Received June 30, 2020, accepted July 16, 2020, date of publication July 27, 2020, date of current version August 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3012176

Fingerprinting SDN Policy Parameters: An Empirical Study

BILAL AHMED[1,2], NADEEM AHMED[1,3], ASAD WAQAR MALIK[1], (Senior Member, IEEE), MOHSIN JAFRI[1,2], AND TAIMUR HAFEEZ[1], (Graduate Student Member, IEEE)

[1]School of Electrical Engineering and Computer Science (SEECS), National University of Sciences and Technology (NUST), Islamabad 44000, Pakistan
[2]National University of Sciences and Technology (NUST) Baluchistan Campus, Quetta 85300, Pakistan
[3]Cyber Security Cooperative Research Centre (CSCRC), Perth, WA 6027, Australia

Corresponding author: Bilal Ahmed (bilal.ahmed2@seecs.edu.pk)

ABSTRACT Research in Software Defined Networks (SDN) has gained momentum in recent years. SDNs are getting mature, however, there are still many research challenges that need to be considered before SDN become ubiquitous. The adaptation of the technology brings immediate focus to its security aspects. The centralized nature of the SDN makes it prone to many denial of service attacks, especially if the policy parameters of SDN are known to adversaries. In this work, we present techniques to perform fingerprinting of SDN including policy parameters such as hard and idle/soft timeouts, OpenFlow match-fields used by the SDN controller, controller reaction at table full event and information about the topology of the targeted network. An adversary can launch a carefully planned attack, especially on the SDN data plane, if these policy parameters are easily discoverable for a SDN domain. Assuming access to the SDN domain's host and customized packet generation from the compromised host, we propose efficient techniques to discover these aforementioned policy parameters. The results of the proposed fingerprinting techniques are verified by using Mininet.

INDEX TERMS Data plane, network security, OpenFlow, SDN.

I. INTRODUCTION

Software Defined Networks (SDNs) are getting significant research attention in the networking domain due to the programmability of the networks. In a typical SDN environment, the control plane is decoupled from the forwarding plane. This feature enables the network administrators to apply policy parameters to the control plane and to program the network features at the forwarding plane. OpenFlow [1], [2] has become a de-facto standard for communication between the control and forwarding planes, with newer versions supporting more options to manage the SDN intelligently [3]. SDNs are getting mature, however, there are many research challenges to be addressed before SDN becomes ubiquitous. One of the key challenges that seek immediate attention is SDN security vulnerabilities. Introduction of SDN has in fact increased the attack vectors, authors in [4] found that SDN has 114 medium and high severity level vulnerabilities. There exist multiple attacks targeting both the data plane and control plane of the SDN [5]. Shaghaghi et.al, in [6] discuss attacks on the data plane, whereas, attacks on the control plane are covered in survey papers [7], [8].[1]

In this paper, we focus on a specific data plane attack referred to as Flow Table Entry Attack (FTEA). FTEA exploits the limited capacity of forwarding tables in the SDN enabled switches. Once the Flow Entry Table (FET) gets full, the newly arriving flows are either dropped or a prior flow is removed from the FET to make space for the new request. However, removal of an existing entry from FET induces latency and extra workload on the controller. This triggers a chain of message passing between the controller and the switch which in turn controls the flow capacity. Therefore, FTEA consumes the SDN controller's resources by constantly engaging it to install attacker initiated bogus entries in the FET [11]. The network faces denial of service (DoS) attack which prevents it from handling the legitimate flow installation requests. Clearly, FET size is a key parameter index in such attacks and have a huge impact on the performance of the network [11].

The associate editor coordinating the review of this manuscript and approving it for publication was Guangjie Han.

[1]We additionally refer the readers to [9] and [10] that explore fingerprinting SDN controllers and applications respectively.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

There are three main configurable SDN policy parameters, namely flow timeout value, match field and flow replacement policy that affect how quickly the capacity of the FET can be exhausted. An effective attack can be executed by an attacker if the current values of these parameters are known and exploited accordingly.

- *Flow timeout values* determine for how long the flow entries stay in the FET. The attacker tries to overwhelm the table by inserting a large number of bogus flows. The timeout values, on the other hand, triggers the removal of the installed flows.
- *Match fields* are used for by a switch to ascertain if a new flow can be matched to an existing flow in FET. Otherwise, it has to be referred to the controller for decision making. The number of entries in FET depends on the employed match fields. For example, if the policy to install flows is based on the source and destination IP addresses only, then all TCP flows between two nodes only require two entries in the FET for forward and backward directions. However, if the policy permits the use of source and destination ports as the match fields, it may result in $2 * M * N$ entries (M and N denotes the sender and receiver port numbers with theoretical maximum values of 65535, due to 16 bit port number fields in the TCP header).
- *Flow replacement policy* comes into play when the controller installs a new flow on a switch with a full FET. The switch replies with a table full message to the controller that can simply drop the new flow or otherwise installs it by removing an earlier flow table entry. The attack using the knowledge of the current flow replacement policy is of particular importance [12] as it results in a drastic degradation of the network's performance.

Beside these configurable parameters, SDN topology discovery also reveals important information about round trip times between end hosts which enables the attacker to choose its preferred hosts for maximum attack efficiency. For this work, we assume that we have access to one of the hosts within the SDN domain and that we can generate custom packets from the networking stack of the compromised host(Figure. 1). Our methodology does not require any support from other networking elements in the SDN fabric such as switches and the controller. Our main contribution in this paper is the efficient discovery of the aforementioned policy parameters and SDN topology mechanism that can be used to exacerbate the effect of FTEA in SDN. Specifically:

1) We present a detailed analysis of fingerprinting techniques used to discover the configurable policy parameters in an SDN based environment.
2) We adopt an empirical latency based approach to discover the current timeouts set by the controller and narrow down match fields configured in the controller by proposing a series of algorithms.
3) We propose an analysis technique to learn the flow replacement policy and network topology of the target SDN network. It is worth mentioning that we exploit resources available at end host(s) without any changes in the SDN fabric.
4) We discuss the impact of data plane attacks utilizing information of these configurable parameters and SDN topology information. Specifically, we study the impact of FTEA on the data plane and explore the SDN policy parameters that affect the intensity of such attacks. This research work presents concerns for network administrators to properly configure the SDN such that adversary/attacker find it hard to predict these policy parameters.

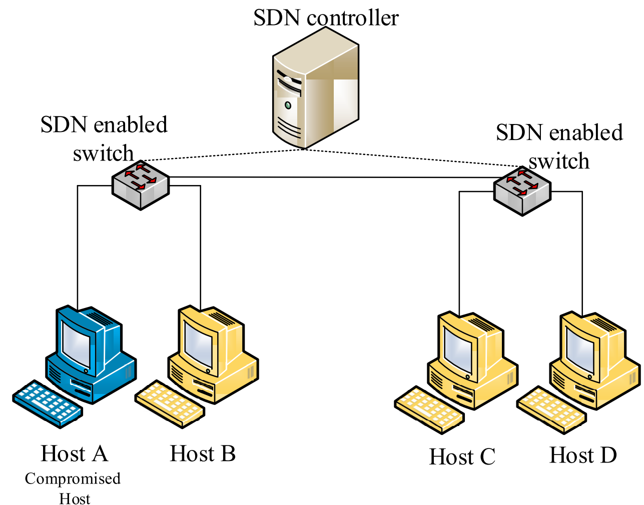


FIGURE 1. Fingerprinting scenario.

This article has been organized as follows. Section II details the literature analysis and the limitations of existing works in the domain of fingerprinting SDN. Section III discusses the background information necessary to understand the methodologies adopted in this article. Section IV explains the methodology of proposed fingerprinting modules. The results and analysis are discussed in the Section V. Section VI summarizes the research work.

II. RELATED WORK

There are several research works that focus on the fingerprinting techniques used for the identification of SDN controllers [13]. Our focus is on fingerprinting policy parameters rather than just identifying the type of the controller. Identification of an SDN controller based on timeout is inherently an error-prone process. As we know that, default hard timeout (T_{hard}, expiry time after which the entry is expunged from FET) of POX controller is 30 sec while of RYU controller is 0 sec so, identification of a particular controller based on T_{hard} becomes dubious if the administrator specifies the POX T_{hard} to 0 sec.

Methods to discover timeout values based on latency measurement for SDN have been discussed by Zeitlin [14]. This research work assumes that entries in FET are not removed

by T_{hard} when finding the value of the soft/idle timeout(T_{idle}, expiry time if entry is not refreshed) that seems unrealistic. Both T_{hard} and T_{idle} values exist for the flow entries that may result in a discovery of wrong timeout values. Similarly, Azzouni *et al.* [9] work on fingerprinting controllers under their default parameters ignoring the combined existence of T_{hard} and T_{idle} values.

Kandoi and Antikainen [15] present the analysis and impact of T_{idle} only in the Distributed DoS (DDoS) attacks. However, they neither include how to find timeouts nor do they cover all the combinations of timeouts. Their analysis is incomplete for the case when FET is full assuming only timeouts expiry method.

Zhou *et al.* [16] estimate the numbers of entries in FET assuming the table is empty to begin with whereas, in a production network there are already thousands of entries installed for on-going communication. It is highly likely that these entries are never removed because of T_{idle}, if T_{hard} is not being set. Zhang *et al.* [17] identify the default match fields of SDN controllers whereas the match fields vary from policy to policy within the same controller. Moreover, they use the UDP packets for experiments and ignore the time synchronization errors in analyzing the results. In contrast, our study considers all timeout combinations. Moreover, our work is the first to identify the impact of the mitigating policy for handling the overflow of FETs on the DDoS attack intensity. We have also fingerprinted the basic firewall omitting the blocked values of match fields which considerably increases the robustness of FTEA attacks.

III. BACKGROUND INFORMATION

SDN packet forwarding mechanism is different from the traditional forwarding in legacy networks. The separation of the control and the data planes enables the controller to play a vital role in admitting flows and making packet forwarding decisions. When a packet is first transmitted in the SDN, a switch looking at the flow for the first time does not know how to forward the packet triggering a ''Table miss'' event. The switch forwards it to the controller by encapsulating it in OpenFlow ''Packet-In'' message. Pre-defined controller's policy then decides the forwarding path and generates an appropriate flow rule message (a Packet-Out or Flow-Mod message), which contains the action to be performed for that flow at the switch. The switch installs this flow rule for future reference.

When the next packet arrives that matches the flow rules already installed at the switch, it is forwarded directly through the data plane without referring it to the SDN controller. It is pertinent to note that the processing of a packet through the data plane alone is much faster than referring it to the controller as it avoids the message passing delay between switch and controller, and the processing delay involved at the controller. The observed round trip time (RTT) for flow installation is thus very high when there is no matching flow rule at a switch and switch has to ask the controller for a decision regarding that flow. Moreover, this difference in RTT increases significantly if there is more than one switch installed between the communicating end hosts, whereby each switch repeats the process of sending the unknown flow to the controller.

For the rest of this study, we denote the RTT when the flow is being installed for the first time along a path as RTT_{FE} and RTT_{avg} as the RTT when the flow is already installed. Figure. 2 highlights the difference between RTT_{FE} and RTT_{avg} along with details of all involved steps. Intuitively, $RTT_{FE} > RTT_{avg}$ and the difference between the two values increases when multiple switches are involved along a particular path. Table 1 lists the nomenclatures used in this article. Typically, each flow rule installed at a switch contains match fields, forwarding actions and automatic expiry mechanism. Each entry is associated with timers to delete the entry from the switch FET. Considering the memory constraint of a switch, the expiry mechanism removes entries that become stale or has been installed for a long time. The timers used for flow management are known as idle/soft timeout and hard timeout, which is explained below.

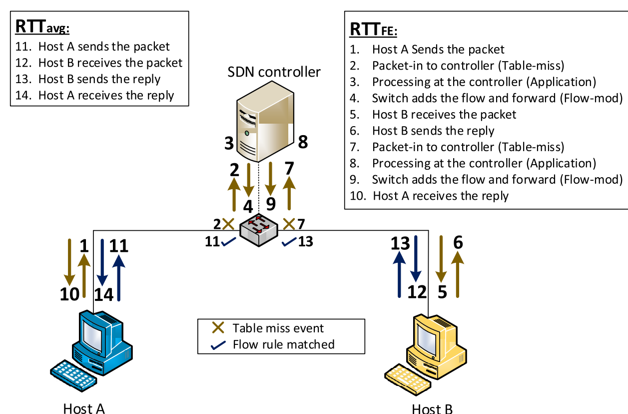


FIGURE 2. Flow installation latency and data plane latency.

TABLE 1. Nomenclature.

Symbols	Description
T_{hard}	Hard timeout
T_{idle}	Idle timeout
RTT_{FE}	RTT when new flow entry is installed
RTT_{avg}	RTT when flow entry is already installed
σ_{RTT}	Standard deviation in RTT
T_{thresh}	Threshold for new flow entry installation detection
T_{sleep}	Sleeping interval between two consectuive packets
T_{step}	Step size to increase sleep time
RTT_{ping}	RTT of last ping
T_{sl-ini}	Initial value of sleep time
n_{rounds}	Number of hard timeout occurance while discovering idle timeout
D_p	Number of dropped packets
n_{FE}	Number of flow entries
r_{p-g}	Rate for packet forging
$RTT_{reinstall}$	RTT when flow entry is replaced at the switch

Idle timeout – The idle timer triggers the removal of the entry when inter-packet time value becomes equal to set idle timeout (T_{idle}) value. In other words, entry is removed if the

TABLE 2. Flow Entry Timeouts.

Pair ID	Idle/soft Timeout	Hard Timeout	Effect
A	0	0	Infinite Idle and Hard timeouts. The entry will not be deleted until controller instructs to do so
B	0	Y	Entry will be only be deleted after Y seconds since its installation time
C	X	0	Entry will be only be deleted if inter packet gap equals X seconds
D	X	Y	Entry deleted when either of timeout is fulfilled irrespective of their priority

entry's soft timeout is not refreshed by the arrival of a new packet belonging to the flow.

Hard timeout – The hard timeout (T_{hard}) signals the removal of entry even if packets of the flow are passing the switch but the flow has been installed for a long time. The SDN administrator configures the timeouts to any values within the range 0 to 65535 sec with the T_{hard} value always greater than the T_{idle} [18]. A value of zero means that timeout is infinity and entry is not deleted until the controller explicitly instructs the switch to do so. There are thus four possible combinations of these timeouts as listed in Table 2.

The network administrator sets the timeout based on the traffic density and the network load. Usually, the combination 'A' in Table 2 is rarely used because the manual deletion of all flow entries by the controller is a resource-consuming task. A network configured to use only T_{idle} can have an entry for an infinite time provided the flow has periodic transmissions within the T_{idle} period. Choosing a good combination of these timeout values is thus imperative for configuring an efficient SDN deployment. On the other hand, information regarding these timeout values is also of immense value for an attacker targeting to overwhelm FETs.

IV. METHODOLOGY

Having covered the background information in the previous section, we are in a position to discuss our latency based fingerprinting methodology to ascertain the timeout values, flow matching rules used by the controller and controller reaction policy when FET gets full.

Description of the attack:

An adversary generates a burst of unique flows to overflow the FET in minimum possible time. For this purpose, the attacker forges packets that force the controller to install separate entry for every packet. The window of opportunity available to the attacker depends on how the timeouts are set in the SDN policy. The easiest target is the case when both these timeouts are not set (pair ID 'A' in Table 2). As there would be no timeouts, the flow would remain installed and the attacker quickly generates enough flows to fill the FET without worrying that the flows would be automatically removed. In case only T_{idle} is set, the host needs to send flow installation requests at a high rate to install as much flow as possible before the switch starts removing flows through expiry of T_{idle}. The host refreshes FET entries by repeating flows within T_{idle} to avoid the flow expiry. If T_{hard} is set with no T_{idle}, the entries expire at T_{hard} only. If both T_{idle} and T_{hard} are employed in the policy, the attacker needs to keep track of both time out values. The window of opportunity is limited to T_{idle} requiring refreshed entries, while flows need to be reinstalled after T_{hard}.

We now detail algorithms to compute the T_{hard} and T_{idle} values that are used by the attacker.

A. COMPUTATION OF FIRST TIMEOUT AND ITS NATURE

We design an algorithm, employing Round Trip Time (RTT) measurements with simple ICMP pings, to find the timeout values being used in a particular SDN deployment. As discussed in Section III, RTT measurements indicate whether a new flow initiation results in FET miss and the subsequent Switch-Controller-Switch communication results in additional latency. Also, the initial pings confirm whether the network is SDN enabled, based on the difference of RTT of first and consequent packets of a flow because it takes a considerable more time for the flow to get installed through the controller ($RTT_{FE} > RTT_{avg}$). We start from finding the first timeout and its nature (Algorithm 1) and ascertain whether it is a hard or idle timeout. Depending on the nature of the first timeout, we discover the other timeout value i.e., if the first timeout is hard, we find whether T_{idle} has been set or not.

We assume that a host A is under our control in the SDN domain. The host A records the current system time as T_0 and sends a ping to host B to install a new flow entry in switches in the path towards B assuming that there exists no prior flow entry for this communication. On receipt of the ping reply, the RTT of this flow installation is measured as RTT_{FE}. Now, the host sends 'n' number of pings to calculate the average RTT (RTT_{avg}) and the standard deviation of RTT (σ_{RTT}). We define a threshold T_{thresh} based on RTT values to distinguish between packets that traverse the data plane and those that are referred to the controller for decision making. We calculate T_{thresh} value by using Equation 1 (line 5, Algorithm 1) and assume that any RTT value below T_{thresh} is simply going through the data plane without incurring the switch-controller-switch delay of initial flow installation.

$$T_{thresh} = RTT_{avg} + 4 * \sigma_{RTT} \tag{1}$$

Next, we initialize T_{sleep} with 0 and T_{step} with 100 ms and start sending pings after every T_{sleep} while incrementing sleep timer with step value if the RTT_{ping} is less than T_{thresh}, for each iteration of ping. Note that the time difference between two consecutive pings increases as the number of pings increases. The sleep time at i^{th} ping is calculated using Equation 2. Whenever RTT_{ping} exceeds T_{thresh}, it shows the expiry of the entry and installation of a new flow. We note the

Algorithm 1 Find First Timeout and Its Nature

1: Note current system time as T_0
2: Send the first ping to install the flow entry
3: Calculate RTT as RTT_{FE}
4: Send n pings to calculate average RTT (RTT_{avg}) and the standard deviation σ_{RTT}
5: Calculate $T_{thresh} = RTT_{avg} + 4 * \sigma$
6: Set $T_{sleep} = 0$ and $T_{step} = 100ms$
7: $T_{sleep} = T_{sleep} + T_{step}$
8: Sleep for T_{sleep}
9: Send a ping and calculate the RTT (RTT_{ping})
10: **if** ($RTT_{ping} < T_{thresh}$) **then**
11: Go to 7
12: **else**
13: Note the current system time as T_1 and last sleep time T_{sleep}
14: **end if**
15: Sleep for T_{sleep}
16: Send a ping and calculate the RTT (RTT_{ping})
17: **if** ($RTT_{ping} > T_{thresh}$) **then**
18: Idle timeout $T_{idle} = T_{sleep}$
19: **else**
20: Hard timeout $T_{hard} = T_1 - T_0$
21: Note current system time as T_2
22: **while** $(T_2 - T_1) < (0.8 * T_{hard})$ **do**
23: Note current system time as T_2
24: Sleep for T_{sleep}
25: Send a ping
26: **end while**
27: **while** ($RTT_{ping} < T_{thresh}$) **do**
28: Send a ping and calculate RTT_{ping}
29: Sleep for T_{step}
30: **end while**
31: Hard timeout T_{hard} = Current time $-T_1$
32: **end if**

current T_{sleep} and system time T_1.

$$T_i = T_{sl-ini} + (i) \times T_{step} \tag{2}$$

Remark 1: The timeout occurs however it is not visible whether the flow expires due to hard or idle timeout? If the flow is removed due to T_{idle}, it means that T_{sleep} is the T_{idle} of the network since it is the idle time between last two consecutive pings. Otherwise, the flow expiry can also be due to T_{hard}, the total elapsed time since T_0.

To probe for a clear distinction, Host A sleeps for T_{sleep} and then sends a ping again. If the RTT_{ping} exceeds the T_{thresh}, it means that flow expiry is due to T_{idle}. On the other hand, if the RTT_{ping} does not exceed the T_{thresh}, this shows the removal of flow due to the T_{hard}. Intuitively, we set $T_{hard} = (T_1 - T_0)$.

Note that for T_{hard}, the discovered value is not accurate with the estimation error limited by the difference of the last two consecutive pings (T_{sleep}). The T_{hard} may have occurred anywhere in the duration of the last T_{sleep}. We reduce the error in estimation by pinging at a granular rate than T_{sleep}. The host knows the value of T_{hard} with error and that T_{sleep} is not the T_{idle} as entry is not expunged after sleeping for T_{sleep}. The host sends the pings gapped at T_{sleep} for 80% of T_{hard} (lines 22 - 25, Algorithm 1) and then starts sending pings after every T_{step} i.e., very small value as compared to T_{sleep}, to reduce the estimation error. Pings are sent until an entry expiry event occurs (line 27, Algorithm 1). The T_{hard} is now calculated (line 31) that is not dependent on the ratio between sums of all pings and actual value deployed in the network. Algorithm 1 thus returns the value of the first timeout and ascertains its nature as well.

B. COMPUTATION OF T_{idle} GIVEN T_{hard}

Algorithm 2 computes the value of T_{idle} given that timeout value discovered in Algorithm 1 was T_{hard}. If the value discovered is T_{idle}, Algorithm 3 is used to compute the T_{hard}. Only one of these algorithms will follow Algorithm 1. We first describe the working of Algorithm 2.

Algorithm 2 Find Idle Timeout Given Hard Timeout (T_{hard})

1: Sleep for T_{hard}
2: Initialize T_0 = Current system time and $T_{step} = 50ms$
3: Use T_{sleep} and T_{thresh} from Algorithm 1
4: Send a ping to install the flow
5: $T_{sleep} = T_{sleep} + T_{step}$
6: **if** $T_{sleep} \geq T_{hard}$ **then**
7: Idle/soft timeout $T_{idle} = 0$
8: Terminate
9: **end if**
10: Sleep for T_{sleep}
11: Send a ping and calculate RTT (RTT_{ping})
12: **if** ($RTT_{ping} < T_{thresh}$) **then**
13: Go to 5
14: **else**
15: T_1 = Current system time
16: **if** $(T_1 - T_0) \geq T_{hard}$ **then**
17: $T_0 = T_1$
18: Go to 5
19: **else**
20: Idle timeout $T_{idle} = T_{sleep}$
21: **end if**
22: **end if**

We use the value of T_{step} as 50 ms and calculate RTT_{ping} using sleep increments of T_{step} to increase the time gap between successive pings as in Algorithm 1. These RTT_{ping} are compared with T_{thresh} to detect whether the flow has expired or not. In case, the flow is not expunged, the process repeats with an increased value of T_{sleep}. This is repeated till the value of T_{sleep} exceeds the T_{hard} whereby the Algorithm terminates by declaring that no value of T_{idle} has been set (lines 6-9, Algorithm 2). In case RTT_{ping} exceeds the T_{thresh} indicating that the flow has expired, we note the current system time as T_1 and calculate the total elapsed time $(T_1 - T_0)$ since the flow was originally installed. If this total

elapsed time is equal or more than the T_{hard}, the flow expires due to the T_{hard}. This happens because we are incrementally testing for the T_{idle}.

$$T_{Elapsed} = T_0 + \sum_{i=1}^{n} (T_{sl-ini} + (i) \times T_{step}) \quad (3)$$

Example 1: Consider a SDN with T_{hard} 30 sec, T_{idle} 15 sec and T_{sleep} 3 sec. In Algorithm 2, using T_{step} of 50 msec, the host sends a ping and waits for 3.05 sec before sending another ping and waits for 3.10 sec. After sending the 2nd ping (testing for T_{idle} of 3.10 sec), total elapsed time is 6.15 sec since T_0. This relationship is expressed in the form of Equation 3 where T_{sl_ini} represents the initial T_{sleep}. With 10 pings, the total time elapsed will be around 32.75 sec. At the 11th ping T_{sleep} will be 3.55 sec that is still less than T_{idle} value of 15 sec. However, a new flow is installed by this ping as the previous entry is removed due to total elapsed time being greater than T_{hard}. In such a case, we change the reference time of flow installation and proceed with the last probed value of T_{idle} (lines 16-18, Algorithm 2). The total number of rounds iterations (n_{rounds}) can be found using Equation 4. Finally, the algorithm terminates when the T_{idle}, if set, is found.

$$n_{rounds} = \frac{T_{Elapsed}}{T_{hard}} = \frac{T_0 + \sum_{i=1}^{n} (T_{sl-ini} + (i) \times T_{step})}{T_{hard}} \quad (4)$$

C. COMPUTATION OF T_{hard} GIVEN T_{idle}

We describe Algorithm 3 to find the T_{hard} given we have discovered T_{idle} from Algorithm 1. In Algorithm 3, host A sleeps for T_{sleep} to remove any existing entry in the FET. We use T_{sleep} value as 75% of T_{idle} to avoid the removal of entry during probing due to T_{idle}. Host A sends ping and calculates RTT_{ping} to check for new flow installation. After each ping, current system time T_1 is noted and total elapsed time ($T_1 - T_0$) is checked. Lines 9 - 11 in Algorithm 3 specifies the threshold of the range (10 times the value of T_{idle}). If T_{hard} is found within this range, total elapsed time (current time $- T_0$) takes the value of T_{hard}, otherwise it declares the T_{hard} equal to 0. Similar to Algorithm1, we now fine tune the estimated value of T_{hard} (lines 17-27, Algorithm 3).

D. FLOW MATCHING RULES

In this module, we detect elements of firewall policy implemented by the controller. The controller allows/blocks installation of a flow based on certain match fields. Forwarding mechanism in SDN's switches is based on these match fields. Every new packet arriving is matched for certain fields in already existing flow rules. If matched, then the respective action specified in the flow rule is applied otherwise, the flow miss event occurs at the switch and the packet is forwarded to the controller. The increase in match fields means more entries are required for uniquely matching the installed criteria. We find out the flow match fields rules that are not blocked for installation by the controller. In other words, we find the fields, whose manipulation yields new entries at the FETs. This information is important when an adversary consumes the maximum resources of the SDN by installing as many flows as possible for producing a DoS scenario. The compromised host first fingerprints the SDN to determine the allowed/blocked match fields to populate a database of information about the implemented firewall policy. For efficiency reasons, the blocked match fields are not used while launching an attack.

Algorithm 3 Find the Hard Timeout Given the Idle Timeout (T_{idle})

```
1:  Sleep for T_idle
2:  T_0 = Current system time
3:  Use T_thresh from Algorithm 1
4:  T_sleep = 0.75 * T_idle
5:  Sleep for T_sleep
6:  Send a ping and calculate RTT_ping
7:  if (RTT_ping < T_thresh) then
8:      T_1 = Current system time
9:      if (T_1 − T_0) ≥ (10 ∗ T_idle) then
10:         Hard timeout T_hard = 0
11:         Terminate
12:     else
13:         Go to 5
14:     end if
15: else
16:     Hard timeout T_hard = Current system time −T_0
17:     Note current system time as T_2
18:     while (T_2 − T_1) < (0.8 * T_hard) do
19:         Note current system time as T_2
20:         Sleep for T_sleep
21:         Send a ping
22:     end while
23:     while (RTT_ping < T_thresh ) do
24:         Send a ping and calculate RTT_ping
25:         Sleep for T_step
26:     end while
27:     Hard timeout T_hard = Current time −T_2
28: end if
```

Allowed match fields are discovered by comparing RTT of each new packet with a unique value of a particular match field and RTT_{thresh}. If that particular match field is allowed, two different packets with different match fields are classified as belonging to two different flows based on changed RTTs due to flow installation. On the other hand, if the flow is blocked by the controller, a timeout occurs. Note that limited information can be gathered for blocked flows when there is only one compromised host in the network (Figure. 1). In this case, it is hard to determine if a particular flow is blocked by the controller policy or by the receiving host firewall. The situation improves when we have access to two hosts under control whereby it is assured that receiving host has not blocked any port or any flow at its firewall. Consequently, it is inferred that whatsoever is being dropped is due to the controller firewall policy.

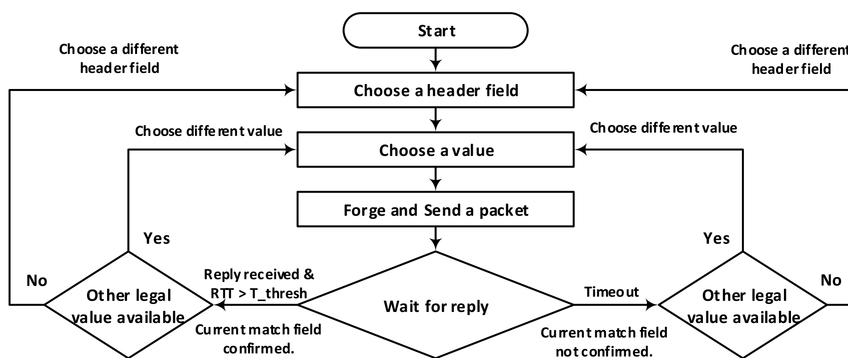


FIGURE 3. Flow chart for match fields fingerprinting.

The actual process of fingerprinting the match fields is quite straight forward. The host picks a match field and forges a packet using a random legal value of the match field. The host now waits and checks whether a new flow has been installed for this forged packet or not (using RTT_{thresh}). If the packet is dropped for a value of the match field, the algorithm marks this value as not allowed. This is followed by choosing another legal value for the same match field (keeping all other fields as unmodified). For example, at the transport layer, we probe for different TCP port numbers (Flow chart in Fig. 3).

RTT measurements for each matched field depends upon that particular header field communication behaviour. For example, RTT for ICMP Ping is calculated on the reception of the reply sent by the host (Echo Reply). Similarly, in case of TCP, if the targeted host is listening on a particular port being probed then SYN + ACK is received in the response of a SYN packet, while RST + ACK is received otherwise. However, no response is received if a particular port TCP is blocked either by the host firewall or by the network policy. Note that nested matching is also possible. The TCP flag field is a common example for such a scenario where TCP flag field is treated as a separate match field along with TCP port numbers.

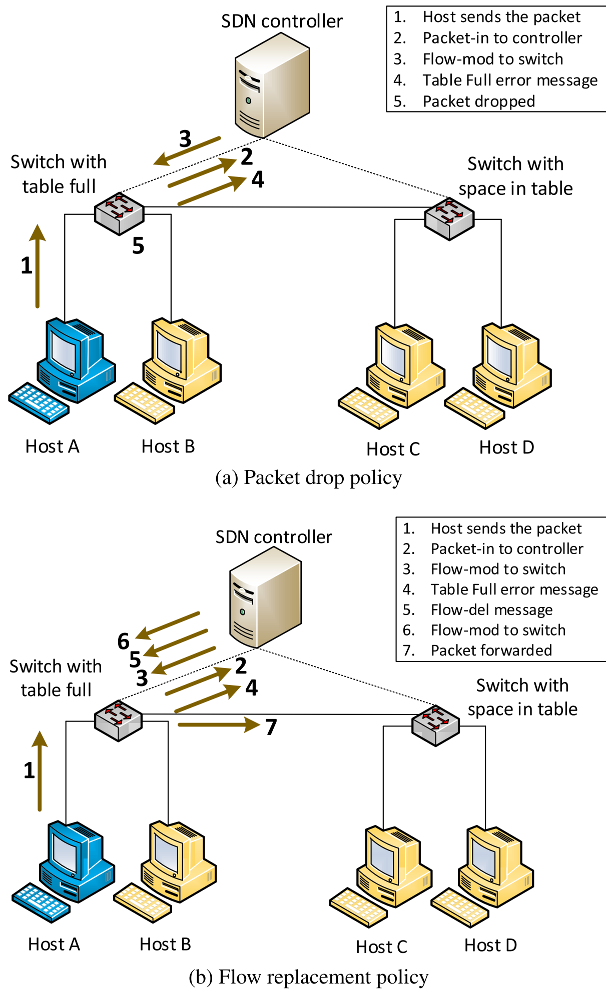
OpenFlow [2] has become a de facto standard south-bound API for communication between the controller and the switches. In Openflow 1.0, there were only 12 match fields available. However, the list of supported match fields has grown to 44 in the latest available version Openflow 1.5. The provision of existing 44 match fields in OpenFlow 1.5 provides a lot of flexibility for network applications, however, these also impose a threat to a SDN deployment. For example, generally speaking, if a maximum number of 44 match fields are being used, one might need least effort to overflow the switch FETs by launching an attack with various protocols and their unique header values (ports, addresses etc.).

There are match fields that are supported in all versions of OpenFlow e.g. TCP ports, Ethernet and IP addresses, to name a few. However, support of few newer match fields is enabled in specific versions. For instance, MPLS was not part of OpenFlow 1.0 specifications but has been incorporated since version 1.1. We looked at the possibility of predicting the OpenFlow version based on supported match fields. We observed that this prediction is dependent on the match fields that are employed/enabled in the SDN deployment. The issue is that any later version can be down-graded to work in an earlier version mode by switching off the support of new features making the prediction in-accurate. For example, MPLS BoS (Bottom of Stack) was introduced as a Match field in OpenFlow 1.3 and is also supported by all later versions. So a network supporting MPLS BoS can be identified as using OpenFlow 1.3 or any later version. But on the other hand, if a network has deployed OpenFlow 1.5 and but have only enabled basic match fields that were introduced in OpenFlow 1.0 (working in OpenFlow mode 1.0), the prediction based on match fields can only identify it as version 1.0.

E. CONTROLLER REACTION POLICY FOR TABLE FULL EVENT

SDN enabled switches has very limited memory to store entries in FETs. Usually, these switches are equipped with both hardware flow tables maintained in either Binary or Ternary content-addressable memory (BCAM or TCAM) and software flow tables maintained in SDRAM of the switch. As a rule of thumb, hardware flow tables work at almost line speed but can support only a few hundreds of flow rules. If the matching rule is located in the software flow table, the forwarding performance is degraded by up to two orders of magnitude [19]. Furthermore, if the FETs are full, the performance degrades further due to extra interactions required between the switch and the controller to delete/add flows from the table to accommodate newly arrived requests [11].

In this section, we elaborate our proposed technique to find the controller reaction policy when a flow arrives at a switch while its FET is full to the capacity. There are two common mitigating strategies for dealing with overflowing tables at switches. The default controller applications do not react to table full error messages, as they rely upon auto removal of flow entries through the timeout mechanism. The switch sends a Packet-In informing the controller about the arrival of a flow, controller replies with Flow-Mod instructing the switch to install the flow. At this stage, the switch replies



(a) Packet drop policy

(b) Flow replacement policy

FIGURE 4. Mitigating table full scenario.

with Table-Full OpenFlow message to inform the controller that there is no space left in the FET. The controller does not respond to this error message and eventually, this flow is considered as dropped (Figure. 4a). This controller behaviour is evident when the attacker starts receiving consistent packet drops within the limits of idle/hard timeouts.

Number of dropped packets (D_p) can be estimated by an end host using Equation 5, where we represent the packet forging rate as (r_{p-g}) and total number of entries supported in a FET by n_{FE}. Similarly, an end host can use Equation 6 to estimate the size of FET, but it can be of any bottle neck switch between the two corresponding end hosts.

$$D_p = \left(T_{hard} - \frac{n_{FE}}{r_{p-g}}\right) \times r_{p-g} \tag{5}$$

$$n_{FE} = \left(T_{hard} - \frac{D_p}{r_{p-g}}\right) \times r_{p-g} \tag{6}$$

Another mitigating strategy in which the controller instructs the switch to remove specific flow(s) whenever it receives the table full error message and install the new requested flow (Figure. 4b). The selection of flows to be removed can be based on port statistics or on match fields or perhaps the oldest installed flow is removed. Note that in this strategy, there will be an increased load on the controller as well as increased RTT due to additional messages exchanged between the switch and the controller. On a table full event, switch sends the error message Table-Full to the controller who now sends the flow deletion message and then finally sends the Flow-Mod rule to install the new flow resulting in at least two extra messages (labelled as 5 and 6 in Figure. 4b). This increased round trip time $RTT_{reinstall}$ is significantly higher than normal flow installation time RTT_{FE}.

The table full state can thus be inferred from consistent packet drops (first strategy) or when the RTT of subsequent packets increases to $RTT_{reinstall}$ in the 2nd strategy. The attacker can estimate the number of unique flows can be installed in the FET till the FET is inferred as full. However, the exact number of flows that can be accommodated at switches are unpredictable because in a production network there would be many existing flow entries for supporting various ongoing communications. Nevertheless, the attacker would know specifically when its goal of exhausting the capacity of a switch's FET has been achieved. The only effort required from this point onwards would be to sustain the flow of specific packets to keep the SDN fabric in its current state of exhaustion.

F. TOPOLOGY DISCOVERY

The information about the network topology of the SDN is very useful for an attacker enabling the launch of more effective attacks on the network. On one hand, the attacker would prefer to target a correspondent host that involves multiple switches so that controller will have to install flows at multiple hops. On the other hand, a correspondent node attached with the same switch as the attacker would be ideal for quick launch and feedback of the attack. Detailed information about the network topology would thus enable the attacker to cherry-pick the target nodes to effectively overwhelm the FETs of a target switch.

We employ the same difference in RTT technique to learn about the network topology in this module. As explained earlier, there is a significant difference in RTT, when a flow already exists (RTT_{avg}) as compared to when a controller installs a new flow entry (RTT_{FE}). This difference increases further for a path involving multiple switches. For this module, we have assumed the well known fat-tree topology (Figure 5), a special use-case of SDN deployment in data centres as it provides many redundant links [20]. It consists of three layers, named as edge layer, aggregation layer and the core layer. Edge switches are also known as ToR (Top of Rack) switches. It allows the full bisection bandwidth (whereby a host gets the full bandwidth of its interface while communicating with another host) within all hosts of the same rack. Interconnected ToR switches using aggregation layer switches only i.e, without involving core layer forms a pod, as highlighted in dotted squares in Figure. 5

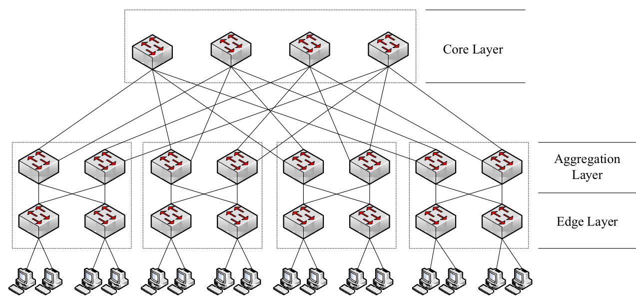


FIGURE 5. Fat tree topology.

The topology discovery mechanism is based on the observation that communication between any two hosts in fat-tree topology involves either one, three or five switches. An in-rack communication involves one switch only while in-pod communication involves three switches and inter pod communication involves five switches at the maximum. If an adversary knows the IP addresses of hosts inside the network (using a network scanning application), it can ping these hosts and observe their respective RTTs. These hosts can be sorted into in-rack, in-pod and inter pod groups as their respective RTTs will have an increasing trend as the packets are traversing one, three and five switches respectively. Especially, when a new flow is being installed on each switch, the RTT_{FE} would be a candidate parameter for this decision making. We have not considered $RTT_{reinstall}$ (table full scenario) because, to observe the value of $RTT_{reinstall}$, we have to first launch a successful attack on the network. We comment that there is no point in discovery/re-affirmation of the network topology at this post-attack stage.

V. PERFORMANCE EVALUATIONS

These experiments are performed on a system with the specification given in Table 3. Mininet [21] was utilized to run an emulated network where each node of the network is provided as a virtual machine with individual Linux kernel enabling configurable link parameters such as bandwidth. RYU [22] was used as a SDN controller. Our proposed algorithms have been implemented in Python. Scapy [23] API is used to write python scripts for packet forging (for determining flow rules and flow table size) purpose. We tested on the commonly used match fields such as IP addresses, Protocol types, MAC addresses, transport layer protocols (TCP & UDP), Transport layer port numbers, TCP flags, ICMP, ICMP types and ToS. Python scripts return the RTT to analyze the flow installation time, which has been cross checked by a packet analyzer, Wireshark. We use the topology shown in Figure 1 for discovering the timeouts, flow match rules and controller policy for table full events, while fat-tree topology (fig.5) was employed for classification of different hosts.

A. GRANULARITY OF STEP SIZE

In Algorithm 1 (Section IV) selection of step size, T_{step}, affects the performance of the proposed fingerprinting mechanism. Intuitively, the step size is inversely proportional to the overall running cost of these algorithms and directly proportional to the estimation error in timeouts. However, the error is also dependent on the actual values being deployed in the network especially in the case of T_{hard}. To illustrate this point, let's suppose a network is using T_{idle} as 10 sec and T_{hard} 30 sec with Algorithm 1 implemented until line 20. If we use step size as 100 msec in Algorithm 1, within 25 increments the total elapsed time will be 32.5 sec and the last sleep time will be 2.5 sec. Now the flow entry will be expunged by the controller due to T_{hard} and host will detect an error of approximately 2.5 sec. Comparing this with the case when we choose step size of 125 msec, within 22 increments the total elapsed time will be 31.625 sec and the last sleep time will be 2.75 sec resulting in an error of approximately 1.625 sec. Figure 6 show this relationship between step size and the error in estimation where the error is fluctuating depending on the values of T_{hard} and T_{step}. We thus have to introduce additional processing (lines 21-31 in Algorithm 1) to reduce this estimation error resulting in the error shown in Figure 6 where even the step size of 250 ms is producing error within 1 sec.

TABLE 3. System Specification.

Name	Specification
CPU	Intel Core (TM) i5-7200 CPU 2.7 GHz
RAM	8 GB - DDR4 2133 MHz
GPU	2 GB - GeForce 920 MX
OS	Ubuntu 16.04.03 LTS
Emulator	Mininet 2.3.0d1 [21]
Packet Forger	Scapy [23]
Packet Analyzer	Wireshark [24]

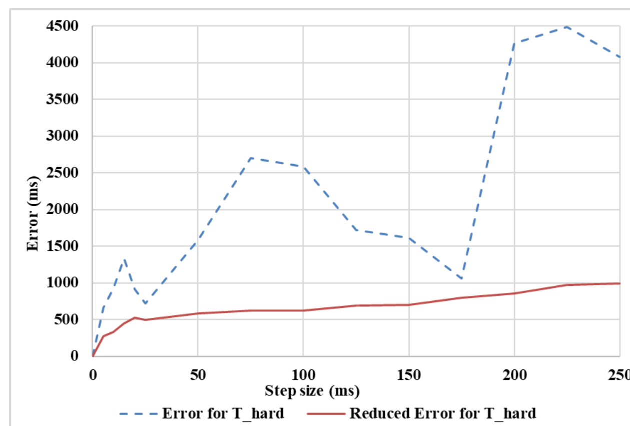
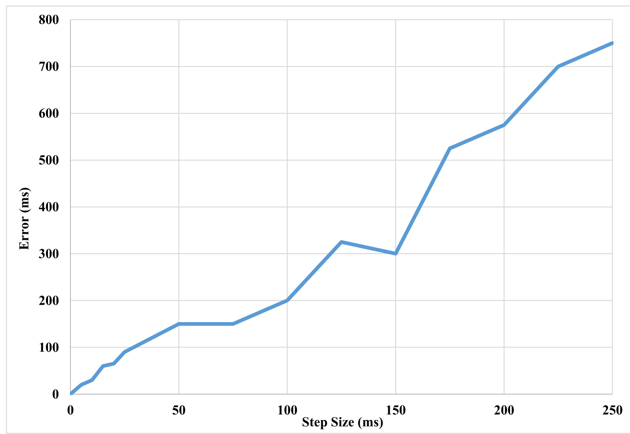
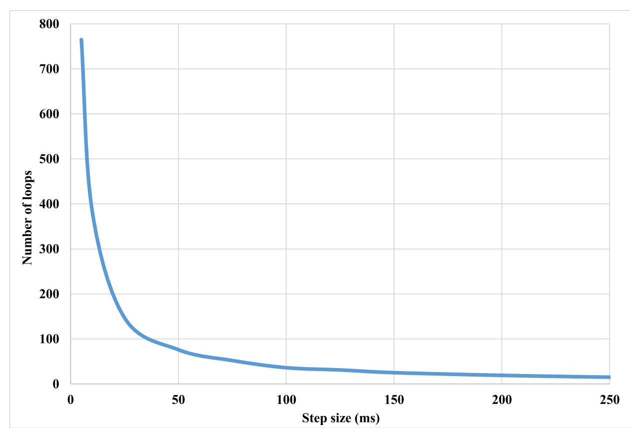


FIGURE 6. Granularity of step size in Algorithm 1.

Once the value of T_{hard} is discovered through Algorithm 1, this value is used again in Algorithm 2 to find the T_{idle}. The fingerprinting mechanism also induces error in the estimation of T_{idle} as shown in figure 7 where the estimation error for T_{idle} values using T_{step} of 100 msec is about 200 msec (2%) that increases to 750 msec for step size of 250 msec (7.5%). On the other hand, step size has a significant impact on the

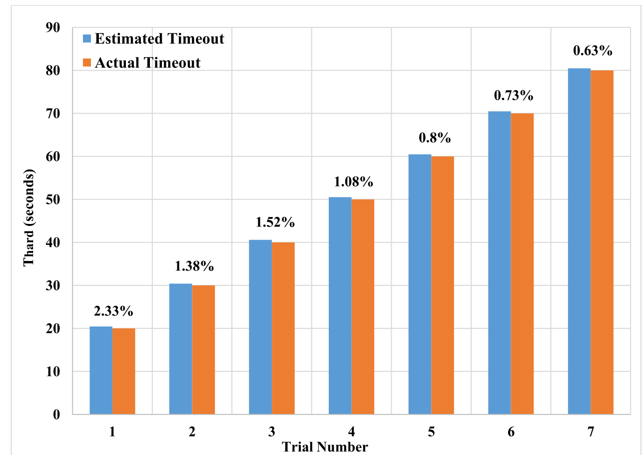


(a) Step size and Error for T_{idle}

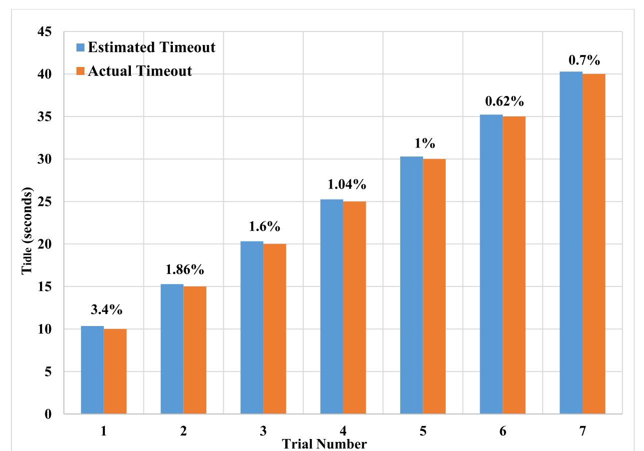


(b) Step size and Number of loops in Algo 2

FIGURE 7. **Granularity of step size in Algorithm 2 for T_{idle}.**



(a) T_{hard} discovery



(b) T_{idle} discovery

FIGURE 8. **Timeouts discovery using Algorithm 1.**

number of loops i.e., the occurrence of T_{hard} before discovering the T_{idle}. Note that number of loops when multiplied with T_{hard} donates the overall simulation time. The reduction in the number of loops becomes almost flat beyond a step size of 200 ms (Figure 7b). We choose the value of 100 ms for T_{step} for rest of our experiments as this value gives a low error in estimation for both T_{hard} and T_{idle} with reasonable execution time based on the number of involved loops.

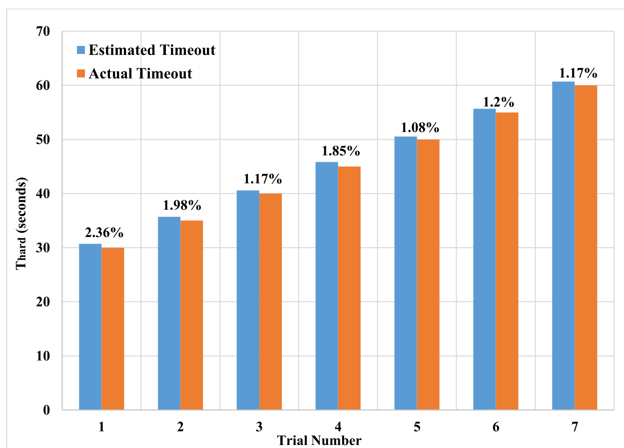
B. TIMEOUT ESTIMATIONS

We run a set of experiments to ascertain the error in estimation for different combinations of the time out values given in Table 2. Values reported in these experiments are the average of five simulations runs with the step size of 100ms. Figure 8a shows the difference between actual (we used values ranging between 20ms to 80 ms) and estimated timeout values (using Algorithm 1) for the case when only T_{hard} is set by the administrator. It highlights different timeout values and their respective estimated values along with the %age difference. Algorithm 1 show low estimation error when discovering the T_{hard} values with the error getting amortized with higher values of T_{hard}. Figure 8b shows the error in the estimation of T_{idle} when this timeout value is discovered by Algorithm 1 (Combination C in Table 2, T_{hard} is set as 0). The experiment was conducted with T_{idle} values ranging between 10 sec to 40 sec. The results show the maximum average estimation error of about 3.4% for the case when T_{idle} is being estimated by Algorithm 1 for the actual timeout value of 10 sec.

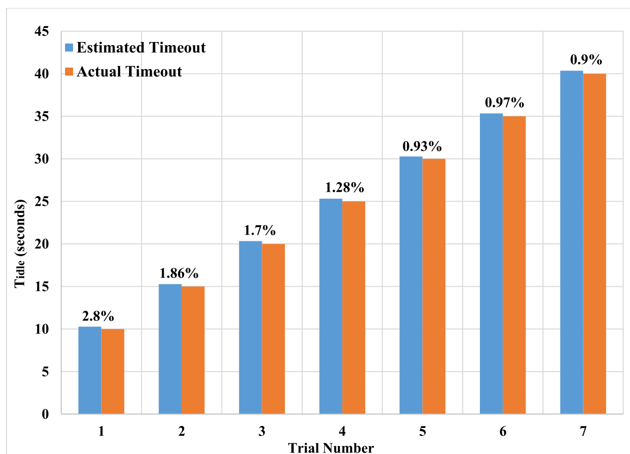
The combination D listed in Table 2, where both T_{hard} and T_{idle} values are being used in a deployment can be discovered using Algorithm 1 followed by either Algorithm 2 or Algorithm 3. The actual sequence of algorithms depends on the ratio between the two timeout values and the T_{step}. For example, let's consider a scenario where the T_{idle} is set as 6 sec and T_{hard} is set as 40 sec and the step size is 100 ms. In this case, a flow entry will be expunged within 28 sec because total elapsed time would be 40.6 sec, resulting in the discovery of the T_{hard} value first. A1 will be followed by Algorithm 2 for the discovery of T_{idle}. Whereas, if we change the step size to 750ms than after 8 pings, the flow entry will be expunged because of T_{idle} and at that instance total elapsed time would be 27 sec. So, Algorithm 1 will be

followed by Algorithm 3 for the discovery of T_{hard}, if that value exists. Given the normal ranges for the step size, idle and hard timeouts, it is thus more likely that T_{hard} will be discovered first by Algorithm 1 rather than the T_{idle}.

We have considered both combinations of these algorithms and have plotted the estimation error percentage in Figures 9a and 9b. When Algorithm 1 is followed by Algorithm 2, the value of T_{hard} was set as 60 sec and the T_{idle} values were changed between 10 to 40 sec. For the case when Algorithm 1 is followed by the Algorithm 3, the value of T_{idle} was 6 sec and in this case only, T_{step} was set at 750 ms whereas in all remaining experiments T_{step} was set at 100 ms. For these set of experiments, the maximum average estimation error was 2.8% for the case when actual T_{idle} is 10 sec, T_{hard} is 60 sec with a step size of 100ms, resulting in the combination Algorithm 1 followed by Algorithm 2 (Figure 9b).



(a) T_{hard} discovery using Algorithm 1 & 3



(b) T_{idle} discovery using Algorithm 1 & 2

FIGURE 9. Timeouts discovery using combinations of Algorithms.

C. CONTROLLER's REACTION AT TABLE FULL EVENT

We discussed two mitigating policies for controller's reaction against table full scenarios (see Section IV-E). In the default case, where the controller is not taking any action and letting the new request timeout resulting in packet drops, packets are not allowed to traverse any further in the network. Other switches in the path towards the destination thus remain unaffected. Figure 10 shows a scenario where the table full events result in 100% packet drops until the T_{hard} (set at 60 sec) starts removing earlier installed flow entries. For this experiment, we set the switch buffer to accommodate 1024 flows and set the packet generation rate at the host as 45 packets per second resulting in the installation of 90 flows per second. It can be observed that the FET becomes full after only 11.3 sec of simulation time. Consequently, 100% packets were dropped due to table full until T_{hard} started expunging old flows i.e., after 60 sec since the installation of first flow in FET. First packet drop interval lasted for almost 48.7 sec (T_{drop}) and 2211 packets were dropped. After that new flows are admitted again for approximately 11.5 sec i.e., until the table was again full. Packet drops started again due to full flow table at 71.5 sec. This repeated fashion of packet drops and new flow installations due to T_{hard} resulted in about 80% of total packets being dropped.

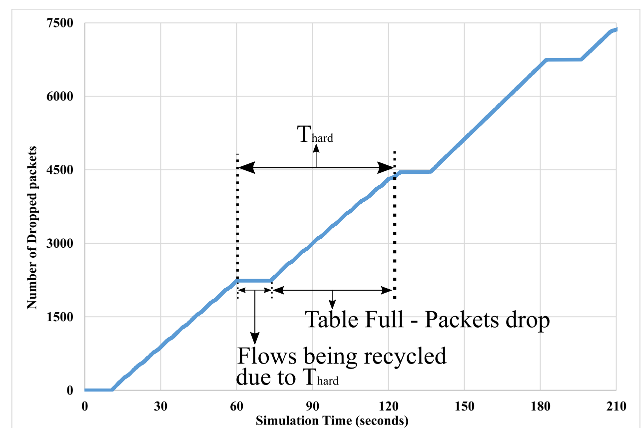
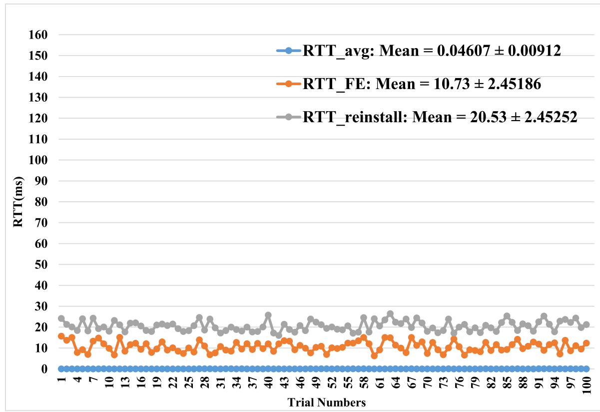


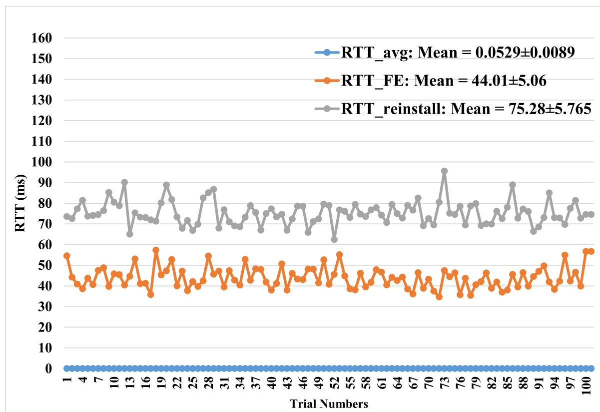
FIGURE 10. Packet drop in default reaction of the controller to table full event.

The second available mitigation policy is the flow replacement/swap policy where the controller directs the switch to remove an earlier installed flow and replace it with the new request. In this case, the controller's resources will be under additional stress although there will be no packet drops. It affects the performance of the whole network as the RTT of flow installation messages will increase, due to extra communications involved in flow removal and new flow installation, especially if it involves multiple switches between the two end hosts.

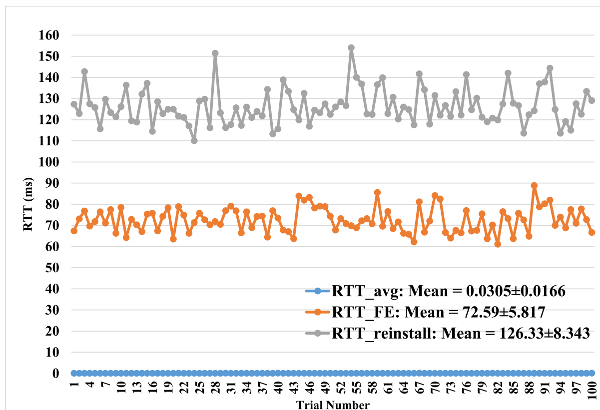
The normal RTT (RTT_{avg}), flow installation RTT (RTT_{FE}) and flow replacement RTT ($RTT_{reinstall}$) is shown in Figure. 11a, 11b and 11c for one, three and five switches respectively. For three and five switches, the mean $RTT_{reinstall}$ value is approximately 1.75 times the mean RTT_{FE}. Moreover, in these graphs RTT_{FE} and RTT_{avg} also reflects the match fields testing i.e., when a new flow entry is being installed for each forged packet with new header field value



(a) RTTs of in-rack communication



(b) RTTs of in-pod communication



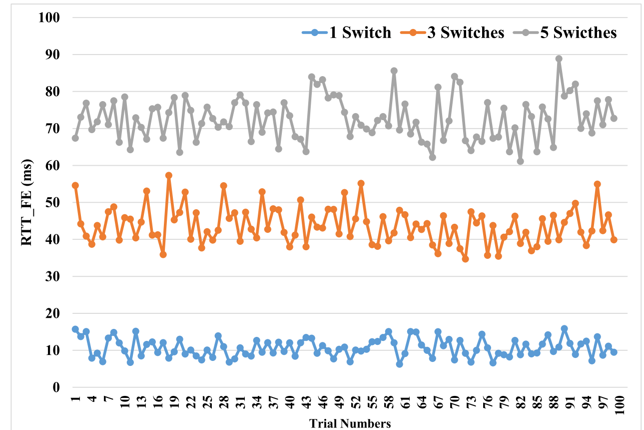
(c) RTTs of inter-pod communication

FIGURE 11. Measurements of different RTTs for different hosts.

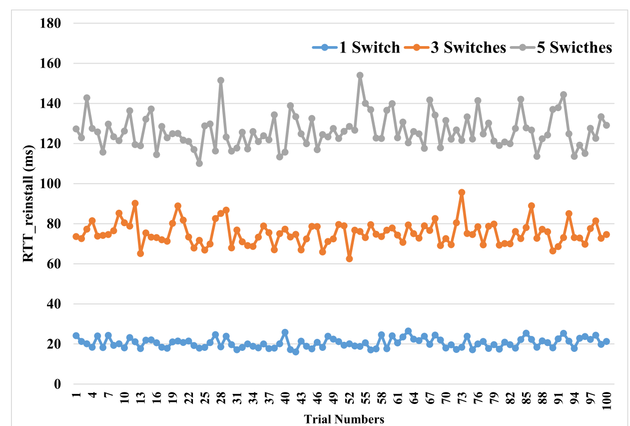
and packet with same value when transmitted directly on the data plane. Each category in each of these plots has been tested for 100 packets. Flow replacement policy thus affects the performance of the network, by increasing the RTT.

D. TOPOLOGY DISCOVERY

For the topology discovery module, we employ a fat-tree topology (Figure 5) that was implemented in the Mininet and different flows were initiated from a single host to all



(a) Variation in RTT_{FE}



(b) Variation in $RTT_{reinstall}$

FIGURE 12. RTTs comparison for different hosts.

other hosts in the SDN deployment to monitor different types of RTTs i.e., RTT_{FE}, RTT_{avg} and $RTT_{reinstall}$. We observed that differences in RTT_{avg}, when the flow is only utilizing the data plane, is not a good measure to differentiate between different neighbours located across the data centre (Figure 11). On the other hand, RTT_{FE} and $RTT_{reinstall}$ proved to be reliable metrics to identify the location of the correspondent hosts. For example, sending packets to a host located in the same pod results in about 4 times the RTT_{FE} (average 44 ms) as compared to when the packet is going through the same rack (average 10.7 ms, going through the same ToR switch) as shown in Figure 11a. This increased to 72.6 ms (Figure 11c) when the communication involved inter-pod communication (traversing a core switch) that is about 6.7 times the RTT_{FE} for communication with the same rack neighbour. We also plot the variation in RTT_{FE} in Figure 12a and $RTT_{reinstall}$ in Figure 12b to illustrate the differences when flow traverses 1, 3 or 5 switches in the data center. The results are even better with $RTT_{reinstall}$ being used as the metric showing clear differentiation for RTT observed for intra-rack (1 switch), intra-pod (3 switches) and inter-pod (5 switches) communications. However, to observe the values of $RTT_{reinstall}$, we have to first launch an attack to exhaust the

FET capacity. Topology discovery by launching an attack, at a belated stage, loses its purpose to help launch an efficient and targeted attack at the network. These results demonstrate that it is possible to identify the locations of the hosts, without launching a FET exhaustion attack, based solely on their initial flow installation time (RTT_{FE}).

VI. CONCLUSION & FUTURE WORK

We have presented efficient techniques to fingerprint the SDN allowing the discovery of its implemented policy parameters. These parameters are of immense importance for an attacker who can exploit the characteristics of the network to launch an effective attack to force denial of service conditions. This research work also provides insight into the modalities of selecting different policy parameters and its effect on the resilience of SDN in light of a denial of service attack on its data plane. In future work, we plan to explore the use of dynamic policy parameters rather than static parameters. A network should change its timeout values according to the traffic load and existing space in the FET. Similarly, if the FET is full, the controller should reduce the number of match fields so that more incoming packets match with a single flow entry. This dynamicity is currently not supported in OpenFlow and is worth further research.

REFERENCES

[1] N. McKeown, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[2] *Openflow Switch Specifications Version 1.5.1*. Accessed: Jun. 29, 2020. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-swit%ch-v1.5.1.pdf

[3] P. Anand, M. Hlady, D. Kampmann, and J. Scheurich, "Packet processing in an openflow switch," U.S Patent 10 439 962, Oct. 8, 2019.

[4] S. Zerkane, D. Espes, P. Le Parc, and F. Cuppens, "Vulnerability analysis of software defined networking," in *Proc. Int. Symp. Found. Pract. Secur.*, 2016, pp. 97–116.

[5] C. Yoon, S. Lee, H. Kang, T. Park, S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Flow wars: Systemizing the attack surface and defenses in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3514–3530, Dec. 2017.

[6] A. Shaghaghi, M. A. Kaafar, R. Buyya, and S. Jha, "Software-defined network (SDN) data plane security: Issues, solutions, and future directions," in *Handbook of Computer Networks and Cyber Security*. Cham, Switzerland: Springer, 2020, pp. 341–387.

[7] Q. Yan, F. R. Yu, Q. Gong, and J. Li, "Software-defined networking (SDN) and distributed denial of service (DDoS) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 602–622, 1st Quart., 2016.

[8] J. C. C. Chica, J. C. Imbachi, and J. F. Botero, "Security in SDN: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 2020, Oct. 2020, Art. no. 102595.

[9] A. Azzouni, O. Braham, T. M. Trang Nguyen, G. Pujolle, and R. Boutaba, "Fingerprinting OpenFlow controllers: The first step to attack an SDN control plane," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2016, pp. 1–6.

[10] J. Cao, Z. Yang, K. Sun, Q. Li, M. Xu, and P. Han, "Fingerprinting SDN applications via encrypted control traffic," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses*, Beijing, China, Sep. 2019, pp. 501–515. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/cao

[11] G. Zhao, L. Huang, Z. Yu, H. Xu, and P. Wang, "On the effect of flow table size and controller capacity on SDN network throughput," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–6.

[12] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (Datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 123–137.

[13] H. Cui, G. O. Karame, F. Klaedtke, and R. Bifulco, "On the fingerprinting of software-defined networks," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 10, pp. 2160–2173, Oct. 2016.

[14] Z. J. Zeitlin, "Fingerprinting software defined networks and controllers," Graduate School, Air Force Inst. Technol., Wright-Patterson AFB, OH, USA, Tech. Rep. AFIT-ENG-MS-15-M-067, 2015.

[15] R. Kandoi and M. Antikainen, "Denial-of-service attacks in openflow SDN networks," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, Oct. 2015, pp. 1322–1326.

[16] Y. Zhou, 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," *Secur. Commun. Netw.*, vol. 2018, Jan. 2018, Art. no. 4760632.

[17] M. Zhang, J. Hou, Z. Zhang, W. Shi, B. Qin, and B. Liang, "Fine-grained fingerprinting threats to software-defined networks," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, Aug. 2017, pp. 128–135.

[18] J. Suarez-Varela and P. Barlet-Ros, "Towards a NetFlow implementation for OpenFlow software-defined networks," in *Proc. 29th Int. Teletraffic Congr.*, Sep. 2017, pp. 187–195.

[19] P. Rygielski, M. Seliuchenko, S. Kounev, and M. Klymash, "Performance analysis of SDN switches with hardware and software flow tables," in *Proc. 10th EAI Int. Conf. Perform. Eval. Methodologies Tools*, 2017, pp. 1–7.

[20] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 63–74.

[21] I. Z. Bholebawa, R. K. Jha, and U. D. Dalal, "Performance analysis of proposed OpenFlow-based network architecture using mininet," *Wireless Pers. Commun.*, vol. 86, no. 2, pp. 943–958, Jan. 2016.

[22] M. T. Islam, N. Islam, and M. Al Refat, "Node to node performance evaluation through RYU SDN controller," *Wireless Pers. Commun.*, vol. 112, pp. 555–570, Jan. 2020.

[23] G. M. Kumar and A. Vasudevan, "D-SCAP: DDoS attack traffic generation using scapy framework," in *Advances in Big Data and Cloud Computing*. Singapore: Springer, 2019, pp. 207–213.

[24] *Wireshark*. Accessed: Jun. 29, 2020. [Online]. Available: https://www.wireshark.org

BILAL AHMED received the B.S. degree in electrical (telecommunication) engineering from the COMSATS Institute of Information Technology, Islamabad, and the M.S. degree in information technology from the School of Electrical Engineering and Computer Science (SEECS), National University of Sciences and Technology (NUST), Pakistan. He is currently a Lecturer with the Computer Science Department, NUST. His research interests include data science, ML-based optimization models, software-defined networks, and data center networking.

NADEEM AHMED received the M.S. and Ph.D. degrees in computer science from the University of New South Wales (UNSW), Sydney, Australia, in 2000 and 2007, respectively. He was the Head of the School of Electrical Engineering and Computer Science (SEECS), National University of Sciences and Technology (NUST), Pakistan. He is currently a Senior Research Fellow at the Cyber Security Cooperative Research Centre (CSCRC), Australia. His research interests include cybersecurity, wireless sensor networks, and software-defined networking.

ASAD WAQAR MALIK (Senior Member, IEEE) received the Ph.D. degree in computer software engineering from the National University of Sciences and Technology (NUST), Pakistan. He was a Senior Lecturer at the University of Malaya, Malaysia. He is currently an Assistant Professor with the School of Electrical Engineering and Computer Science, NUST. His research interests include parallel and distributed simulation, cloud computing, and large-scale networks. He is the PI on the distributed simulation projects, such as CloudNetSim, FogNetSim, and SEECSim.

MOHSIN JAFRI received the Ph.D. degree in computer science from Università Ca' Foscari Venezia, in 2019. He has been an Assistant Professor of computer science at the National University of Sciences and Technology (NUST), Pakistan, since June 2019. His main research interests include communication system design, wireless sensor networks, and underwater sensor networks. He has contributed in developing network simulators and energy-efficient algorithms for underwater communication. Moreover, he has also developed stochastic models for the performance analysis of underwater sensor networks.

TAIMUR HAFEEZ (Graduate Student Member, IEEE) received the B.S. degree from the COMSATS Institute of Information Technology, Islamabad, Pakistan, in 2016, and the M.S. degree from the School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan, in 2018. He is currently pursuing the Ph.D. degree with the School of Computer Science, University College Dublin, Ireland. His research interests include the Internet of Things, machine learning, and software-defined networks. He is a Student Member of ACM, ACM SIGCSE, and the Irish Computer Society.