}$)

and idle ($T_{idleActual}$) timeout values. Our algorithms can infer the hard timeout value in $T_{hardActual} + 1$ seconds. Duration of idle timeout calculation is strongly affected by $T_{hardActual}$, $T_{idleActual}$ and T_{step}. Like in the simulations, if $T_{hardActual}$ and $T_{idleActual}$ are set to 10 and 5 seconds, respectively, it takes approximately one minute. Despite these variations in duration, the fingerprinting process generally has a sufficiently short duration to avoid detection, much like in simulations.

There are some improvement points for *RAFA* as future work. First, binary search can be used instead of incremental increase in the sleep interval while discovering flow rule timeouts. In this way, fewer packets and less time can be spent, which can then reduce the chance of being detected as an attacker.

In simulations, delay is one of the parameters investigated to understand its effect on the accuracy of the proposed solution. Actually, jitter can be more relevant since its variation can remarkably degrade the success ratio of the fingerprinting mechanism. However, implementing jitter based experiments in network environment can be quite challenging since it is affected from a plethora of dynamic network parameters and states.

Typically it is thought that flow rules are deleted immediately when timeout value reaches to zero. However, generally each switch software has different internal mechanism for flow deletion. Some schedule deletion according to the status of the switch CPU or current traffic load while other can use predefined intervals for deletion. Hence our method uses majority principle in timeout fingerprinting which can increase overhead for the attacker but also provides better results in terms of accuracy.

While on the road to the target network, our implicit assumption is that the probing packets exclusively traverse a single SDN network. The viability of our algorithm in real-world scenarios remains uncertain, primarily due to the lack of data regarding the deployment ratio of SDN in the telecommunications sector. Exploring the transit of packets through a multi-domain SDN in a fingerprinting scenario could

be an interesting research direction for future work. Similar to wireless networks, multi point calculation from multiple measurement locations can be employed to figure out multi-SDN domains and adjust probing packets accordingly to efficiently fingerprint the network. This becomes particularly intriguing when switches on the path have different flow tables sizes. Intuitively, the edge switch is the most vulnerable to saturation attacks as it is directly exposed to the attacker. It remains an open question how feasible it is to conduct similar reconnaissance and attacks on switches deeper in the network.

There are a number of SDN controllers such as ONOS [54], OpenDaylight [60] and Floodlight [61]. In simulations, RYU was used as the controller since the main idea of the attack is essentially to exploit the difference of RTTs of the first and second packets. This difference consists of processing time of the switch and controller, and link delay. Naturally, the processing time of the controller can affect the result. However, the main component of the difference comes from the communication cost. Hence repeating the simulations with different types of controller are expected to yield similar results except some border-line conditions such as really restricted resource assignment to a controller.

The simulation topology employed in our research is relatively simple, as the inclusion of additional hosts or switches has minimal effect on the fingerprinting process. In fact, more host and switches can cause more load on the controller which potentially increases success ratio of the fingerprinting since the delay in installing flow rules is contingent on the controller's workload. We already consider path length, link delay, background traffic and bandwidth of the network. End to end latency of the probing packet is the key factor for the fingerprinting since the whole idea relies on RTT difference of different packets. Moreover, the difference in RTT can significantly increase if there is more than one switch installed between the communicating end hosts, with each switch repeating the process of sending the unknown flow to the controller. Therefore, the number of hop count and link delay have the most significant impact on the success ratio of the fingerprinting. This outcome can also be seen from the simulation results. However, please note that more scenarios with different network

characteristics may provide additional insights.

As mentioned before, in simulations, background traffic (BT) appears to have a negligible impact on the success ratio. However, various factors such as flow setup delay, switch load and the load on the target server or attacker can influence the success ratio. For instance, an increase in BT can elevate the `packet-in` count and the load on the controller, thereby increasing the flow setup delay. Conversely, an increase in BT can also heighten jitter, causing artificial RTT surges and thus reducing the success ratio. To mitigate this, timeout fingerprinting algorithms are executed multiple times. In network type fingerprinting, a set of measurements is collected. Additionally, two packets are sent almost consecutively to minimize jitter-related issues.

The idle timeout fingerprinting algorithm takes the hard timeout as an input. Hence, a miscalculation in the hard timeout can lead to successful fingerprinting with/without sending extra packets or completely failing the fingerprinting. For example, if the actual hard and idle timeouts are 30 and 15 seconds, respectively, and the calculated hard timeout is 25 seconds, the idle timeout fingerprinting can still yield correct results but may require more packets than usual. However, if the calculated hard timeout is 35 or 10 seconds, the idle timeout fingerprinting will fail. However, there can be scenarios where miscalculation does not affect the idle timeout's accuracy. For instance, if the actual hard and idle timeouts are 360 and 5, respectively, and the calculated hard timeout is 300 seconds, the idle timeout calculation proceeds correctly. In this context, interval step times (T_{step}) is a crucial parameter. Thus, the error in hard timeout calculation can influence accuracy in varied settings. Modeling this discrepancy with a distribution and assessing its impact on accuracy can be an intriguing future research direction.

A mathematical model or formulation can be developed for timeout fingerprinting. However, modeling network characteristics can be cumbersome. Additionally, fingerprinting necessitates the exchange of packets with the target network, making it a practical challenge. If the response times of appropriately designed packets are

available, a model or formulation can be generated, potentially with the assistance of machine learning techniques.

In network type fingerprinting, a receiver operating characteristic curve can be generated by adjusting α values, as it is the only threshold we use. However, it is important to note that the generated curve will be highly dependent on network characteristics. Without knowing or assuming these characteristics, setting a lower α value, such as 0.01, will likely increase the accuracy.

3.8. Conclusions

In this work, we introduce the concept of SDN fingerprinting and the type of information that can be gathered from it. While there is a significant amount of work that uses SDN as a security enabler, the security of SDN itself is a critical concern. To address this, we propose *RAFA*, an automated and robust SDN fingerprinting agent that can infer the network type (traditional or SDN) and hard and idle flow rule timeout values.

RAFA is robust and achieves high accuracy in challenging network conditions. Extensive simulations have been conducted with a diverse range of network types and parameters, showing that *RAFA* performs nearly perfectly in all circumstances. We also present a lightweight and dynamic flow rule timeout randomization mechanism, *DAFTR*, that takes into account the network status and increases the cost for attackers, as changing the features of the network makes it harder to explore and exploit.

As a future research direction, machine learning algorithms could be leveraged to extract more diverse information from an SDN network through fingerprinting techniques. Besides additional network parameters can be randomized by considering a wider set of network states.

4. UNMASKING SDN FLOW TABLE SATURATION: FINGERPRINTING, ATTACKS AND DEFENSES

4.1. Introduction

SDN stands as a pivotal technology in attaining the essential levels of flexibility and scalability demanded by pervasive and high-performance network infrastructure required for digital connected services. Nonetheless, its disaggregated and layered architecture makes it open to the time-based fingerprinting attacks. Besides, limited flow table capacity of the switches alleviates table saturation attacks. In this section, an automated attacker tool called *TASOS* is proposed to infer flow table utilization rate, size and replacement algorithm. With this set of information, the attacker can conduct intelligent saturation attacks. Furthermore, a lightweight defense mechanism (*LIDISA*) for proactively deleting flow rules is described. A comprehensive simulation setup with different network conditions shows that the proposed techniques achieve superior success rate in diverse settings.

In previous section, we inferred network type (SDN or traditional) and flow timeout values (hard and idle). Besides, a lightweight mechanism that adjusts timeout values to increase complexity and costs for attackers was proposed for defense. Likewise, this new section covers both offensive and defensive techniques regarding SDN fingerprinting, and their effectiveness is assessed through simulation-based experiments. The principal contributions of our work can be summarized as follows:

- A comprehensive and automated module for conducting flow table saturation attacks, named *Table Analyzer and Saturation Orchestrator for SDN (TASOS)*. This module effectively fingerprints flow table-related details, including size, utilization rate and replacement policy, facilitating the implementation of effective saturation attacks through informed decision-making.
- The development of an agile and dynamic defense mechanism that increases the

complexity and costs for potential attackers by dynamically adjusting flow table attributes in response to fluctuations in network status.

- Conducting extensive simulations to showcase the remarkable accuracy and adaptability of these proposed methods across various network conditions.

The structure of this section is as follows: We begin by presenting fundamental aspects of SDN relevant to this work in Section 4.2. Subsequently, we delve into a discussion of related work in Section 4.3. This is followed by an introduction to *TASOS* in Section 4.4 and our defense mechanism in Section 4.5. In Section 4.6, we detail the methodology of our extensive simulations and present the results, with a particular emphasis on the performance of *TASOS* and our defense mechanism. Section 4.7 addresses limitations and open questions, providing a critical perspective on our outcomes. Finally, in Section 4.8, we explore potential avenues for future research and conclude the section.

4.2. SDN Fundamentals

Figure 3.1 in Section 3.2 shows the packet processing logic of SDN. We will use the designations T_1, T_2 and T_3 to represent RTT required for the *Fast Path*, *Slow Path* and *Slowest Path*, respectively. An attacker can dispatch an initial packet to generate a new flow, and then the corresponding RTT, T_2, can be calculated. Subsequently, a second packet is transmitted and its RTT, designated as T_1, is calculated. Following this, packets are dispatched with the intention of inducing the installation of a new flow for each of them, until the RTT for such a packet significantly surpasses T_2. This occurs when the flow table reaches full capacity, making the slowest path relevant. This specific RTT is indicated as T_3. For successful SDN fingerprinting, T_2 and T_3 should be significantly bigger than T_1 and T_2, respectively.

When the flow table becomes full, newly arriving flows face two possibilities: they are either dropped, or an existing flow is evicted from the table to make space for the new request. However, removing an existing entry from the table introduces

latency and places an additional workload on the controller. This initiates a sequence of message exchanges between the controller and the switch, ultimately driving the flow table operations. The flow replacement policy becomes relevant when the controller attempts to install a new flow on a switch with a full table. The switch responds with a `table_full` message and the controller can either choose to drop the new flow or install it by removing an earlier flow table entry.

OpenFlow currently offers three mechanisms for flow table management: flow timeouts, flow entry deletion and flow entry eviction, as specified in the OpenFlow specifications [46]. Switches are responsible for implementing timeouts and eviction, while the controller can delete flows by sending flow delete message. In OpenFlow 1.4, an "importance" field for the flow rules was introduced as an optional feature. This field is used in the eviction process and flow entries are evicted strictly in order of importance. Flow entries with lower importance will always be evicted before those with higher importance.

4.3. Related Work

In this section, we offer a comprehensive overview of the literature concerning general SDN fingerprinting and strategies aimed at devising advanced fingerprinting techniques. We begin by providing an overview of general SDN fingerprinting works, followed by a more detailed examination of those specifically related to flow tables and saturation attacks.

The controller and SDN switches are also susceptible to saturation attacks. Early studies, such as [15] and [62], explore packet-in flooding attacks in which attackers aggressively generate a massive volume of `packet-in` messages to flood the SDN controller. High packet-in rates are a key characteristic of this attack type. AVANT-GUARD [63] introduces a connection migration module and actuating triggers module to extend the data plane functions, but it is applicable to TCP protocol only. Subsequently, FloodGuard [64] presents a protocol-in-dependent defense framework that

pre-installs proactive flow rules to reduce table-miss packets and forwards those packets to additional data plane caches for mitigation. FloodDefender [62] takes a different approach by offloading table-miss packets to neighbor switches and employing a two-phase filtering strategy to filter out attack traffic. Lineswitch [65] extends OpenFlow with the goal of mitigating the same type of attack.

In [18], the authors endeavor to identify a broad spectrum of SDN policy parameters, encompassing attributes such as hard and idle timeouts, the match fields of flow rules and the controller's response to a `table_full` event (whether it rejects or deletes flow rules). Additionally, they leverage the common utilization of the fat-tree topology in SDN-deployed data centers to deduce certain information about the network's structure, including in-rack, in-pod and inter-pod groupings. As a defensive strategy, the paper proposes dynamic timeout assignment and the adjustment of the number of match fields in accordance with the load on the flow table. However, there is no performance evaluation available for the `table_full` event fingerprinting.

In [19], the state of the flow table, including its size and utilization rate, is deduced using peak and level-shift detection algorithms in conjunction with the T_1, T_2 and T_3 logic. For the replacement algorithm, only FIFO is considered. Notably, their algorithms are tailored to operate in scenarios where no background traffic is present and flow rules are permanent, which severely limits their practical applicability.

In [12, 20], the authors employ the T_1, T_2 and T_3 metrics to ascertain the size of the flow table and the number of active flow rules within it. Additionally, they leverage measurements to deduce whether the flow table replacement algorithm used is LRU or FIFO, achieving an 80% accuracy rate. Furthermore, the authors put forth two defensive strategies, namely routing aggregation and multilevel flow table architecture, although these solutions remain theoretical and lack practical implementation. It's worth noting that their timeout fingerprinting algorithms have been identified as having certain limitations, as pointed out in previous section.

In [21], the authors identify the match fields of flow rules by changing packet headers. This information is then utilized to perform a packet-in flood. Additionally, they infer the policy for statistics collection enabling statistics-related attacks, which are out of scope of this thesis. This allows the attacker to trigger statistics collection at their discretion. However, the paper does not provide performance tests to evaluate the accuracy of these methods. For defense, the authors propose the use of a multi-queue based structure in the switches, along with scheduling different downlink messages with specific types and priorities. Simultaneously, the controller monitors downlink message events and assigns varying priorities to different messages as a protective measure. This defense mechanism needs to modify the switch design to be integrated into a practical system.

In [22], an intelligent control plane saturation attack is devised, which incorporates a reconnaissance stage to gather information about the flow table size, utilization and the replacement policy (LRU or FIFO). The success ratio of the attack is reported as 90%. It's important to note that the attack design assumes that all the rules are stored in a single table. The reconnaissance stage is ideally carried out during periods of light background traffic, as heavy traffic significantly diminishes the accuracy of the reconnaissance. The algorithm is unable to precisely determine the flow table size and replacement policy, necessitating multiple iterations for improved accuracy.

In [11], a flow table saturation attack targeting dynamic timeouts is introduced. The attack is divided into three phases: inference, triggering and maintenance. Firstly, in the inference phase, the attacker deduces the type of timeout strategy being used (fixed, random, or dynamic) and identifies the potential timeout values for inserted flow rules. In the triggering phase, the attacker initiates flow rule installation. This is accomplished by employing multiple rounds of probes to induce the reinstallation of flow rules, compelling the controller to assign lengthy timeouts to them. This phase generates attack flows and contaminates the flow table with malicious flow rules. Finally, in the maintenance phase, the attacker crafts and dispatches packets with inter-arrival times falling within the idle timeout period, ensuring that the corresponding flow entries

remain active indefinitely. Apparently, the timeout fingerprinting algorithm outlined does not account for hard timeouts when searching for idle timeouts. This oversight could potentially yield misleading results.

In [66], the authors combine the packet-in rate and the control channel occupation rate (CCOR) as inputs for Linear Discriminant Analysis (LDA) to detect saturation attacks. Subsequently, the controller assembles a white-list and installs migration flow rules as part of their defense strategy.

In [23], match fields, including their bit-masks and flow rule timeouts are inferred during the probing phase. To flood the flow tables based on the inferred network configurations, the lower bound of the attack rate is calculated. Each attack packet is designed to trigger or maintain the consumption of one flow rule. However, it's worth noting that their timeout fingerprinting algorithm demands numerous iterations, resulting in a substantial number of probing packets and potentially requiring several hours to complete.

In [24], flow timeouts and flow table size are determined using algorithms from the literature. Subsequently, a slow TCAM exhaustion attack is executed based on the inferred parameters by compelling the installation of new forwarding rules and perpetually maintaining them active. Botnets are employed for flow installation. As a defense mechanism, it suggests randomly selecting rules to be dropped whenever the system is overloaded.

The flow replacement algorithm determines how the flow table manages entries when it becomes full. Therefore, knowledge of the flow table replacement algorithm is crucial for an effective saturation attack. That algorithm governs the internal state transition of an SDN network, making it an essential part of the inference model. Replacement fingerprinting has been explored in [18,20,22]. However, [18] only determines if a replacement algorithm exists or not, while [20, 22] focus on LRU and FIFO. Our algorithm covers four different algorithms: REJECT (rejecting new flows), LRU, FIFO

and RANDOM. REJECT is important since default controller applications typically do not respond to `table_full` error messages, relying on automatic removal of flow entries through the timeout mechanism [18]. RANDOM is also popular, as it performs better than FIFO with considerably less overhead [67].

[19, 20, 22] try to find flow table size and utilization. However, [20, 22] use fixed thresholds and none of them employ statistical methods to calculate thresholds based on response times. Additionally, [23] does not provide an algorithm for table size fingerprinting but mainly presents their measurement procedure. Our algorithm, *TASOS*, stands out by incorporating statistical methods in threshold calculation for a more adaptive approach.

In the saturation attack, it is imperative not only to deplete the TCAM memory of the SDN switch by compelling the installation of new flow rules but also to sustain their perpetuity, thereby obstructing new rules from being installed to cater to legitimate endpoints. Consequently, the inference of timeouts becomes essential. Furthermore, a mechanism must exist to ensure the steady activation of the inserted flow rules. The attacker sends packets with a meticulously calculated inter-arrival time, ensuring that the flow rules never expire due to timeouts.

Another consideration is that the rate and duration of packet-in messages serve as common indicators for detecting such attacks, as observed in studies like [64, 66]. Hence, maintaining a low packet-in rate during the attack is crucial to elude detection. Notably, [19, 21, 22] do not incorporate flow timeouts into their analyses. In [20], attempts are made to flood the flow table within a single timeout duration, potentially revealing the probing attack. In *TASOS*, the attacker retains the ability to adjust the packet-in rate, allowing for a gradual flooding of the flow table with malicious flow rules and ultimately achieving saturation of the entire storage space, thereby circumventing the mentioned defenses.

Evidently, the dimensions of the flow table, encompassing its size and replacement algorithm, are of central importance in orchestrating efficient and sustained saturation attacks. Without precise knowledge of the flow table size, the attacker remains incapable of determining when to cease the influx of new flow rules. The choice of replacement algorithm is also pivotal, as it dictates the fate of the attacker's flows when the table reaches its capacity. For example, if FIFO is implemented as the replacement algorithm, the attacker simply needs to dispatch update packets without letting the flows expire. Conversely, in the case of LRU, the attacker must extend the update packet intervals, considering that the least recently used flows are evicted. Notably, [11,21,23,24] overlook the consideration of replacement algorithms, while [11,21] neglect to account for the flow table size.

When executing a saturation attack, the number of packets required is directly related to the cost of the attack, which is why an attacker is typically interested in keeping it as low as possible. During the reconnaissance phase, *TASOS* systematically gathers flow table-related data. Subsequently, guided by this intelligence and the attacker's chosen packet-in rate, the saturation attack is orchestrated with minimal resource expenditure. Noteworthy is also the $O(n)$ computational complexity of *TASOS*, surpassing the at least $O(n^2)$ complexities evident in works such as [11, 20, 22–24].

In summary, *TASOS* stands out as the only work that comprehensively addresses all essential fingerprinting information, including flow table size, utilization rate, and replacement algorithm. *TASOS* fingerprints a broader range of replacement algorithms, including the new RANDOM algorithm, and employs statistical methods to determine thresholds. It maintains a low rate relative to flow timeouts and replacement algorithms in a significantly more efficient manner. Additionally, *TASOS* uniquely performs fingerprinting and saturation attacks simultaneously.

Table 4.1 succinctly encapsulates key technical contributions, delineating their core methodologies, inferred SDN information through fingerprinting, and any applied defense mechanisms where present.

Table 4.1. Related work of flow table fingerprinting and saturation attack.

Work	**Mechanism and Fingerprint Info**
Ahmed et al. (2020) [18]	Infer SDN policy parameters such as hard and idle timeouts, match-fields, controller reaction at `table_full` event and information about the topology.
Gregorczyk et al. (2021) [19]	Infer flow table capacity and flow table usage.
Leng et al. (2018) [12, 20]	Infer flow table capacity and flow table usage. Determine flow replacement algorithm (LRU or FIFO).
Zhang et al. (2020) [21]	Infer match-fields and statistics policy (period and type). Packet-in flood and statistic query attacks. Defense via multi-queue based structure to schedule messages.
Yu et al. (2021) [22]	Infer flow table capacity and flow table usage. Determine flow replacement algorithm (LRU or FIFO). Intelligent packet-in flood attack.
Shen et al. (2023) [11]	Infer flow timeouts. Advanced saturation attacks against dynamic timeouts.
Huang et al. (2022) [66]	Detection of saturation attacks by using packet-in rate and control channel occupation rate.
Cao et al. (2018) [23]	Infer match-fields including their bit-masks and flow timeouts. Low rate saturation attacks.
Pascoal et al. (2020) [24]	Low rate saturation attacks by using existing fingerprinting techniques. As defense, randomly evict some flow rules when the flow table is full.

In a nutshell, a comprehensive survey of existing literature underscores the plethora of SDN fingerprinting approaches, yet their robustness remains subpar, often thriving solely in controlled environments. Concurrently, current saturation attack designs exhibit drawbacks such as inefficiencies and a lack of practical/real-world implementations, signaling the need for enhancements. Furthermore, the scarcity of solutions addressing slow-rate saturation attacks in the domain of defense mechanisms underscores the need for more robust approaches on the attacker side. In particular, adopting a lightweight defense strategy centered on saturation attacks becomes pivotal to foster broader acceptance of anti-fingerprinting techniques.

4.4. Table Analyzer and Saturation Orchestrator for SDN (*TASOS*)

In this section, we introduce *TASOS* as a tool to deduce flow table size, load, and replacement algorithm, facilitating efficient saturation attacks. *TASOS* demonstrates enhanced efficiency and accuracy in fingerprinting compared to previous methods, particularly in diverse and challenging network conditions. Employing statistical methods tailored for such conditions, *TASOS* operates as an automated saturation attack tool. Once the attacker selects the target network, the fingerprinting process operates autonomously. In prior work, we established that an attacker can discern the network type (traditional or SDN) and flow rule timeouts (hard and idle). *TASOS* builds upon this information, enabling an attacker to initiate a saturation attack effectively.

The procedure initiates with the identification of the network type. If recognized as an SDN, the process starts timeout fingerprinting to deduce hard and idle timeouts. Subsequently, flow table fingerprinting is activated, discerning flow table utilization rate, replacement policy and flow table size, respectively.

TASOS operates under the same three key assumptions as *RAFA*: the knowledge of one flow match-field, the availability of a host capable of reaching a server within the target network, and the existence of only one SDN network along the path between these entities. Besides, the same threat model mentioned in Chapter 3 is also

applicable.

OpenFlow provides two timeouts for flow rules: hard and idle. According to the OpenFlow specification, these timeouts can be assigned values ranging from 0 to 65535 seconds [46]. A value of 0 signifies that the timeout is unset, allowing flow rules to persist until prompted for deletion by the controller. The idle timeout mechanism is deemed more adaptable than hard timeouts. Hard timeouts may result in rule expiration during bursts of packet transmission, leading to issues such as packet loss, increased latency, or degraded network performance [68]. Furthermore, the literature indicates limited adoption of hard timeouts [69]. Consequently, hard timeouts are not factored into the execution of our saturation attack.

To fingerprint an SDN and infer flow table states, careful determination of T_1, T_2 and T_3 thresholds is crucial. Therefore, in *TASOS*, the initial two packets of a new flow are dispatched to a server within the target network. The first RTT (RTT_1) and second RTT (RTT_2) values are subsequently measured. To render this statistically meaningful, these measurements need to be conducted multiple times. The measurements can be performed in parallel or sequentially; however, parallel execution may elevate the risk of detection for the attacker. Once a set of RTT_1 and RTT_2 measurements is amassed, the sample mean and variance of RTT values are leveraged to determine thresholds, which are then used to infer the slow and slowest paths. Consequently, the sample means (RTT_{1mean} and RTT_{2mean}) and sample standard deviations (RTT_{1std} and RTT_{2std}) of RTT values are computed initially.

The slow path or T_2 indicates the installation of a new flow, providing valuable information for flow deletion control in the context of saturation attack. Statistically, 99.7% of the data falls within three sample standard deviations of the mean. Consequently, to establish a statistically valid threshold, the formula $T_2 = RTT_{2mean} + 3 \times RTT_{2std}$ is employed. Thus, when the RTT of a packet surpasses T_2, it is inferred that the flow associated with the packets has been deleted.

Table 4.2. Parameters for flow table fingerprinting algorithm.

Parameters	**Explanation**
iterationCount (τ)	Number of times the replacement algorithm identifier algorithm is executed.
flowCountPerSecond	Number of new flows per second.
$\mathbf{T_{idle}}$	Idle timeout.
flowUpdatePeriod	Period of the flow update mechanism (in seconds).
$\mathbf{T_{residual}}$	Residual idle timeout time used for flow update.
$\mathbf{FT_{size}}$	Flow table size.
$\mathbf{FT_{residual}}$	Residual flow size of the flow table.
RA	Flow replacement algorithm.
NoAF	Number of active flows.

The slowest path with T_3 indicates an attempt to install a new flow when the table is full, offering insights into the flow table state. Similar to T_2, the statistically valid threshold is determined using the formula $T_3 = RTT_{1\text{mean}} + 3 \times RTT_{1\text{std}}$. If the RTT of a packet exceeds T_3, it is concluded that a flow rule is sent to the switch while the switch's flow table is full.

The parameters utilized in our algorithms are detailed in Table 4.2. Initially, *TASOS* operates in two distinct threads: one for the installation of new flows and another for sending periodic packets to ensure the longevity of generated flows. Both threads run periodically. Given that SDN saturation attacks do not demand extensive traffic and existing defenses are often triggered by traffic volume [24], the attacker can strategically choose a flow installation rate, denoted as *flowCountPerSecond*, to circumvent defense mechanisms. The period of the flow installation thread is determined as $1/$*flowCountPerSecond*. Additionally, the period of the flow updater thread can be adjusted by setting the *flowUpdatePeriod* parameter. It's essential for *flowUpdatePeriod*

to be less than the idle timeout, denoted as T_{idle}.

Algorithm 1: Flow Update Algorithm

Data: T_{idle}, $\text{T}_{\text{residual}}$

```
1 for flow, timestamp in flows do
2     if (now - timestamp) >(T_idle - T_residual) then
3         update flow and its timestamp
```

The flow update algorithm is outlined in Algorithm 1. The flow update thread assesses active flows and their last update time (Line 1). If the time elapsed since the last update surpasses $T_{\text{idle}} - T_{\text{residual}}$, a packet is dispatched for that flow and its update time is refreshed (Line 3). Here, $\text{T}_{\text{residual}}$ is pivotal, necessitating a setting between $flowUpdatePeriod < T_{\text{residual}} < 2 \times flowUpdatePeriod$ to ensure that flows never expire.

The flow table fingerprinting algorithm is shown in Figure 4.1. Initially, the process commences with the initialization of parameters (Lines 1-5). Subsequently, within a loop, new flows are sent. If the RTT of a newly introduced flow exceeds the threshold denoted as T_3 (Line 8), it is inferred that the flow table has reached its capacity. Following this, a check is performed to ascertain the status of $\text{FT}_{\text{residual}}$ (Line 9). If it remains unset, $\text{FT}_{\text{residual}}$ is established as the count of active flows (Line 10). Thereafter, an assessment is made regarding the existence of a Replacement Algorithm (RA) (Line 11). In the event of its absence, the initiation of the flow table replacement fingerprinting procedure is started (Line 12).

The flow table replacement finder algorithm is delineated in Figure 4.2. Initially, an examination is conducted to determine if the RTT is equivalent to the designated *maxValue* (Line 1). Here, *maxValue* denotes the timeout of the new connection. The intention is to establish a new connection with the transmitted packet. However, when the connection fails, a timeout has occurred. Detecting a timeout implies that the controller rejects the new connection due to a full table.

```
   Data: iterationCount (τ), T_3
   Result: FT_size, FT_residual, RA
1  FT_size ← −1, algorithmCounter ← 0 and FT_residual ← −1
2  RA ←"Searching" and sendNewFlow ← True
3  while sendNewFlow do
4      Send new flow and get RTT
5      if RTT > T_3 then
6          if (FT_residual == −1) then
7              FT_residual ← NoAF
8          if RA is "Searching" then
9              Replacement Algorithm Identifier(RTT)
10         else
11             remove deleted entries
12             FT_size ← NoAF + 1
```

Figure 4.1. Flow table fingerprinting algorithm.

Consequently, the "REJECT" counter is incremented. Nevertheless, it is imperative to acknowledge that timeouts may also arise from network instabilities. Thus, prior to definitively attributing the replacement algorithm as "REJECT", multiple validations are executed to mitigate the impact of variations originating from network fluctuations (Line 3). If the "REJECT" counter surpasses the prescribed iterationCount (τ), the decision is that the replacement algorithm is unequivocally designated as "REJECT."

In the event that the RTT does not equate to *maxValue* and the *algorithmCounter* stands at zero (Line 6), the generation of new flows is terminated (Line 7). Subsequently, at Line 8, a validation ensues to determine whether the new flow has been generated. These checks are imperative due to the iterative nature of the replacement algorithm identifier, leading to diverse execution paths.

Data: RTT, iterationCount (τ)

Result: RA

```
1  if RTT == maxValue then
2      REJECT ← REJECT + 1
3      if REJECT > τ then
4          RA ← "REJECT" and return
5  else if algorithmCounter == 0 then
6      sendNewFlow ← False
7  if not (sendNewFlow) then
8      if some of our entries are deleted and these entries have shortest
         remaining time for flow update then
9          LRU ← LRU + 1
10         if LRU > τ then
11             RA ← "LRU" and T_residual ← T_idle
12             sendNewFlow ← True
13     else
14         algorithmCounter ← algorithmCounter + 1
15         if algorithmCounter > τ then
16             sendNewFlow ← True
17 else
18     if some of our entries are deleted and these entries are in order then
19         FIFO ← FIFO + 1
20         if FIFO > τ then
21             RA ← "FIFO" and sendNewFlow ← False
22     else
23         RANDOM ← RANDOM + 1
24         if RANDOM > τ then
25             RA ← "RANDOM"
```

Figure 4.2. Replacement algorithm identifier.

The halting of new flow generation is essential, as, in the scenario of a full table with a replacement algorithm other than "REJECT", the deletion of a flow becomes imperative. In the case of "LRU" as RA, it is highly probable that one of our flows will be deleted since we update them conservatively. For "FIFO", the likelihood is elevated that a benign flow will be deleted, given their earlier installation. In the case of "RANDOM", the probability of the deleted flow belonging to either our flows or benign ones is contingent upon the ratio of $FT_{residual}$ and FT_{size}.

On Line 8, given its straightforward nature, the next step unfolds with "LRU". Upon the deletion of some of our active flows (this information is derived from the flow updater process), analysis is directed towards ascertaining whether these deleted flows also possess the shortest remaining update time (Line 9). If this condition is met, the LRU counter is incremented; otherwise, the *algorithmCounter* is incremented. RA is not definitively assigned as "LRU", considering the possibility of it being "RANDOM" where deleted flows are coincided by chance. Consequently, a check is instituted to confirm that the LRU counter exceeds the stipulated iterationCount (τ) (Line 11). If this criterion is fulfilled, the replacement algorithm (RA) is set as "LRU" (Line 12). Concurrently, the new flow generation is started (Line 13), aiming at the removal of pre-existing benign rules. To facilitate this, $T_{residual}$ is established as T_{idle} (Line 14), ensuring the periodic update of our flows at each *flowUpdatePeriod* which makes our flows less likely to be selected for removal.

If *algorithmCounter* exceeds τ (Line 17), the generation of new flows recommences (Line 18). In this scenario, RA is either "FIFO" or "RANDOM". Therefore, additional flow generation is indispensable to forcefully displace benign flows. The process includes an examination to ascertain if some of our active flows are deleted and if the deletions adhere to a specific order (Line 20). If so, the FIFO counter is incremented (Line 21); otherwise, the RANDOM counter is increased (Line 26). Once again, "FIFO" is not explicitly assigned to RA, as it may also be "RANDOM", where the deletion occurrences are happenstance. Consequently, a verification is conducted to confirm that the FIFO counter surpasses τ (Line 22). If so, RA is set as "FIFO" (Line 23).

Besides, the generation of new flows is ceased (Line 24) since the table is full and FIFO is started to choose our flow entries for deletion. Similarly, an examination is conducted to determine if the RANDOM counter exceeds τ (Line 27). If affirmative, RA is configured as "RANDOM" (Line 28). Notably, the generation of new flows is not halted, as benign flows may still be present.

Upon determining the replacement algorithm (RA), the initiation of flow table size fingerprinting unfolds. Deleted flows are discarded and excluded from updates (Line 14). This approach enables the preservation of solely the active flows, mitigating the need for further replacements. Subsequently, $\mathrm{FT}_{\mathrm{size}}$ is designated as the count of active flows (Line 15).

Our algorithm is engineered to both fingerprint the network and execute saturation attacks simultaneously in a single run. The attacker can perpetuate this process, sustaining saturation attacks for an extended duration as desired.

4.5. Lightweight Defense against Flow Table Fingerprinting and Saturation Attacks (*LIDISA*)

Flow table fingerprinting is crucial for orchestrating efficient DDoS attacks against an SDN controller. Through the discernment of flow table size and replacement algorithm, the attacker can sustain flow rules within the table with optimal resource allocation. Acquiring insights into the flow table's utilization provides a metric for evaluating the effectiveness of the attack in terms of obstructing legitimate flows.

The efficient management of flow tables stands as an active area of research, as exemplified by works such as [67, 70]. However, these endeavors primarily concentrate on enhancing the efficiency of flow tables rather than fortifying their security. Consequently, an attacker can still exploit SDN fingerprinting to orchestrate saturation attacks. Some works, such as [12, 18], offer theoretical guidance on actions to take when the flow table reaches full capacity or how to prevent flow table overflow but lack

practical implementations or actionable insights.

In [71], the authors prioritize flows based on evaluation scores to counter saturation attacks. The "importance" field of malicious flows is assigned lower values, with evaluation scores determined by the volume of packet-in instances originating from a specific source address within a given time frame. Conversely, [67] proposes the random eviction of flow rules when flow table utilization surpasses a predefined threshold.

In [72], akin to the approach in [67], flow rules are evicted when TCAM utilization exceeds a predetermined threshold. The selection of flow rules for removal is based on historical flow-removed messages data, where flows that have been removed and subsequently reinstalled after an extended period are candidates for removal. These defensive mechanisms, however, do not effectively thwart flow table fingerprinting or saturation attacks, as they become operational only when the table is full or nearing full capacity. Additionally, a learning attacker employing methodologies like those illustrated in [11] could potentially bypass these mechanisms.

For a defense solution, three distinct success criteria are specified: the prevention of flow table fingerprinting, the mitigation of flow table saturation and compatibility to the SDN with a lightweight mechanism. To counteract fingerprinting, we develop a lightweight randomization mechanism, introducing complexity and uncertainty into the network. This approach necessitates increased investments from potential attackers, rendering the network more challenging to explore. Through randomization, data associated with the flow table become a moving target, impeding attackers from predicting network operations and compelling them to expend additional packets and time. The absence of knowledge regarding flow table operations also extends the time and packet requirements for saturation attacks, thereby elevating the likelihood of detection by the defender. This method strategically leverages network statistics while applying randomization to flow table operations, introducing higher levels of unpredictability for attackers while maintaining minimal impact on network functionality.

Our defense proposal in this work embodies an SDN-friendly mechanism, as it can be implemented without necessitating alterations to SDN components or protocols. Given that it randomizes flow rule operations based on basic statistics, its impact on performance is negligible. Moreover, the mechanism is executed by a background thread to prevent any increase in flow setup latencies.

The core concept of *LIDISA* revolves around proactive eviction by the controller. Proactive evictions serve as a valuable tool for evading flow table saturation. However, the eviction of an active flow carries the risk of undesired consequences and degradation of network performance. To mitigate this, evictions should be as infrequent as possible, a goal easily attained by evicting only when utilization of the flow table surpasses an acceptable threshold. Additionally, the selection of the flow to be evicted must be executed judiciously, ideally targeting malicious ones. Evicting such flows serves to save TCAM space without adversely affecting legitimate users.

4.5.1. Design and Operation

The defense parameters of *LIDISA* are presented in Table 4.3. In our solution, the defense module periodically checks the occupancy of the flow table. We measure the number of flow rules by counting `packet-in` and `flow-removed` messages instead of relying on statistic message usage. This approach ensures that the resources of the controller, switches and even the control channel are not wasted. Upon surpassing a designated threshold, proactive rule deletion mechanism is initiated. However, relying on a fixed threshold presents inflexibility and susceptibility to inference by a learning attacker. To address this concern, we introduce randomness to the threshold in each period using the randomization coefficient (β). This randomization ensures dynamic variations in flow table operations, rendering it challenging for even a learning attacker to adapt to these changes. Our mechanism randomly selects the $\text{FTU}_{\text{threshold}}$ value in every period. The formula can be expressed as

$$\begin{aligned} FTUB_{\text{threshold}} \times (1-\beta) < FTU_{\text{threshold}} \\ < FTUB_{\text{threshold}} \times (1+\beta). \end{aligned} \tag{4.1}$$

Table 4.3. Parameters for *LIDISA* defense mechanism.

Parameters	**Explanation**
$\mathbf{FTU_{threshold}}$	Flow table utilization threshold.
$\mathbf{FTUB_{threshold}}$	Flow table utilization base threshold.
Randomization Coefficient (β)	Randomization coefficient.
$\mathbf{FTR_{Deletion}}$	Ratio of flow rules for deletion.
Nsf	Number of the selected flows.

For proactive rule eviction, the selection and deletion of certain rule entries are imperative. The objective is to target malicious entries, guided by two key differentiators between normal traffic and attack traffic: the frequency of new flows and the packet count per flow, as noted in [71]. In the pursuit of minimizing attack costs, attackers often opt for a minimal Packet-Per-Second (PPS) ratio, aiming to maximize the impact with fewer packets. To identify malicious flows, we utilize Openflow aggregate flow statistics mechanism to gather information on the number of packets and the duration of flows. The flows are then ranked based on PPS metric, and we select flows with the lowest PPS. The number of selected flows, Nsf, is determined as

$$Nsf = FTU_{\text{threshold}} \times FTR_{\text{Deletion}}. \tag{4.2}$$

In this manner, our objective is to exclude normal flows by focusing solely on mice flows. This approach also mitigates the inadvertent impact of unintentionally selecting legitimate flows. To exemplify our mechanism, let's assume that $\text{FTUB}_{\text{threshold}}$, β and $\text{FTR}_{\text{Deletion}}$ are set at 90%, 0.05 and 0.1, respectively. Additionally, FT_{size} is set as 1000. Therefore, $\text{FTU}_{\text{threshold}}$ can fall within the range of 85.5% and 94.5%. Basically, the $\text{FTUB}_{\text{threshold}}$ is multiplied by the randomization coefficient, and the result is added/subtracted to find the range of $\text{FTU}_{\text{threshold}}$. Suppose in a given period, the flow table has 900 flows, and then $\text{FTU}_{\text{threshold}}$ is assigned as 880. We gather the statistics and subsequently choose $880 * 0.1 = 88$ flows with the least PPS and proceed to delete them.

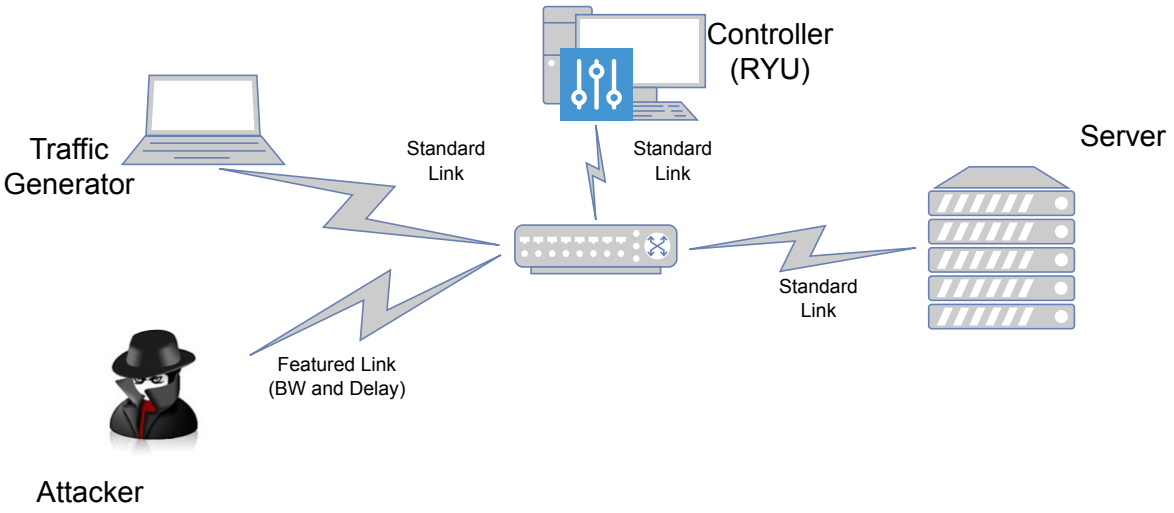


Figure 4.3. Simulation topology for *TASOS* and *LIDISA*.

4.6. Simulations and Performance Evaluation

The topology employed in the experiments is depicted in Figure 4.3. Our configuration encompasses two hosts, serving as a traffic generator and a server, a controller and a single switch. To emulate hosts and switches, Mininet [57] is chosen for its flexibility in configuring link parameters such as bandwidth and delay. RYU [58] serves as the SDN controller. The proposed algorithms are implemented in Python. For simulations and RTT measurements, *TASOS* utilizes UDP packets. However, any other network tool, such as `ping`, can also be employed. Simulations are executed on a computer equipped with an Intel i7-6500U 2.5 GHz (2 cores) processor and 16GB RAM.

4.6.1. Fingerprinting and Saturation Experiments

In this section, we conduct experiments to deduce the flow table information. We assume that the network type has already been fingerprinted and we are aware that it is an SDN with known flow rule timeouts [8]. To comprehensively assess the impact of network parameters and explore the robustness of our solution, we generate the same topology with different characteristics. Thus, we investigate the fingerprinting success rate concerning link delay, bandwidth and background traffic (BT). Network and algorithm parameters for the fingerprinting simulation are listed in Table 4.4 and

Table 4.5, respectively.

Table 4.4. Network parameters.

Network Parameters	
Link Delay (ms)	0, 50, 100
Bandwidth (Mbps)	10, 100, 1000
BT (Mbps)	10, 100, 1000

Table 4.5. Algorithm parameters.

Parameter	**Value**
iterationCount (τ)	3
flowCountPerSecond	1
$\mathbf{T_{idle}}$	10
flowUpdatePeriod	1
$\mathbf{FT_{size}}$	1000
$\mathbf{T_{residual}}$	1.5
$\mathbf{FT_{residual}}$	500
RA	REJECT

There are 27 distinct network conditions and each condition is simulated 100 times to analyze accuracy. The network parameters (link delay and bandwidth) are applied to the link of the attacker (Featured Link in the Figure 4.3). For background traffic, a traffic generator generates traffic for the server. The algorithm parameters remain constant as we aim to observe the effect of the network parameters. T_{idle} is set to 10 seconds, as idle timeout is typically configured within the range of 9 to 11 seconds [24].

Table 4.6. FT_{size} accuracy for different network parameters.

(BW, BT)	**LD = 0**	**LD = 50**	**LD = 100**
(10,10)	1	1	1
(10,100)	1	1	1
(10,1000)	1	1	1
(100,10)	1	1	0.99
(100,100)	1	1	1
(100,1000)	1	1	1
(1000,10)	1	1	0.99
(1000,100)	1	1	1
(1000,1000)	1	1	0.99

flowCountPerSecond is set to 1 to demonstrate that a low-rate packet-in generation is still sufficient for saturation attacks. The benign flows are initiated at the onset of the attack. They are continually updated until the flow table reaches full capacity. Subsequently, they are gradually displaced by malicious flows or evicted due to timeouts.

The accuracy of *TASOS* for flow table size fingerprinting under different network parameters is presented in Table 4.6. *TASOS* consistently identifies the flow table utilization and replacement algorithm correctly. Highlighted cells indicate results that differ from 1. The overall success ratios of the simulations range between 0.99 and 1.0. Notably, as the link delay increases, the success probability appears to be negatively impacted. This observation is significant, as a larger link delay limits the effect of the flow setup delay. Although RTT also increases in such scenarios, the flow setup time for the network remains constant. Bandwidth and background traffic appear to have a negligible impact on the success ratio of *TASOS* similar to result shown in Chapter 3.

Table 4.7. Algorithm parameters.

Algorithm Parameters	
$\mathbf{FT_{size}}$	100, 500, 1000
$\mathbf{FT_{residual}}$	50%, 75%
RA	LRU, FIFO, REJECT, RANDOM

TASOS exhibits superior accuracy under more challenging network conditions compared to related works ([18–20, 22]). For scalability analysis, we conduct 27 different simulations to examine the effects of background traffic, link delay and bandwidth. Notably, [18–20, 22] do not consider link delay or bandwidth in their experiments and only [22] examines the effect of background traffic, which severely degrades its performance.

Subsequently, to understand the impact of certain algorithm parameters, we conduct additional experiments using maximum values for the network parameters for a marginal case. Thus, link delay, bandwidth and BT are set to 100 ms, 1000 Mbps and 1000 Mbps, respectively. Algorithm parameters for this simulation are detailed in Table 4.7. $\mathrm{FT_{residual}}$ is established as a ratio of $\mathrm{FT_{size}}$. There are 24 different conditions and each condition is simulated 100 times to assess the accuracy performance.

Similar to the network parameter experiments, the success ratio for flow table utilization and replacement algorithm is 1.0. The accuracy of *TASOS* for fingerprinting with different algorithm parameters is presented in Table 4.8.

The success ratios of REJECT, LRU and FIFO are consistently high, ranging between 0.98 and 1.0. RANDOM exhibits lower success ratios, as, after the flow table is filled with our flows, it is probable that one of our flows is evicted. Consequently, the flow table size may be inaccurate if the simulation duration concludes before determining the exact table size. However, we generally determine the table size with a margin of error of minus 5%. Another noteworthy finding is that the flow table size or

utilization rate does not significantly affect the success ratio of our algorithms.

We do not exclusively perform simulations for saturation attacks, as accurately fingerprinting flow table parameters necessitates executing saturation attacks. These parameters cannot be determined without saturating the flow table. Consequently, the efficiency of the saturation attacks mirrors the results related to flow table size. Once fingerprinting is completed, the attack can proceed with greater efficiency.

In simulations, *TASOS* operates at a low rate, with *flowCountPerSecond* set to 1. Given that common detection systems rely on packet-in message rates for identifying saturation attacks, *TASOS* can effectively evade these detection mechanisms.

In [18], only the controller's reaction to the table full event (REJECT or not) is inferred and there is no accuracy calculation. In [19, 20], flow table size and utilization are estimated with error margins of 10% and 1%, respectively, without determining the exact values. Moreover, both [19, 20] assume that replacement algorithm is given. Additionally, [20] only considers LRU and FIFO, while [19] considers just FIFO.

Once again, *TASOS* demonstrates superior accuracy under more challenging network conditions compared to related works ([18–20, 22]). However, these comparisons are naturally contingent on the best available knowledge regarding the design and performance of those algorithms by the respective authors and for different (simulated or real) network environments. It is essential to note that, for instance, *TASOS* might be operating in a "more favorable" network environment.

Another crucial consideration is the cost associated with fingerprinting and saturation attacks. Our algorithm is designed to execute flow table fingerprinting and saturation attacks simultaneously. Moreover, *TASOS* acquires all necessary information, including flow table size, utilization rate, and replacement algorithm. Notably, there is no single related work that comprehensively covers all these data. Furthermore, *TASOS* introduces additional replacement algorithms for fingerprinting.

Table 4.8. Flow table size fingerprinting accuracy results.

(FT_{size}, $FT_{residual}$)	**REJECT**	**FIFO**	**LRU**	**RANDOM**
(100,50%)	1	1	0.99	0.98
(100,75%)	0.99	0.98	1	0.98
(500,50%)	1	1	1	0.97
(500,75%)	1	0.98	0.99	0.97
(1000,50%)	0.98	1	1	0.94
(1000,75%)	0.99	0.99	0.99	0.93

4.6.2. Fingerprinting Countermeasure Simulations

We evaluate *LIDISA*, which employs proactive eviction to counteract fingerprinting and saturation attacks. In simulations, $FTUB_{threshold}$, β and $FTR_{Deletion}$ are set at 90%, 0.05 and 0.1, respectively. Additionally, FT_{size} and $FT_{residual}$ are set as 1000 and 500, respectively. The remaining parameters are the same as those in Table 4.7.

In the simulations, we execute both our attack and defense modules. Our attack algorithm consistently provides incorrect estimates for flow table size and utilization. The replacement algorithm is occasionally correctly identified by chance; however, it is predominantly deduced as RANDOM.

To assess the impact of our solution on performance, *TASOS* initiates attacks on the network with the defense mechanism being both active and inactive. We monitor average response time to packet-in, CPU usage and memory usage metrics to gauge the effect of our defense solution. The performance simulation is conducted 100 times, with 50 of them featuring an active defense solution. Our defense module solely compares the active flow count with $FTU_{threshold}$ during normal operations, resulting in minimal impact. In an attack scenario, it gathers statistics and deletes flows based on these statistics. According to the simulations, the average response time for `packet-in` increases by 3.5%, rising from 5.45 to 5.64 milliseconds during attack times. Similarly,

CPU and memory usage experience increase of 6.54% and 1.86%, respectively. These results indicate that the proposed method has a slight impact on performance even during attack times while effectively preventing fingerprinting and saturation attacks.

4.7. Discussion

SDN has become an integral part of today's ICT system, finding applications in various segments of networking and telecommunications infrastructure. However, since its widespread deployment in controlled and restricted environments such as data centers and core networks, security concerns within the realm of SDN are often relegated to academic discussions. Moreover, the perceived performance implications of security countermeasures are often cited as justifications for less risk-aware treatments of potential threats in this domain.

Traditional flooding DDoS attacks and SDN saturation attacks share similarities but differ in several aspects. Notably, if the connection between the controller and switches is out-of-band (as is often the case), traditional flooding DDoS attacks cannot be executed by simply flooding the link. Additionally, flooding attacks are generally more brute-force in nature, whereas saturation attacks employ more sophisticated methods. For example, saturation attacks typically require far less bandwidth compared to traditional DDoS attacks (measured in Kbps versus Gbps).

The duration of the SDN fingerprinting process varies based on network characteristics, fingerprinting algorithm parameters and the execution method of the algorithms. The parameter most dominant in affecting duration is *flowCountPerSecond*. Its careful assignment is crucial, as large values can risk detection of the attack, while small values can extend the attack's duration. This extension also increases the required number of packets for fingerprinting, given that the flow update process can generate numerous refreshing packets. Therefore, there exists an implicit trade-off between *flowCountPerSecond*, duration and the cost of the attack. An interesting research problem could involve finding the optimal value for *flowCountPerSecond* according to different

network characteristics.

Another critical consideration is the dependency of fingerprinting accuracy on several factors, including the attack's duration, the chosen replacement algorithm, and the characteristics of benign flows (such as volume, duration, and rate). For instance, if the replacement algorithm is set to REJECT and certain legitimate flows never expire due to timeouts, flow table size fingerprinting may yield incorrect results. A similar effect occurs when there are permanent flows (where both flow timeouts (hard and idle) are not set). Moreover, the rate at which new benign flows are generated can significantly impact the algorithm's accuracy. When the replacement algorithm is RANDOM or FIFO, the algorithm may struggle to accurately determine the true flow table size, especially if the rate of benign flows exceeds the *flowCountPerSecond* parameter. In the case of RANDOM, as the flow table fills with malicious flows, the probability increases that one of these malicious flows will be selected for eviction. Consequently, evacuating all legitimate flows may take a considerable amount of time or even fail altogether, leading to inaccurate estimations of flow table size. However, even in such scenarios, the saturation attack can still be successful as the network fails to handle legitimate flow installation requests continuously. Additionally, the eviction of benign flows will degrade network performance. Exploring the fingerprinting of the rate of new benign flows and adjusting the *flowCountPerSecond* parameter accordingly could be an intriguing research avenue to enhance the accuracy and efficiency of fingerprinting and saturation attacks. Thus, there is an implicit trade-off between accuracy and overhead for the attacker.

The algorithm may encounter prolonged waiting periods in scenarios where the table reaches full capacity (Line 8 in Algorithm 2), the RTT is not *maxValue* (Line 1 in Algorithm 3), and benign flows are evicted due to timeouts. In such cases, where the algorithm ceases to send new flows and there is still available space in the flow table, it must await the gradual filling of the table with benign flows, a process that may take noticeable time. In that case, attacker can reactivate new flow creation manually, or the algorithm can make it automatically.

If there are simultaneous attackers attempting to perform fingerprinting and/or saturation attacks, they likely diminish each other's accuracy and effectiveness since each attacker tries to evict the other's flows to saturate the flow table. If the attackers collaborate, it can be considered as a single attacker utilizing multiple machines. We also anticipate that our defense mechanism remains unaffected by the number of hosts or attackers, as it primarily monitors flow table occupancy and evicts the least used flows when necessary. However, our mechanism may evict more benign flows in the presence of multiple simultaneous attackers. This presents an intriguing avenue for future research.

With our proposed defense mechanism, *LIDISA*, our goal isn't to completely eliminate saturation attacks but rather to increase the difficulty for the attacker. Presently, attackers can accurately deduce $\mathrm{FT}_{\mathrm{size}}$ and $\mathrm{FT}_{\mathrm{residual}}$ with relative ease and reliability. Even if the attacker is aware of our defense algorithm, *LIDISA* introduces an element of uncertainty, as these parameters are approximated based on random thresholds that the attacker cannot predict. Furthermore, attackers can evict benign flows without expending significant traffic. However, with *LIDISA*, attackers must exceed an unknown flow rate to evict these flows. If attackers are willing to invest additional resources by sending more packets, they can evict more active flows. Therefore, our objective of preventing fingerprinting and mitigating saturation attacks is achieved by increasing their costliness.

The performance of our algorithm, *LIDISA*, can be influenced by the characteristics of benign flows (e.g., volume, duration, and rate). If numerous benign flows exhibit significantly low rates, *LIDISA* may select these benign flows for eviction. However, since these flows do not carry substantial traffic, their eviction has a minimal impact on overall performance.

The flow table fingerprinting algorithm relies on the idle timeout in the flow update process. Hence, an error in the idle timeout can lead to successful fingerprinting with additional packets or failure of the fingerprinting process. For example, if the

actual idle timeout is 10 seconds and the calculated one is 5 seconds, the fingerprinting can still yield correct results but will require significantly more packets, as the flow update process will need to run twice as often. However, if the calculated idle timeout is 20 seconds, the flow table fingerprinting will likely fail.

4.8. Conclusions

While there is a significant body of work utilizing SDN as a security enabler, the security of SDN itself remains a critical concern. In this work, we introduce the concept of SDN flow table fingerprinting and the type of information that can be extracted from it. Additionally, we address saturation attacks against SDN flow tables.

We propose *TASOS*, an efficient and automated fingerprinting agent capable of inferring the flow table size, utilization rate and replacement algorithm. Subsequently, using the fingerprint data, *TASOS* proceeds to execute efficient saturation attacks. *TASOS* achieves high accuracy even in challenging network conditions. Extensive simulations have been conducted across a diverse range of network parameters, demonstrating that *TASOS* performs exceptionally well in those scenarios. We also present a lightweight proactive eviction mechanism (*LIDISA*) that increases the cost for attackers, making it more challenging for them to explore and exploit the network.

As a potential avenue for future research, machine learning algorithms, particularly deep learning, could be employed to detect and mitigate SDN fingerprinting techniques. Exploring defense solutions aiming to blur the difference between T_1 and T_2 could also be an intriguing research idea.

5. CONCLUSION

SDN introduces promising opportunities to address the inherent security vulnerabilities of traditional networks. Nevertheless, it also introduces new challenges, particularly vulnerabilities in data and control plane communications. This thesis presents a security framework aimed at mitigating security concerns surrounding data exchange among SDN entities, including controllers, switches, and applications.

Our focus in this thesis has been on enhancing the security of SDN communication channels. We have examined three key communication interfaces within SDN: the Southbound (SB) interface, the East-West Bound (EWB) interface, and the Northbound (NB) interface. While the SB interface primarily relies on OpenFlow protocol, we have identified the need for TLS with mutual authentication to ensure confidentiality and integrity of communication. However, mere deployment of TLS is not sufficient; it requires rigorous hardening and mutual authentication to address potential security flaws and obsolete versions. Additionally, ACLs are implemented to regulate network element access, thereby strengthening communication security.

To comprehensively address these security challenges, we introduced the *Secure Communication Architecture for Software-Defined Networks (SECAS). SECAS* includes a cryptographic key generation model and an accompanying SDN application for key material generation and management. It also issues and manages X.509 certificates to ensure secure communication, with specialized certificate fields to mitigate the impact of private key hijacking. Real-time security alarms are dispatched for security breaches such as unauthorized access requests, and TLS usage in each interface can be managed or monitored separately via a user interface.

Furthermore, we recognized the vulnerability of SDN switches to flow table saturation attacks, which exploit communication with the controller to overwhelm flow table capacities. Our proposed SDN fingerprinting modules, *Robust and Automated*

Fingerprinting Agent for SDN (RAFA) and *Table Analyzer and Saturation Orchestrator for SDN (TASOS)*, not only identify SDN networks but also infers critical network characteristics such as flow timeout values (hard and idle) and flow table attributes (flow table utilization rate, size and replacement algorithm). This enables efficient execution of saturation attacks while ensuring high accuracy across diverse network conditions. Additionally, we propose two lightweight defense mechanisms, *Dynamic Adaptive Flow Timeout Randomization (DAFTR)* and *Lightweight Defense against Flow Table Fingerprinting and Saturation Attacks (LIDISA)*, with the aim of randomizing rule timeouts and proactive flow rule deletion with respect to network status to counteract such attacks.

In summary, our contributions encompass the development of SECAS for secure communication, *RAFA* and *TASOS* for SDN fingerprinting and flow table saturation attack, and *DAFTR* and *LIDISA* for defense against mentioned attacks. Through extensive simulations, we have validated the efficacy and adaptability of our proposed solutions under various network conditions.

As future research directions, the security module of *SECAS* can be extended to automatically detect and revoke certificates of malicious network devices. Additionally, a more automated key distribution and regeneration method for SDN could further enhance *SECAS* efficiency.

Improvements for *RAFA* and *TASOS* could involve the implementation of more efficient algorithms, such as incorporating binary search methods while discovering flow rule timeouts. This optimization would result in fewer packets and less time spent during the discovery process, consequently reducing the likelihood of detection as an attacker. Additionally, leveraging machine learning algorithms could enhance the extraction of diverse information from SDN networks through fingerprinting techniques. Exploring optimal values for parameters like *flowCountPerSecond* based on various network characteristics presents another avenue for enhancement. Furthermore, the pursuit of multi-domain SDN fingerprinting could broaden the scope of network analysis

and security assessment.

Regarding *DAFTR* and *LIDISA*, potential improvements include randomizing additional network parameters and exploring defense solutions aiming to blur the difference between T_1 and T_2.

Looking ahead, our thesis lays the groundwork for future research in novel defense strategies and enhancing network security against evolving threats. We hope that our work contributes to the continued advancement of secure SDN communication and network resilience.

REFERENCES

1. Ahmad, S. and A. H. Mir, "Protection of Centralized SDN Control Plane from High-Rate Packet-In Messages", *International Journal of Information Security*, Vol. 22, No. 5, pp. 1197–1206, 2023.

2. Deb, R. and S. Roy, "A Comprehensive Survey of Vulnerability and Information Security in SDN", *Computer Networks*, Vol. 206, p. 108802, 2022.

3. Yigit, B., G. Gur, B. Tellenbach, and F. Alagoz, "Secured Communication Channels in Software-Defined Networks", *IEEE Communications Magazine*, Vol. 57, No. 10, pp. 63–69, 2019.

4. Correa Chica, J. C., J. C. Imbachi, and J. F. Botero Vega, "Security in SDN: A Comprehensive Survey", *Journal of Network and Computer Applications*, Vol. 159, p. 102595, 2020.

5. Maleh, Y., Y. Qasmaoui, K. El Gholami, Y. Sadqi, and S. Mounir, "A Comprehensive Survey on SDN Security: Threats, Mitigations, and Future Directions", *Journal of Reliable Intelligent Environments*, Vol. 9, No. 2, pp. 201–239, 2023.

6. Ayodele, B. and V. Buttigieg, "SDN as a Defence Mechanism: a Comprehensive Survey", *International Journal of Information Security*, Vol. 23, No. 1, pp. 141–185, 2024.

7. Kaur, S., A. K. Sandhu, and A. Bhandari, "Investigation of Application Layer DDoS Attacks in Legacy and Software-Defined Networks: A Comprehensive Review", *International Journal of Information Security*, Vol. 22, No. 6, pp. 1949–1988, 2023.

8. Yiğit, B., G. Gür, F. Alagöz, and B. Tellenbach, "Network Fingerprinting via Timing Attacks and Defense in Software Defined Networks", *Computer Networks*,

Vol. 232, p. 109850, 2023.

9. Sezer, S., S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are We Ready for SDN? Implementation Challenges for Software-Defined Networks", *IEEE Communications Magazine*, Vol. 51, No. 7, pp. 36–43, 2013.

10. Kreutz, D., F. M. Ramos, and P. Verissimo, "Towards Secure and Dependable Software-Defined Networks", *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN, pp. 55–60, New York, USA, 2013.

11. Shen, Y., C. Wu, D. Kong, and Q. Cheng, "Flow Table Saturation Attack against Dynamic Timeout Mechanisms in SDN", *Applied Sciences*, Vol. 13, No. 12, p. 7210, 2023.

12. Zhou, Y., K. Chen, J. Zhang, J. Leng, and Y. Tang, "Exploiting the Vulnerability of Flow Table Overflow in Software-Defined Network: Attack Model, Evaluation, and Defense", *Security and Communication Networks*, Vol. 2018, pp. 1–15, 2018.

13. Guo, Z., R. Liu, Y. Xu, A. Gushchin, A. Walid, and H. J. Chao, "STAR: Preventing Flow-Table Overflow in Software-Defined Networks", *Computer Networks*, Vol. 125, pp. 15–25, 2017.

14. Klöti, R., V. Kotronis, and P. Smith, "OpenFlow: A Security Analysis", *21st IEEE International Conference on Network Protocols (ICNP)*, pp. 1–6, Goettingen, 2013.

15. Shin, S. and G. Gu, "Attacking Software-Defined Networks: A First Feasibility Study", *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, p. 165–166, Association for Computing Machinery, New York, NY, USA, 2013.

16. Cui, H., G. O. Karame, F. Klaedtke, and R. Bifulco, "On the Fingerprinting of

Software-Defined Networks", *IEEE Transactions on Information Forensics and Security*, Vol. 11, No. 10, pp. 2160–2173, 2016.

17. Khorsandroo, S. and A. S. Tosun, "Time Inference Attacks on Software Defined Networks: Challenges and Countermeasures", *IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 342–349, San Francisco, CA, USA, 2018.

18. Ahmed, B., N. Ahmed, A. W. Malik, M. Jafri, and T. Hafeez, "Fingerprinting SDN Policy Parameters: An Empirical Study", *IEEE Access*, Vol. 8, pp. 142379–142392, 2020.

19. Gregorczyk, M. and W. Mazurczyk, "Inferring Flow Table State through Active Fingerprinting in SDN Environments: A Practical Approach", *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT*, pp. 576–586, Online, 2021.

20. Leng, J., Y. Zhou, J. Zhang, and C. Hu, "An Inference Attack Model for Flow Table Capacity and Usage: Exploiting the Vulnerability of Flow Table Overflow in Software-Defined Network", *arXiv preprint arXiv:1504.03095*, 2015.

21. Zhang, M., G. Li, L. Xu, J. Bai, M. Xu, G. Gu, and J. Wu, "Control Plane Reflection Attacks and Defenses in Software-Defined Networks", *IEEE/ACM Transactions on Networking*, Vol. 29, No. 2, pp. 623–636, 2020.

22. Yu, M., T. Xie, T. He, P. McDaniel, and Q. K. Burke, "Flow table security in SDN: Adversarial reconnaissance and intelligent attacks", *IEEE/ACM Transactions on Networking*, Vol. 29, No. 6, pp. 2793–2806, 2021.

23. Cao, J., M. Xu, Q. Li, K. Sun, Y. Yang, and J. Zheng, "Disrupting SDN via the Data Plane: A Low-Rate Flow Table Overflow Attack", *Security and Privacy in Communication Networks: 13th International Conference, SecureComm, Niagara Falls, ON, Canada, October 22–25, Proceedings 13*, pp. 356–376, Niagara Falls,

ON, Canada, 2018.

24. Pascoal, T. A., I. E. Fonseca, and V. Nigam, "Slow Denial-of-Service Attacks on Software Defined Networks", *Computer Networks*, Vol. 173, p. 107223, 2020.

25. Samociuk, D., "Secure Communication between OpenFlow Switches and Controllers", *Proceedings of the 7th International conference on Advances in Future Internet (AFIN).*, Vol. 39, 2015.

26. Lam, J. H., S.-G. Lee, H.-J. Lee, and Y. E. Oktian, "TLS Channel Implementation for ONOS's East/West-Bound Communication", *Electronics, Communications and Networks V: Proceedings of the 5th International Conference on Electronics, Communications and Networks (CECNet 2015)*, pp. 397–403, Shanghai, China, 2016.

27. Lam, J., S.-G. Lee, H.-J. Lee, and Y. E. Oktian, "Securing SDN Southbound and Data Plane Communication with IBC", *Mobile Information Systems*, Vol. 2016, 2016.

28. Kang, J. W., S. H. Park, and J. You, "Mynah: Enabling Lightweight Data Plane Authentication for SDN Controllers", *24th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–6, Las Vegas, USA, 2015.

29. Agborubere, B. and E. Sanchez-Velazquez, "OpenFlow Communications and TLS Security in Software-Defined Networks", *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 560–566, Exeter, UK, 2017.

30. Liyanage, M., M. Ylianttila, and A. Gurtov, "Securing the Control Channel of Software-Defined Mobile Networks", *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pp. 1–6, 2014.

31. Othman, O. M. and K. Okamura, "Securing Distributed Control of Software De-

fined Networks", *International Journal of Computer Science and Network Security (IJCSNS)*, Vol. 13, No. 9, p. 5, 2013.

32. Cui, H., Z. Chen, L. Yu, K. Xie, and Z. Xia, "Authentication Mechanism for Network Applications in SDN Environments", *20th International Symposium on Wireless Personal Multimedia Communications (WPMC)*, pp. 1–5, Yogyakarta, Indonesia, 2017.

33. Paladi, N. and C. Gehrmann, "TruSDN: Bootstrapping Trust in Cloud Network Infrastructure", *International Conference on Security and Privacy in Communication Systems*, pp. 104–124, Cham, 2016.

34. Qin, H. and N. Wang, "A Data-origin Authentication Protocol Based on ONOS Cluster", *MATEC Web of Conferences*, Vol. 56, 2016.

35. Peng, Y., C. Wu, B. Zhao, W. Yu, B. Liu, and S. Qiao, "QKDFlow: QKD based Secure Communication towards the OpenFlow Interface in SDN", *4th International Conference on Geo-Informatics in Resource Management and Sustainable Ecosystems GRMSE*, pp. 410–415, Hong Kong, China, 2016.

36. Toseef, U., A. Zaalouk, T. Rothe, M. Broadbent, and K. Pentikousis, "C-BAS: Certificate-Based AAA for SDN Experimental Facilities", *Third European Workshop on Software Defined Networks*, pp. 91–96, Budapest, Hungary, 2014.

37. Banse, C. and S. Rangarajan, "A Secure Northbound Interface for SDN Applications", *IEEE Trustcom/BigDataSE/ISPA*, Vol. 1, pp. 834–839, Helsinki, Finland, 2015.

38. Rafaeli, S. and D. Hutchison, "A Survey of Key Management for Secure Group Communication", *ACM Computing Surveys*, Vol. 35, No. 3, pp. 309–329, 2003.

39. Aji, A., K. Jain, and P. Krishnan, "A Survey of Quantum Key Distribution (QKD) Network Simulation Platforms", *2nd Global Conference for Advancement in Tech-*

nology (GCAT), pp. 1–8, Bangalore, India, 2021.

40. Dayal, N., P. Maity, S. Srivastava, and R. Khondoker, "Research Trends in Security and DDoS in SDN", *Security and Communication Networks*, Vol. 9, No. 18, pp. 6386–6411, 2016.

41. Azzouni, A., O. Braham, T. M. T. Nguyen, G. Pujolle, and R. Boutaba, "Fingerprinting OpenFlow Controllers: The First Step to Attack an SDN Control Plane", *IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, Washington DC, USA, 2016.

42. Wang, T., H. Chen, and J. Cao, "A Lightweight SDN Fingerprint Attack Defense Mechanism Based on Probabilistic Scrambling and Controller Dynamic Scheduling Strategies", *Security and Communication Network*, Vol. 2021, 2021.

43. Sonchack, J., A. J. Aviv, and E. Keller, "Timing SDN Control Planes to Infer Network Configurations", *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security, p. 19–22, New York, NY, USA, 2016.

44. Zhang, M., J. Hou, Z. Zhang, W. Shi, B. Qin, and B. Liang, "Fine-Grained Fingerprinting Threats to Software-Defined Networks", *IEEE Trustcom/BigDataSE/ICESS*, pp. 128–135, Sydney, Australia, 2017.

45. Hou, J., M. Zhang, Z. Zhang, W. Shi, B. Qin, and B. Liang, "On the Fine-Grained Fingerprinting Threat to Software-Defined Networks", *Future Generation Computer Systems*, Vol. 107, No. C, p. 485–497, 2020.

46. "OpenFlow Switch Specification 1.5.1", `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`, accessed on June 3, 2024.

47. Soussi, W., M. Christopoulou, G. Xilouris, and G. Gür, "Moving Target Defense

as a Proactive Defense Element for Beyond 5G", *IEEE Communications Standards Magazine*, Vol. 5, No. 3, pp. 72–79, 2021.

48. Yaacoub, J.-P. A., H. N. Noura, O. Salman, and A. Chehab, "Robotics Cyber Security: Vulnerabilities, Attacks, Countermeasures, and Recommendations", *International Journal of Information Security*, Vol. 21, No. 1, pp. 115–158, 2022.

49. Xie, F., H. Wen, J. Wu, W. Hou, H. Song, T. Zhang, R. Liao, and Y. Jiang, "Data Augmentation for Radio Frequency Fingerprinting via Pseudo-Random Integration", *IEEE Transactions on Emerging Topics in Computational Intelligence*, Vol. 4, No. 3, pp. 276–286, 2020.

50. Ding, G., Z. Tan, J. Wu, and J. Zhang, "Efficient Indoor Fingerprinting Localization Technique using Regional Propagation Model", *IEICE Transactions on Communications*, Vol. 97, No. 8, pp. 1728–1741, 2014.

51. Babun, L., H. Aksu, and A. S. Uluagac, "CPS Device-Class Identification via Behavioral Fingerprinting: From Theory to Practice", *IEEE Transactions on Information Forensics and Security*, Vol. 16, pp. 2413–2428, 2021.

52. Laštovička, M., S. Špaček, P. Velan, and P. Čeleda, "Using TLS Fingerprints for OS Identification in Encrypted Traffic", *NOMS IEEE/IFIP Network Operations and Management Symposium*, pp. 1–6, Budapest, Hungary, 2020.

53. Sonchack, J., A. Dubey, A. J. Aviv, J. M. Smith, and E. Keller, "Timing-based Reconnaissance and Defense in Software-Defined Networks", *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC, pp. 89–100, New York, NY, USA, 2016.

54. "ONOS : Open Network Operating System", `https://github.com/opennetworkinglab/onos`, accessed on June 3, 2024.

55. Haynes, W., "Student's t-test", *Encyclopedia of Systems Biology*, pp. 2023–2025,

New York, NY, 2013.

56. Cho, J.-H., D. P. Sharma, H. Alavizadeh, S. Yoon, N. Ben-Asher, T. J. Moore, D. S. Kim, H. Lim, and F. F. Nelson, "Toward Proactive, Adaptive Defense: A Survey on Moving Target Defense", *IEEE Communications Surveys & Tutorials*, Vol. 22, No. 1, pp. 709–745, 2020.

57. "Mininet", `http://mininet.org/`, accessed on June 3, 2024.

58. "RYU SDN Framework", `https://ryu-sdn.org/`, accessed on June 3, 2024.

59. "Impacket", `https://github.com/SecureAuthCorp/impacket`, accessed on June 3, 2024.

60. "OpenDaylight", `https://www.opendaylight.org/`, accessed on June 3, 2024.

61. "FloodLight Controller", `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview`, accessed on June 3, 2024.

62. Shang, G., P. Zhe, X. Bin, H. Aiqun, and R. Kui, "FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks", *IEEE INFOCOM - IEEE Conference on Computer Communications*, pp. 1–9, Atlanta, GA, USA, 2017.

63. Shin, S., V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and Vigilant Switch Flow Management in Software-Defined Networks", *Proceedings of the ACM SIGSAC conference on Computer & Communications Security*, pp. 413–424, Berlin, Germany, 2013.

64. Wang, H., L. Xu, and G. Gu, "Floodguard: A DOS Attack Prevention Extension in Software-Defined Networks", *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 239–250, Rio de Janeiro, Brazil, 2015.

65. Ambrosin, M., M. Conti, F. De Gaspari, and R. Poovendran, "Lineswitch: Tackling Control Plane Saturation Attacks in Software-Defined Networking", *IEEE/ACM Transactions on Networking*, Vol. 25, No. 2, pp. 1206–1219, 2016.

66. Huang, X.-B., K.-P. Xue, Y.-T. Xing, D.-W. Hu, R. Li, and Q.-B. Sun, "An Efficient Scheme to Defend Data-to-Control-Plane Saturation Attacks in Software-Defined Networking", *Journal of Computer Science and Technology*, Vol. 37, No. 4, pp. 839–851, 2022.

67. Vishnoi, A., R. Poddar, V. Mann, and S. Bhattacharya, "Effective Switch Memory Management in OpenFlow Networks", *Proceedings of the 8th ACM International Conference on Distributed Event-based Systems*, pp. 177–188, Mumbai, India, 2014.

68. Li, Q., N. Huang, D. Wang, X. Li, Y. Jiang, and Z. Song, "HQTimer: a Hybrid Q-Learning-Based Timeout Mechanism in Software-Defined Networks", *IEEE Transactions on Network and Service Management*, Vol. 16, No. 1, pp. 153–166, 2019.

69. Zarek, A., "Openflow Timeouts Demystified", *M.S. Thesis, University of Toronto*, 2012.

70. Zhang, L., S. Wang, S. Xu, R. Lin, and H. Yu, "TimeoutX: An Adaptive Flow Table Management Method in Software Defined Networks", *IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, San Diego, USA, 2015.

71. Zhang, M., J. Bi, J. Bai, Z. Dong, Y. Li, and Z. Li, "FTGuard: A Priority-aware Strategy against the Flow Table Overflow Attack in SDN", *Proceedings of the SIGCOMM Posters and Demos*, pp. 141–143, Los Angeles, CA, USA, 2017.

72. Shen, Y., C. Wu, Q. Cheng, and D. Kong, "AFTM: An Adaptive Flow Table Management Scheme for OpenFlow Switches", *IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International*

Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 917–922, Fiji, 2020.

73. Conti, M., F. De Gaspari, and L. V. Mancini, "A Novel Stealthy Attack to Gather SDN Configuration-Information", *IEEE Transactions on Emerging Topics in Computing*, Vol. 8, No. 2, pp. 328–340, 2020.

74. Lin, P.-C., P.-C. Li, and V. L. Nguyen, "Inferring OpenFlow Rules by Active Probing in Software-Defined Networks", *19th International Conference on Advanced Communication Technology (ICACT)*, pp. 415–420, Pyeongchang, Korea, 2017.

75. Liu, S., M. K. Reiter, and V. Sekar, "Flow Reconnaissance via Timing Attacks on SDN Switches", *IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 196–206, Atlanta, GA, USA, 2017.

76. Alshamrani, A., "Reconnaissance Attack in SDN based Environments", *27th International Conference on Telecommunications (ICT)*, pp. 1–5, Kuta, Bali, 2020.

77. Do Hoang, H., V.-H. Pham, *et al.*, "Empirical Study on Reconnaissance Attacks in SDN-aware Network for Evaluating Cyber Deception", *RIVF International Conference on Computing and Communication Technologies (RIVF)*, pp. 1–6, Hanoi, Vietnam, 2021.

78. Achleitner, S., T. La Porta, T. Jaeger, and P. McDaniel, "Adversarial Network Forensics in Software Defined Networking", *Proceedings of the Symposium on SDN Research*, pp. 8–20, Santa Clara, CA, USA, 2017.

79. Smyth, D., V. Cionca, S. McSweeney, and D. O'Shea, "Exploiting Pitfalls in Software-Defined Networking Implementation", *International Conference on Cyber Security and Protection of Digital Services (cyber security)*, pp. 1–8, London, UK, 2016.

80. Priyadarsini, M. and P. Bera, "Software Defined Networking Architecture, Traffic

Management, Security, and Placement: A Survey", *Computer Networks*, Vol. 192, p. 108047, 2021.

81. Soussi, W., M. Christopoulou, G. Gür, and B. Stiller, "MERLINS – Moving Target Defense Enhanced with Deep-RL for NFV In-Depth Security", *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 65–71, Dresden, Germany, 2023.

82. Yadav, T. and A. M. Rao, "Technical Aspects of Cyber Kill Chain", Abawajy, J. H., S. Mukherjea, S. M. Thampi, and A. Ruiz-Martínez (editors), *Security in Computing and Communications*, pp. 438–452, Cham, 2015.

83. Chen, S., H. Wen, J. Wu, A. Xu, Y. Jiang, H. Song, and Y. Chen, "Radio Frequency Fingerprint-Based Intelligent Mobile Edge Computing for Internet of Things Authentication", *Sensors*, Vol. 19, No. 16, 2019.

84. Ding, G., Z. Tan, J. Wu, J. Zeng, and L. Zhang, "Indoor Fingerprinting Localization and Tracking System using Particle Swarm Optimization and Kalman Filter", *IEICE Transactions on Communications*, Vol. 98, No. 3, pp. 502–514, 2015.

85. Qiao, S., C. Hu, X. Guan, and J. Zou, "Taming the Flow Table Overflow in OpenFlow Switch", *Proceedings of the ACM SIGCOMM Conference*, pp. 591–592, Florianopolis, Brazil, 2016.

86. Kandoi, R. and M. Antikainen, "Denial-of-Service Attacks in OpenFlow SDN Networks", *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 1322–1326, Ottowa, Canada, 2015.

87. Zhu, H., H. Fan, X. Luo, and Y. Jin, "Intelligent Timeout Master: Dynamic Timeout for SDN-based Data Centers", *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 734–737, Ottowa, Canada, 2015.

88. Qian, Y., W. You, and K. Qian, "Openflow Flow Table Overflow Attacks and Coun-

termeasures", *European Conference on Networks and Communications (EuCNC)*, pp. 205–209, Athens, Greece, 2016.

89. Yuwen, H., L. Zhang, Z. Wang, and Y. Kong, "Probability-based Delay Scheme for Resisting SDN Scanning", *2nd IEEE International Conference on Computer and Communications (ICCC)*, pp. 1096–1101, Chengdu, China, 2016.

90. Zhao, G., L. Huang, Z. Yu, H. Xu, and P. Wang, "On the Effect of Flow Table Size and Controller Capacity on SDN Network Throughput", *IEEE International Conference on Communications (ICC)*, pp. 1–6, Paris, France, 2017.

91. Panda, A., S. S. Samal, A. K. Turuk, A. Panda, and V. C. Venkatesh, "Dynamic Hard Timeout based Flow Table Management in OpenFlow Enabled SDN", *International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pp. 1–6, Vellore, India, 2019.

92. Cao, J., Z. Yang, K. Sun, Q. Li, M. Xu, and P. Han, "Fingerprinting SDN Applications via Encrypted Control Traffic", *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 501–515, Limassol, Cyprus, 2019.

93. Isyaku, B., M. B. Kamat, K. bin Abu Bakar, M. S. M. Zahid, and F. A. Ghaleb, "IHTA: Dynamic Idle-Hard Timeout Allocation Algorithm based OpenFlow Switch", *IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, pp. 170–175, Penang, Malaysia, 2020.

94. Yang, H., G. F. Riley, and D. M. Blough, "Stereos: Smart Table Entry Eviction for OpenFlow Switches", *IEEE Journal on Selected Areas in Communications*, Vol. 38, No. 2, pp. 377–388, 2019.

95. Mendiboure, L., M. A. Chalouf, and F. Krief, "Load-aware and Mobility-aware Flow Rules Management in Software Defined Vehicular Access Networks", *IEEE Access*, Vol. 8, pp. 167411–167424, 2020.

96. Wu, D., L. Qiao, and Q. Chen, "Research and Implementation of LRU-based Flow Table Management for Onboard Switch", *Prognostics and Health Management Conference (PHM-Besançon)*, pp. 300–303, Online, 2020.

97. Abbasi, M., S. Maleki, G. Jeon, M. R. Khosravi, and H. Abdoli, "An Intelligent Method for Reducing the Overhead of Analysing Big Data Flows in OpenFlow Switch", *IET communications*, Vol. 16, No. 5, pp. 548–559, 2022.

98. Fang, X., W. Zhang, J. Lin, and Y. Liu, "Research on SDN Fingerprint Attack Defense Mechanism Based on Dynamic Disturbance and Information Entropy Detection", *Security and Communication Networks*, Vol. 2022, 2022.