

# Network fingerprinting via timing attacks and defense in software defined networks

Beytullah Yiğit<sup>a,\*</sup>, Gürkan Gür<sup>b</sup>, Fatih Alagoz<sup>a</sup>, Bernhard Tellenbach<sup>c</sup>

<sup>a</sup> Department of Computer Engineering, Bogazici University, 34342 Istanbul, Turkey

<sup>b</sup> Zurich University of Applied Sciences ZHAW, 8401 Winterthur, Switzerland

<sup>c</sup> Cyber-Defence Campus, Thun, Switzerland

## ARTICLE INFO

### Keywords:

SDN  
OpenFlow  
SDN security  
Fingerprinting  
Reconnaissance

## ABSTRACT

Software-Defined Networking (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 paper, 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.

## 1. Introduction

Future networks are expected to meet more challenging flexibility and scalability requirements with the adoption of ubiquitous and diverse digital services addressing an extensive range of use cases and applications. Software-Defined Networking (SDN) has emerged as a fundamental networking paradigm that can meet these escalating demands of future networking. The basic idea behind SDN is the separation of the network control plane and data plane. The Control plane comprises logically centralized, software-based control and network monitoring by means of one (or more) SDN controller(s). This component performs various tasks, such as maintaining an inventory of the devices in the network or supplying the network devices with rules for processing data packets. The data plane then implements the desired network functions by means of the rules defined by the controller. Logically centralized software-based control, i.e., SDN controller, provides the ability to have global visibility of a network in SDN. However, the superior flexibility and control gained with SDN also come with new security risks, including control plane saturation attacks, compromised SDN controllers, and fraudulent flow rule insertion by malicious applications. Hence, the security of SDN is a vivid and diverse research area [1–5].

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 choke the switch with a minimal number of packets effectively. 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 the First Packet Processing Time (FPPT), 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 [6–16].

With this information, attackers can then move from the reconnaissance phase to the subsequent 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 essential to note that SDN fingerprinting

\* Corresponding author.

E-mail addresses: [beytullah.yigit@boun.edu.tr](mailto:beytullah.yigit@boun.edu.tr) (B. Yiğit), [gueu@zhaw.ch](mailto:gueu@zhaw.ch) (G. Gür), [fatih.alagoz@boun.edu.tr](mailto:fatih.alagoz@boun.edu.tr) (F. Alagoz), [bernhard.tellenbach@ar.admin.ch](mailto:bernhard.tellenbach@ar.admin.ch) (B. Tellenbach).

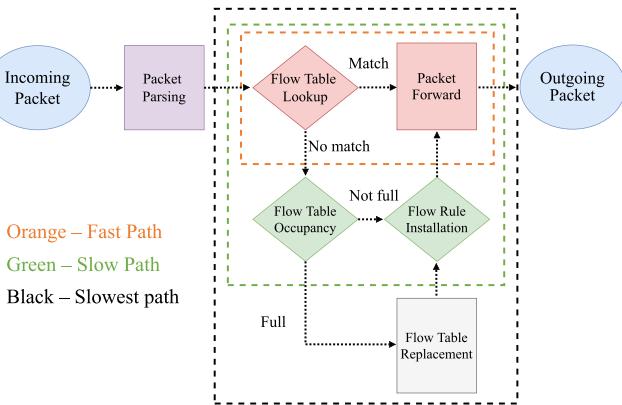


Fig. 1. SDN packet processing flowchart.

research is still at an early stage and focuses mainly 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 paper, both attack and defense methods are proposed, and their performance is investigated 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 highly accurate 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 paper is structured as follows: First, we present some SDN fundamentals relevant to this paper in Section 2 and continue with a discussion of related work in Section 3. This is followed by a presentation of RAFA in Section 4 and our defense method in Section 5. We then use Section 6 to describe how we conducted our extensive simulations and to document the results focusing on the performance of RAFA and our defense method. The limitations and relevant open questions are discussed in Section 7 to provide a critical view of what we achieved. Finally, Section 8 discusses future research directions and concludes the paper.

## 2. SDN fundamentals

Fig. 1 shows the packet processing logic of SDN. There are three types of path for a packet which are shown by dashed rectangular regions in that figure. When a packet arrives at data plane, its header information is compared to the match field of the existing flow rule entries. If there is a match, the action field of the matched entry is enforced (*Fast Path*). If not, the data plane issues a packet-in message to the controller. The controller processes the packet and installs a flow rule to handle this new flow (*Slow Path*). When the switch installs the new flow entry, it must check whether there is enough space in the flow table. If the flow table is full, the controller must perform flow table replacement operations by deleting some flow rules according to table replacement algorithm, resulting in further network performance overhead (*Slowest Path*). Flow table replacement operations can be done via *flow-mod* messages.

Another important point of the processing logic is to understand how and when flows get deleted when the flow table is not full. SDN has three interfaces: southbound, east–westbound and northbound.

Southbound interface lies between the controller and switches. East–westbound interface is for communication of the controllers. Finally, the controllers and network applications communicate via northbound interface. Since this is governed by the southbound protocol and OpenFlow has emerged as the *de facto* southbound protocol, most software-defined networks have the following two mechanisms to delete flow rules: deletion by the controller or deletion via timeouts. OpenFlow defines two timeout values: hard and idle [18]. In idle timeout, flow rules are deleted after a specific inactivity period has passed. In hard timeout, flow rules are removed after a given time, regardless of whether it is active or not. Timeout values are valuable information for attackers since it can directly affect their fingerprinting and attack performance.

The packet processing and flow rule logic described above has at least two consequences. First, an attacker can derive information about the state of a switch from comparing the RTT values of packets because each path of a packet has different interactions and iterations. From the path information of a packet, which depends also on flow rule timeouts, the attacker can gain information about the state and size of the flow table, the timeouts used, and the flow rules themselves (i.e., the match fields). Second, an attacker could use this information to perform more efficient and tailored attacks.

## 3. Related work

Fingerprinting refers to the practice of identifying and gathering information about a target computer system or network to determine its unique characteristics and vulnerabilities. This includes collecting data such as open ports, operating system, software applications, network topology, and configuration settings. The information obtained through fingerprinting can then be used to launch targeted attacks or to enhance security measures to protect against potential threats. There are lots of relevant works in different areas such as radio frequency, indoor localization, browser and TLS parameter fingerprinting [19–22].

In this section, we present the literature on SDN fingerprinting and on methods to make fingerprinting more difficult. Table 1 shows some related technical works summarizing their main methods and inferred SDN information via fingerprinting. [6] 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. [7] is the first practical paper which uses fingerprinting via analysis of RTT differences between packets. In [8], 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 action 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 [9], 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, the fingerprinting accuracy is greatly affected. Furthermore, 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 [10], by sending an initial packet, a new flow is installed and the corresponding RTT ( $T_1$ ) is calculated. Afterwards, a second packet

**Table 1**  
Related literature.

Work	Mechanism and fingerprint info
[6]	- First theoretical work on SDN fingerprinting. - Flow timeout randomization for defense.
[7]	- SDN identification via time inference using t-test.
[8]	- Check the fingerprint success rate w.r.t. hop count and bandwidth. - Infer active flows. - Delay a few packets of inactive flows according to a fixed distribution as defense.
[9]	- 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.
[10]	- Infer flow table capacity and flow table usage. - Determine flow replacement algorithm (LRU or FIFO).
[11]	- Fingerprint OpenFlow controllers via default parameters and packet header information.
[12]	- Infer SDN policy parameters such as hard and idle timeouts, match fields, controller reaction at table full event and information about the topology.
[13]	- Add delay to the packets of every flow with decreasing probability.
[14,17]	- Send timing probes and test streams together to infer match fields of flow rules and active flows. - The proposed defense mechanism is timeout proxy in the switch.
[15]	- Infer match field of flow rules and controller type.
[16]	- Infer match fields and from that information infer controller type. - For defense, packet-out usage is considered.

is sent and its RTT ( $T_2$ ) is calculated. Then, 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.

[11] aims to determine the controller type (ONOS [23], 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 [12], 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 and they can infer some information (in-rack, in-pod and inter pod groups) about the topology. In [13], 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 probability 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 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 significant 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 [12]. 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 a 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 [9–13]. [10,13] do not provide an algorithm for timeout measurement and mainly present their procedure of the measurement. [9–13] 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 [9–11,13]. 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 [14,17], two types of flows (timing probes and test streams) are sent together. Timing probes are specially crafted to be processed in the controller like Address Resolution Protocol (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. 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.

**Table 2**  
Parameters for timeout algorithms.

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

In [15], the 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 [16], similar to [15], 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 related 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 work reliably only 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 widespread adoption of anti-fingerprinting techniques.

#### 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 essentially 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.

For RAFA, we have three clear-cut assumptions. First, the attacker knows at least one flow match field. The match field can also be extracted by SDN fingerprinting, but it is out of scope for this paper. Secondly, the attacker uses a host which can send packets to a server in the target network. Lastly, there is a single SDN network on the path to the target network. Basically, the attacker sends some packets to measure RTTs and infer knowledge according to the RTT deviation. To better cope with network jitter, the attacker sends two packets almost consecutively (back-to-back mode). If the interval between packets becomes longer, network jitter may considerably affect the results.

##### 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 [24] to determine whether these two groups have the same mean. If they are, the network is traditional. 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  $\alpha$ -value of 0.01 indicates that there is only an 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  $T_1$ .

##### 4.2. Timeout fingerprinting

---

##### Algorithm 1: Hard Timeout Calculation

---

```

Data: iterationCount ( $\tau$ )
Result:  $T_{\text{hard}}$ 
1 Get response times and calculate  $RTT_{\text{mean}}$  and  $RTT_{\text{std}}$ 
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  $T_{\text{step}}$ 
10  Send packet and measure RTT
11  if  $RTT < T_{\text{threshold}}$  then
12    | go to 9
13  else
14    |  $T_1 \leftarrow currentTime$ 
15    |  $T_{\text{hard}} \leftarrow T_1 - T_0 - RTT$ 
16    |  $T_{\text{hardArray}}.append(T_{\text{hard}})$ 
17 Find the most common  $T_{\text{hard}}$ ,  $T_{\text{hardMostCommon}}$ , in  $T_{\text{hardArray}}$ 
18 Find the number of occurrence of  $T_{\text{hardMostCommon}}$ ,  $T_{\text{hardOccurrence}}$ 
19 if  $T_{\text{hardOccurrence}} > \rho * \tau$  then
20  | return  $T_{\text{hardMostCommon}}$ 
21 else
22  | return 0

```

---

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 65 535 s [18]. The value 0 means that timeout is not set and 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 2. To eliminate inaccuracies caused by delay variance due to the network conditions, we measure the timeout values iterationCount ( $\tau$ )

times. Majority Coefficient ( $\rho$ ) 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  $\tau$  and  $\rho$  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 ( $T_{\text{step}}$ ) for hard and idle timeout fingerprinting are 0.5 and 0.3, respectively.  $T_{\text{step}}$  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.

$T_{\text{sleep}}$  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,  $T_{\text{step}}$  is used to increase  $T_{\text{sleep}}$  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 Algorithm 1. To understand a significant RTT surge, we need a threshold. Hence, initially, we calculate the sample mean ( $RTT_{\text{mean}}$ ) and sample standard deviation ( $RTT_{\text{std}}$ ) 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  $\tau$  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  $T_{\text{step}}$  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  $\rho * \tau$ , 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 Algorithm 2. 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  $T_{\text{sleep}}$  and the step  $T_{\text{step}}$  are set to 1 and 0.3, respectively (Line 7). After that, the first packet is sent (Line 8).  $T_{\text{sleep}}$  is incremented with  $T_{\text{step}}$  to achieve gradually increasing sleep time (Line

## Algorithm 2: Idle Timeout Calculation

---

```

Data: iterationCount ( $\tau$ ),  $T_{\text{hard}}$ ,  $T_{\text{threshold}}$ 
Result:  $T_{\text{idle}}$ 

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

```

---

9). It is added to the total sleep time (Line 10) and then we wait for  $T_{\text{sleep}}$  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  $T_{\text{sleep}}$  to test longer timeouts.

If the RTT is bigger, we check whether  $T_{\text{sleep}}$  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  $T_{\text{step}}$  from  $T_{\text{sleep}}$  (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  $1+2+0.5=6$ , it is bigger than 5. Therefore, the hard timeout deletes the flow when  $T_{\text{sleep}}$  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  $T_{\text{step}}$  from  $T_{\text{sleep}}$ .

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  $T_{\text{sleep}}$  only covers the sleep time. However, it took approximately  $RTT/2$  until the arrival of the sent packet at 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

**Table 3**  
Parameters for defense mechanism.

Parameters	Explanation
PacketInCount	Packet-in count.
FRM <sub>Count</sub>	Flow removal messages count.
PC <sub>threshold</sub>	Packet count threshold for the flow removal messages.
FD <sub>threshold</sub>	Flow duration threshold for the flow removal messages (in seconds).
Ratio <sub>threshold</sub>	Ratio threshold between the packet-in and the flow removed messages.
Randomization coefficient ( $\beta$ )	Randomization coefficient for the flow rule timeouts.

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.

## 5. Lightweight defense against SDN 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 [6,9]. [6] provides theoretical recommendations for the randomization of flow timeouts but does not offer any implementation or practical insights. In [9], 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.

To prevent timeout fingerprinting, a lightweight randomization mechanism is suggested to make the network more complex and uncertain in this work. In 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 domain [25,26]. 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 in this work is a lightweight defense mechanism because 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. In our mechanism, initially, we randomize the timeouts of all flows with a randomization coefficient,  $\beta$ . Then a time window is started. In each time window, the number of packet-in, PacketIn<sub>Count</sub>, and flow removed messages, FRM<sub>Count</sub>, for each switch port are 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.

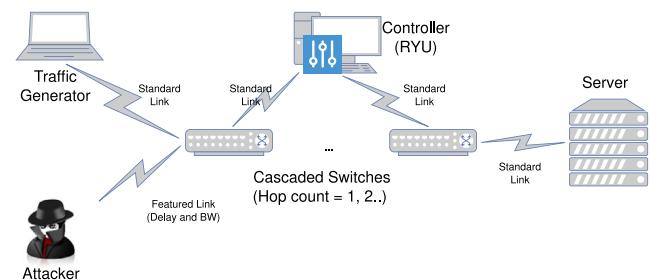


Fig. 2. Simulation topology.

Our mechanism counts flow removed messages according to the following rule:

$$\text{duration} < \text{FD}_{\text{threshold}} \wedge \text{packetCount} < \text{PC}_{\text{threshold}} \quad (1)$$

Eq. (1) shows the counting of flow removed messages according to the packet count, PC<sub>threshold</sub>, and duration, FD<sub>threshold</sub>, 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, for a server which processes numerous flows, 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 PacketIn<sub>Count</sub> and FRM<sub>Count</sub> is checked. Our mechanism increases randomization coefficient according to the following criterion:

$$(\text{PacketIn}_{\text{Count}}/\text{FRM}_{\text{Count}}) < \text{Ratio}_{\text{threshold}} \quad (2)$$

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  $\beta$ . For instance, if  $\beta$  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.

## 6. Simulations and performance evaluation

The topology used in the experiments is shown in Fig. 2. Our topology includes two hosts (a traffic generator and a server), a controller, and one or more switches according to the simulation configuration. To emulate hosts and switches, Mininet [27] is chosen since it can allow configuring link parameters such as bandwidth and delay flexibly. RYU [28] is used as the SDN controller. Our proposed algorithms are implemented in Python. RAFA uses Impacket [29] 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 16 GB RAM.

### 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,  $\alpha$  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. For this purpose, 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

**Table 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
Background traffic (Mbps)	10, 100, 1000

**Table 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

in our selection: RTTs of the servers varies between 5 ms to 350 ms. In addition to delay, to consider changing network conditions like background traffic, the simulation is executed in every hour for 24 h. Thus, 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. In this way, the fingerprinting success rate with respect to hop count, link delay, bandwidth (BW) and background traffic (BT) is investigated. Network parameters for the SDN fingerprinting simulation are listed in Table 4. 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 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 reasonable since as the hop count increases, the 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 while 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 a packet becomes smaller. However, this change is not significant. RAFA has better accuracy with more challenging network conditions compared to the related work [7–9]. For scalability analysis, we conduct 54 different simulations by examining the effects of background traffic, link delay, bandwidth and hop count in our experiments. [8] only considers hop count and bandwidth while [9] looks at background traffic. In [7], 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 [8], 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.

**Table 6**

Network parameters for timeout simulations.

Timeout simulation network parameters	
Hop count	1, 2, 3
Link delay (ms)	0, 100, 200
Bandwidth (Mbps)	1000
Background traffic (Mbps)	1000

The solution in [9] 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.

## 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 s, respectively. Parameters `iterationCount` ( $\tau$ ) and `majority coefficient` ( $\rho$ ) are 5 and 0.5, respectively. Accordingly, a timeout value should be observed at least 3 times. The network parameters for the simulation are shown in Table 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 a noticeable effect on the results; the maximum values are used for them. Thus, there are nine different network conditions and each condition is simulated 50 times to evaluate the accuracy performance.

The success ratio of hard timeout simulations are 1.0. The accuracy of the RAFA for idle timeout fingerprinting is shown in Table 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 [9], 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 [10], the exact values of timeouts cannot be found. The simulations show that timeouts are found in plus-or-minus 10% error margin. The success ratio of the timeout simulations is 96% in [11]. In [12], similar to [10], the timeout values are measured in a plus-or-minus 3% error margin. There is no timeout simulation result in [13]. 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 take into account any network parameters, except [9] which only considers background traffic. Again, RAFA has better accuracy for more challenging network conditions compared to the related work in [9–13].

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. For experimental comparison, we implemented the proposed algorithms in [9,11] from this set of available alternative techniques in the literature. As we pointed out earlier, there is only procedural explanation in [10,13] which has missing parts for a specific implementation such as RTT surge detection method. Besides, [10,13] employ a very similar approach with [9,11]. The simulation results of [9,11] are shown in Tables 8, 9, 10, 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 the idle timeout first, it leads to very low accuracy, especially when the link delay is high.

**Table 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

**Table 8**  
Hard timeout simulation results for [9].

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

**Table 9**  
Idle timeout simulation results for [9].

		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

**Table 10**  
Hard timeout simulation results for [11].

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

**Table 11**  
Idle timeout simulation results for [11].

		Hop count		
		1	2	3
Delay	0	0	0	0
	100 ms	0	0	0
	200 ms	0	0	0.02

### 6.3. Fingerprinting countermeasure simulations

We test our lightweight defense solution which randomizes flow rule timeouts according to the network status. The 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 s. If the randomization coefficient of a port equals to 0.2, hard timeout is randomly set between 8 and 12 for that port's flows. 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 s, respectively. To better focus on probing flows, long and large flows are ignored while counting flow removed messages. Duration threshold,  $FD_{threshold}$ , equals to  $2 * (hardtimeout + idletimeout)$  for this simulation and the duration bigger than 30 s is assumed as long. The threshold for large flows,  $PC_{threshold}$ , is based on measured packet count and the value 500 is considered large. The time window is 180 s while  $Ratio_{threshold}$  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 ms. The CPU and memory usage figures increase by 5.78% and 0.77%, respectively. The results show that the proposed method slightly impacts performance and effectively prevents fingerprinting attacks.

## 7. Discussion

SDN becomes an integral part of networking and telecommunications technology and it is used in many different parts of ICT infrastructure. However, its security deserves more attention since research about SDN security is generally confined to control plane saturation attacks. SDN fingerprinting attacks can give the attackers vast information about a network's structure and operation. Nevertheless, as data centers and core networks are considered SDN's most deployed locations and are assumed to be controlled and restricted environments, SDN fingerprinting can be regarded as a secondary problem. Besides, security countermeasures's performance concerns seem as natural excuses to overlook this threat.

SDN fingerprinting, which involves identifying the network type and timeout values, can be carried out with and without 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, the detection or prevention of SDN fingerprinting is a challenging 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  $T_2$ ) each day. Timeout fingerprinting has similar logic, but it also highly depends on actual hard ( $T_{hardActual}$ ) and idle ( $T_{idleActual}$ ) timeout values. Our algorithms can infer the hard timeout value in  $T_{hardActual} + 1$  seconds. The 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 s, 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, reducing 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 a network environment can be pretty challenging since it is affected from a variety of dynamic network parameters and states.

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

can increase overhead for the attacker but also provides better results in terms of accuracy.

As an underlying assumption, we accept that the probing packets go through only one SDN network on the path to the target network. It could be an interesting research direction to examine the transit of packets through multi-domain SDN in a fingerprinting scenario. Like wireless networks, multi-point calculation from multiple measurement locations can be employed to figure out multi-SDN domains and adjust probing packets accordingly to fingerprint the network efficiently.

There are several SDN controllers, such as ONOS [23], OpenDaylight [30] and Floodlight [31]. 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 the processing times 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 controller types is expected to yield similar results except for some borderline conditions such as really restricted resource assignment to a controller.

The simulation topology employed in our research is relatively simple, as including additional hosts or switches has minimal effect on the fingerprinting process. In fact, more hosts and switches can cause more load on the controller, which potentially increases the 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 fingerprinting since the whole idea relies on the RTT difference of different packets. Therefore, the number of hop counts 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.

## 8. Conclusions

In this work, we elaborate on the concept of SDN fingerprinting and the type of information that can be gathered from it. While a significant amount of work 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 diverse 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 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 them.

As a future research direction, machine learning algorithms could be leveraged to extract more diverse information from an SDN network through fingerprinting techniques. Using this intelligence could be an interesting research idea to design and perform efficient DDoS attacks against the network. Besides, additional network parameters can be randomized by considering a more comprehensive set of network states.

## CRediT authorship contribution statement

**Beytullah Yiğit:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Software, Investigation. **Gürkan Gür:** Conceptualization, Writing – review & editing, Supervision. **Fatih Alagoz:** Conceptualization, Writing – review & editing, Supervision. **Bernhard Tellenbach:** Conceptualization, Writing – review & editing, Supervision.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## References

- [1] R. Deb, S. Roy, A comprehensive survey of vulnerability and information security in SDN, *Comput. Netw.* 206 (2022) 108802, <http://dx.doi.org/10.1016/j.comnet.2022.108802>.
- [2] B. Yiğit, G. Gur, B. Tellenbach, F. Alagoz, Secured communication channels in software-defined networks, *IEEE Commun. Mag.* 57 (10) (2019) 63–69, <http://dx.doi.org/10.1109/MCOM.001.1900060>.
- [3] J.C. Correa Chica, J.C. Imbachi, J.F. Botero Vega, Security in SDN: A comprehensive survey, *J. Netw. Comput. Appl.* 159 (2020) 102595, <http://dx.doi.org/10.1016/j.jnca.2020.102595>.
- [4] Y. Maleh, Y. Qasmaoui, K. El Gholami, Y. Sadqi, S. Mounir, A comprehensive survey on SDN security: threats, mitigations, and future directions, *J. Reliab. Intell. Environ.* (2022) 1–39.
- [5] M. Priyadarsini, P. Bera, Software defined networking architecture, traffic management, security, and placement: A survey, *Comput. Netw.* 192 (2021) 108047.
- [6] R. Klöti, V. Kotronis, P. Smith, OpenFlow: A security analysis, in: 2013 21st IEEE International Conference on Network Protocols, ICNP, 2013, pp. 1–6, <http://dx.doi.org/10.1109/ICNP.2013.6733671>.
- [7] S. Shin, G. Gu, Attacking software-defined networks: A first feasibility study, in: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 165–166, <http://dx.doi.org/10.1145/2491185.2491220>.
- [8] H. Cui, G.O. Karame, F. Klaedtke, R. Bifulco, On the fingerprinting of software-defined networks, *IEEE Trans. Inf. Forensics Secur.* 11 (10) (2016) 2160–2173, <http://dx.doi.org/10.1109/TIFS.2016.2573756>.
- [9] S. Khorsandrou, A.S. Tosun, Time inference attacks on software defined networks: Challenges and countermeasures, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, 2018, pp. 342–349, <http://dx.doi.org/10.1109/CLOUD.2018.00050>.
- [10] J. Leng, Y. Zhou, J. Zhang, C. Hu, An inference attack model for flow table capacity and usage: Exploiting the vulnerability of flow table overflow in software-defined network, 2015, CoRR <abs/1504.03095>. arXiv:1504.03095. URL <http://arxiv.org/abs/1504.03095>.
- [11] A. Azzouni, O. Braham, T.M.T. Nguyen, G. Pujolle, R. Boutaba, Fingerprinting OpenFlow controllers: The first step to attack an SDN control plane, in: 2016 IEEE Global Communications Conference, GLOBECOM, 2016, pp. 1–6, <http://dx.doi.org/10.1109/GLOCOM.2016.7841843>.
- [12] B. Ahmed, N. Ahmed, A.W. Malik, M. Jafri, T. Hafeez, Fingerprinting SDN policy parameters: An empirical study, *IEEE Access* 8 (2020) 142379–142392, <http://dx.doi.org/10.1109/ACCESS.2020.3012176>.
- [13] T. Wang, H. Chen, J. Cao, A lightweight SDN fingerprint attack defense mechanism based on probabilistic scrambling and controller dynamic scheduling strategies, *Secur. Commun. Netw.* 2021 (2021) <http://dx.doi.org/10.1155/2021/6688489>.
- [14] J. Sonchack, A.J. Aviv, E. Keller, Timing SDN control planes to infer network configurations, in: Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, in: SDN-NFV Security '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 19–22, <http://dx.doi.org/10.1145/2876019.2876030>.
- [15] M. Zhang, J. Hou, Z. Zhang, W. Shi, B. Qin, B. Liang, Fine-grained fingerprinting threats to software-defined networks, in: 2017 IEEE Trustcom/BigDataSE/ICESS, 2017, pp. 128–135, <http://dx.doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.229>.
- [16] J. Hou, M. Zhang, Z. Zhang, W. Shi, B. Qin, B. Liang, On the fine-grained fingerprinting threat to software-defined networks, *Future Gener. Comput. Syst.* 107 (C) (2020) 485–497, <http://dx.doi.org/10.1016/j.future.2020.01.046>.
- [17] J. Sonchack, A. Dubey, A.J. Aviv, J.M. Smith, E. Keller, Timing-based reconnaissance and defense in software-defined networks, in: Proceedings of the 32Nd Annual Conference on Computer Security Applications, ACSAC '16, ACM, New York, NY, USA, 2016, pp. 89–100, <http://dx.doi.org/10.1145/2991079.2991081>.
- [18] OpenFlow switch specification 1.5.1, 2022, <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>, [Accessed in 11.11.2022].

- [19] F. Xie, H. Wen, J. Wu, W. Hou, H. Song, T. Zhang, R. Liao, Y. Jiang, Data augmentation for radio frequency fingerprinting via pseudo-random integration, *IEEE Trans. Emerg. Top. Comput. Intell.* 4 (3) (2020) 276–286, <http://dx.doi.org/10.1109/TETCI.2019.2907740>.
- [20] G. Ding, Z. Tan, J. Wu, J. Zhang, Efficient indoor fingerprinting localization technique using regional propagation model, *IEICE Trans. Commun.* 97 (8) (2014) 1728–1741.
- [21] L. Babun, H. Aksu, A.S. Uluagac, CPS device-class identification via behavioral fingerprinting: From theory to practice, *IEEE Trans. Inf. Forensics Secur.* 16 (2021) 2413–2428, <http://dx.doi.org/10.1109/TIFS.2021.3054968>.
- [22] M. Laštovička, S. Špaček, P. Velan, P. Čeleda, Using TLS fingerprints for OS identification in encrypted traffic, in: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2020, pp. 1–6.
- [23] ONOS : Open Network Operating System, 2022, <https://github.com/opennetworkinglab/onos>, [Accessed in 11.11.2022].
- [24] W. Haynes, Student's t-test, in: *Encyclopedia of Systems Biology*, Springer New York, New York, NY, 2013, pp. 2023–2025, [http://dx.doi.org/10.1007/978-1-4419-9863-7\\_1184](http://dx.doi.org/10.1007/978-1-4419-9863-7_1184).
- [25] J.-H. Cho, D.P. Sharma, H. Alavizadeh, S. Yoon, N. Ben-Asher, T.J. Moore, D.S. Kim, H. Lim, F.F. Nelson, Toward proactive, adaptive defense: A survey on moving target defense, *IEEE Commun. Surv. Tutor.* 22 (1) (2020) 709–745, <http://dx.doi.org/10.1109/COMST.2019.2963791>.
- [26] W. Soussi, M. Christopoulou, G. Xilouris, G. Gür, Moving target defense as a proactive defense element for beyond 5G, *IEEE Commun. Stand. Mag.* 5 (3) (2021) 72–79, <http://dx.doi.org/10.1109/MCOMSTD.211.2000087>.
- [27] Mininet, 2022, <http://mininet.org/>, [Accessed in 11.11.2022].
- [28] RYU SDN framework, 2022, <https://ryu-sdn.org/>, [Accessed in 11.11.2022].
- [29] Impacket, 2022, <https://github.com/SecureAuthCorp/impacket>, [Accessed in 11.11.2022].
- [30] OpenDaylight, 2022, <https://www.opendaylight.org/>, [Accessed in 11.11.2022].
- [31] FloodLight controller, 2022, <https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview>, [Accessed in 11.11.2022].



**Beytullah Yiğit** is a Ph.D. candidate in the Department of Computer Engineering, Bogazici University. His current research and thesis are related with network and information security. He received his B.Sc. and M.S. degree in computer engineering in 2012 and 2014, respectively, from Bogazici University.



**Gürkan Gür** is a senior lecturer at Zurich University of Applied Sciences (ZHAW) – Institute of Applied Information Technology (InIT) in Winterthur, Switzerland. He received his B.S. degree in electrical engineering in 2001 and Ph.D. degree in computer engineering in 2013 from Bogazici University in Istanbul, Turkey. He is a senior member of IEEE and a member of ACM. His research interests include Future Internet, information security, next-generation wireless networks and information-centric networking.



**Fatih Alagöz** is a professor in the Computer Engineering Department of Bogazici University, Turkey. He obtained his B.Sc. degree in Electrical Engineering in 1992 from Middle East Technical University, Turkey, and DSc degree in Electrical Engineering in 2000 from George Washington University, USA. His current research areas include wireless/mobile/satellite networks, network security and beyond 5G communications.



**Bernhard Tellenbach** is head of cyber security at the Cyber-Defence Campus at armasuisse Science and Technology and former Professor and head of the information security research group at Zurich University of Applied Sciences. He received his Ph.D. degree in 2013 from ETH Zurich. His research interests include information security, network security, system security, and security training and testing.