Detecting and Mitigating Denial of Service Attacks against the Data Plane in Software Defined Networks

Raphael Durner*, Claas Lorenz†, Michael Wiedemann*†, Wolfgang Kellerer*
*Technical University of Munich - {r.durner, m.wiedemann, kellerer}@tum.de
†genua GmbH, Kirchheim - {claas_lorenz, michael_wiedemann}@genua.de

***Abstract*—Software Defined Networking (SDN) introduces a new network architecture offering means of programmability through an externalized centralized control plane. As a result most security research addresses attacks against this central entity. Contrary to that, attacks against the data plane in SDN did not perceive a broad attention in the scientific community so far. In this work we discuss Denial of Service attacks against the data plane and their impact. We propose a tailored statistical detection approach as well as a lightweight countermeasure. We evaluate the detection by simulation and an analytical approach. Throughout this evaluation, we highlight the trade-off between detection speed and adaptability and show a way to tune the solution analytically. Our results show, that we can detect and mitigate attacks against the data plane in a lightweight and dependable way.**

I. Introduction

The uprise of Software Defined Networking (SDN) as a paradigm that separates the data from the control plane introduces new challenges in network security. Especially, in modern cloud environments where an attacker can get access to the network by simply renting a virtual machine, Denial of Service (DoS) attacks pose a serious threat.

Networking elements, like switches, process traffic according to the entries in their forwarding tables which are set by a logically centralized controller. SDN offers two major modes of operation – *proactive* and *reactive*. In the former case the controller presets all forwarding rules according to the configuration of the networking applications which provide the networking functionality, e.g. switching or routing. Packets that do not match any entry in the forwarding table are dropped by the networking element.

In reactive setups, on the other hand, a table miss results in a query to the controller. In the controller, the networking applications can make a decision based on a global view of the network's state. Then, they are able to enforce a network policy, e.g. routing, by individually forwarding packets, sending out packets, or setting up forwarding rules. The most prominent SDN protocol allowing both modes of operation is OpenFlow (see [1]) which also enables any hybrid approach with proactive and reactive elements.

Figure 1 shows the typical behavior of a reactive SDN setup:

1) A host sends a packet for a new connection which reaches an SDN switch. Then, the switch performs a lookup in its forwarding table.
2) Since the packet belongs to a new flow which is unknown to the switch, the packet is encapsulated into a *PacketIn* message and sent to the controller.
3) In the controller the PacketIn is processed by an SDN application which provides networking functionality like switching or routing. The applications decision may include sending a *FlowMod* message which installs a rule in the forwarding table. Also, the switch may be instructed to forward the original packet through any of its ports.
4) If the controller application has set up a rule in the switch to handle the flow, further packets belonging to that flow will be processed in the fast forwarding hardware without the need for additional communication with the controller.

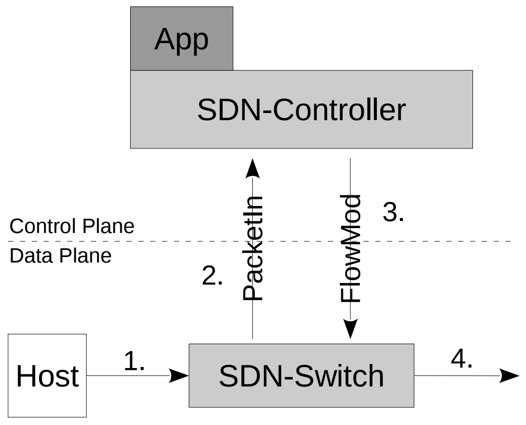

Fig. 1: Normal behaviour of a reactive SDN.

The logically centralized SDN controller represents a potential single-point-of-failure and many researchers have developed comprehensive approaches to deal with these threats [2]–[5]. In this work we focus on a much less investigated area of SDN security – the detection and mitigation of Denial of Service attacks against the data plane.

Forwarding tables have limited memory capacities. Typically, switches rely on Content Addressable Memory (CAM) that performs table lookups at line rate. Especially, Ternary CAM (TCAM) is expensive and therefore very small, ranging in the region of 1k to 2k entries. But also, regular and cheaper Binary CAM (BCAM) is limited to only a couple of 100k entries. An attacker with the ability to remotely trigger flow handling modifications that add entries to the flow tables can cause a DoS by exhausting the switches memory. Some works extend the available table size by using a combination of software and hardware flow tables [6], [7]. If the attacked devices make use of such techniques, the severity of the DoS is reduced, as software tables allow far more entries. Although, also software tables suffer from performance penalties for big table sizes, e.g. Open vSwitch uses a linear search to handle wildcard rules [8]. To deal with this threat we propose a tailored statistical approach for the detection of such an attack. Therefore, our main contributions are:

- A problem specific detection mechanism for attacks on the data plane, that is more comprehensive than existing approaches.
- A lightweight counter measure to stop attacking flows with one or very few flow rules.
- A novel evaluation by analytic means and simulation.

The remainder of this work is structured as follows. In Section II we give an overview of the related work in the field of DoS attacks in SDN followed by a detailed description of the attack model and our detection algorithm in Section III. Finally, in Section IV we provide our evaluation.

II. Related Work

There are several approaches in literature covering DoS in SDN with different perspectives. These approaches are shown in the following:

In [4] an approach for the detection of distributed DoS attacks against the controller is presented that relies on detecting deviations from a normal distribution of PacketIns in terms of destination addresses. An attack is indicated by a significant growth of new flows to a single host compared to the normal situation where new flows reach hosts evenly distributed. Therefore, an attack is indicated by a lower entropy calculated over a window of PacketIns. The emulation results look promising offering a detection rate between 95 to 100%, although most parameters like arrival distribution and network settings remain unclear. Nevertheless, they evaluated their approach using the destination address as fixed and the source address as varied parameter. The more parameters an attacker can shuffle the higher the entropy of the attack packets will be. As is, an attacker who can address the whole subnet under supervision is likely able to circumvent the detection completely. Therefore, it remains unclear whether this approach can be scaled to scenarios with manifold variable header fields, e.g. if the attacker controls a virtual machine in a cloud data center. Further, the approach does not yield information to quickly apply countermeasures against the attack.

A very notable approach called *FlowRanger* is provided by [9]. The basic idea is to classify PacketIns using a trust-level metric and enqueueing them with different priorities. This leads to a faster processing of PacketIns triggered by trusted hosts and a higher probability of untrusted PacketIns being dropped in high load situations. A host's trust level may be adapted over time due to its behavior. Although, invented for mitigating DoS attacks against the controller, this approach also helps to reduce the impact of attacks against the data plane. Since fewer malicious PacketIns are processed by the controller, this also reduces the rate of FlowMods. Nevertheless, *FlowRanger* reduces the attack's impact without removing its root cause – the initial triggering of PacketIns by an attacker. Also, with sufficient resources granted to the controller the approach becomes less effective since also the low priority queues are processed fast. However *FlowRanger* might be a suitable supplement to our work since it helps to reduce an attack's impact before its detection due to its different focus.

In [10] a technique is proposed to minimize the impact of a DoS against the controller and the switch tables by optimizing rule expiration and rule aggregation in the switches. These measures lower the impact of attacks flooding the flow tables by reducing the overall resource usage without tackling the attacks' root cause. Additionally, the reduction of the expiration time could increase the load of the controller and may add delays to flows which timeout prematurely. Nevertheless, this approach is complementary to our efforts and could help in building a robust and efficient system.

Further, [5], [11] propose *FloodGuard*, an approach that tries to anticipate the behavior of the controller as well as the applications and set up rules in the switches proactively. These rules try to reduce the amount of PacketIn events and therefore restrict the abilities of an attacker to be successful. Occurring PacketIns are cached and served using rate limiting to further reduce the impact of an attack. As a side effect this approach causes unfavorable delays due to the caching of packets.

Finally, in [12] a mechanism is proposed to safely remove entries from full flow tables. The ratio of PacketIns and FlowMods is supervised and if the table is going to be full, rules are removed using a least-frequently-used scheme. As a disadvantage the approach causes potentially high load on the switches due to aggressive usage of OpenFlow's statistical features.

Our approach is more comprehensive than exiting works, as it covers all header fields. Additionally we show a novel analytic approach for the dimensioning of the system and the expected detection rates.

III. Denial of Service Attacks against the Data Plane

In this section we begin with the introduction of the class of DoS attacks against the data plane. This attack class was first described in 2013 by [13] and [14]. Afterwards, we present a novel statistical detection approach specifically tailored to the problem. Finally, we introduce a novel lightweight method to mitigate a detected attack with only small restrictions to the networks' functionality.

A. Attack Description

In this work we focus on a DoS attack class against the data plane that is based on a reactive SDN which is the widely accepted standard behavior of OpenFlow switches. As previously seen in Figure 1, upon the incoming of an unknown flow the switch typically consults the controller for further decisions by encapsulating the first packet of the flow into a PacketIn message. Then, the controller can inspect the packet, make a forwarding decision, and set new flow rules in the switch using FlowMod messages if necessary.

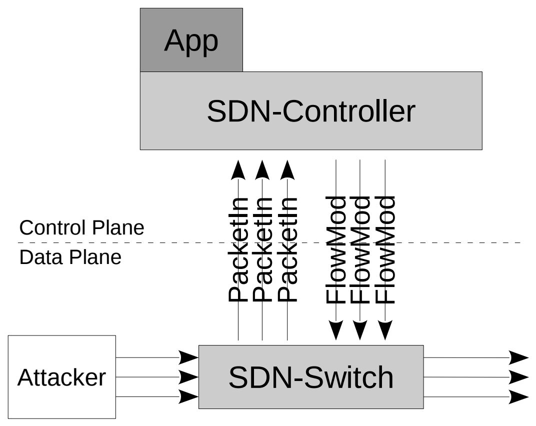

Fig. 2: DoS attack against a reactive SDN.

There are different ways for a controller to handle this decision making, especially regarding the granularity of flow definitions. For instance, flows could be setup using the quintuple Source IP, Destination IP, Source Port, Destination Port, Protocol or just Layer-2 addresses. An attacker knowing about the controller's decision making and flow rule setting behavior is able to craft packets that trigger those FlowMods. For the switch these packets appear as new flows and therefore, are handled by the controller resulting in an increasing number of flow rules in the switching tables as depicted in Figure 2. The concrete impact of a switch with full switching tables is not generally defined and highly depends on the model. Typical behavior includes the dropping of older switching entries or ignoring new rule setting requests.

Note that in general the attacker is not able to arbitrarily craft the packets since all attack packets need to be routed via the target switch. Cloud environments like for example OpenStack could also prohibit source IP spoofing with packet filters. A rule matching the exact characteristics of the attacking flow and applying a standard behavior like dropping the attacker's packets would be able to completely shut down the attack.

B. Detection

The general idea for the detection of DoS attacks against the data plane aims at localizing the fixed header fields of the attacking flow. These impose a regularity that is not observed in normal traffic since PacketIn events are just seen once upon flow establishment. The approach uses a table of counters with the different header fields as columns. The table is regularly, i.e. in fixed time intervals, inspected statistically and the maximum entry is abnormally large in case of an attack. To normalize the table size the header fields are hashed by a uniformly dispersing function with fixed output size. The digest of an input determines the row where to increment the counter. During an attack the entries which correspond to fixed fields of the attacking flow grow very fast and are used for the detection.

Table I shows a simplified example with three header fields a, b and c where the latter is varied. The columns represent the different header fields, while the rows are accessed using the hashed values of the particular header field. After a couple

a	b	c	a	b	c		a	b	c
0	0	0	0	0	0		0	0	1
0	0	1	0	0	1		0	0	1
0	0	0	0	0	0		0	0	1
0	0	0	0	0	1		0	0	1
1	0	0	2	0	0	…	**10**	0	1
0	0	0	0	0	0		0	0	1
0	1	0	0	2	0		0	**10**	1
0	0	0	0	0	0		0	0	1
0	0	0	0	0	0		0	0	1

TABLE I: Simple example showing the growth of a counter table over time for ten consecutive packets with three header fields **a**, **b**, **c** where **c** is varied while **a** and **b** remain fix.

of PacketIns the fixed fields of the attacking flow are clearly distinguishable from the varied fields.

For further explanations we formalize the necessary terms as follows. $\mathcal{H}$ is the set of all header fields considered by the detection algorithm, e.g., $\mathcal{H}$ ={src_mac, dst_mac, src_ip, dst_ip, proto, src_port, dst_port, ...}. A packet p is characterized by a set of tuples (h, v) of header fields $h \in \mathcal{H}$ where h acts as a key and v as value, e.g., $p =$ {(src_mac, 11:22:33:44:55:66), (dst_mac, 66:55:44:33:22:11), (src_ip, 1.2.3.4), ...}. The concrete value of a field h in the packet p is denoted by $v_{h,p}$. Using these terms, a hash function is defined as

$$hash : v_{h,p} \rightarrow \mathbb{N}_0^{<|hash|}$$

where $|hash|$ is the size of the hash function's image set. The table T is a matrix of the dimension $\mathbb{N}^{|hash|,|\mathcal{H}|}$.

We used the 32Bit FNV-1a hashing function (see [15]) folded to an output size of 16Bit by applying an XOR operation of the upper half to the lower half of the hash sum. The chosen hash function is designed to be fast while having a low collision rate which is evenly distributed itself. This results in 2^{16} rows in the counter table.

Algorithm 1 Book keeping of seen PacketIn messages.

```
Require: Counter table $T$
  while true do
    receive PacketIn and unwrap packet $p$
    for all $h \in \{h | h \in \mathcal{H} \wedge (h, v) \in p\}$ do
      $S \leftarrow hash(v_{h,p})$
      $(c, v_{h,p-1}) \leftarrow T_{(S,h)}$
      $T_{(S,h)} \leftarrow (c + 1, v_{h,p}))$
    end for
  end while
```

Every incoming PacketIn is unwrapped and the included packet is handled by the method shown in Algorithm 1. For each field its value is hashed. The hash sum is now used as an index in the table where the counter is incremented. Additionally, the corresponding fields of the most recent packets are stored with the counter for later usage by the mitigation routine upon a detected attack. The employed FNV-1a hash function is designed to have a low collision rate. Thus we can assume that storing of one field is enough in practice.

Since the application of the hash function can be bounded to the largest field size and the field updates run in $\mathcal{O}(1)$, the overall update time is in $\mathcal{O}(|\mathcal{H}|)$. The table may require quite a large amount of memory. Depending on the implementation the memory consumption can be bound to

$$\mathcal{O}(|hash| \cdot (|counter| + |field|) \cdot |\mathcal{H}|)$$

where $|counter|$ is the size of the counter (in Bytes) and $|headerfield|$ is the size of the structure holding the most recent header field value. If the table is statically allocated it would require exactly this amount of memory. Some implementations allow a dynamic allocation at runtime which optimizes memory consumption at the cost of slower data access and allocation overhead. We consider a statically sized table to be the preferable approach since the usage of hash functions distributes the counter updates evenly. Therefore, there should not be too many untouched fields with counters equaling zero.

Algorithm 2 Attack detection and mitigation.

```
Require: Counter table $T$, time interval $t_W$
  $P \leftarrow \{\}$
  for every $t_W$ seconds do
    for $h \in \mathcal{H}$ do
      for $S \in \mathbb{N}_0^{<|hash|}$ do
        $(c, v_{h,p}) \leftarrow T_{S,h}$
        if $c > \theta_m$ then
          $P \leftarrow P \cup \{(h, v_{h,p})\}$
        end if
      end for
    end for
    if $P \neq \emptyset$ then
      Block all packets that match headers in $P$
    end if
    reset $T$
  end for
```

As seen in Algorithm 2, the detection routine runs independently of the book keeping on a regular basis and statistically evaluates the counter values. The table is evaluated every fixed time interval t_W. The counter is evaluated for every header h and every hash value S, i.e. for every field in the table. If the value of a counter is higher than a predefined threshold θ_m, an attack is indicated and the corresponding field $v_{h,p}$ is appended to list P.

The detection algorithm has a complexity of $\mathcal{O}(|hash| \cdot |\mathcal{H}|)$. In conjunction with the data collection seen above, the overall detection is lightweight. Especially, since it can be executed concurrently and thus, no stalling of the packet pipeline is necessary.

C. Counter Measures

After detecting an attack the set P contains the fixed headers of the attacker. From this set it is easy to craft a flow rule that matches the fields from P while treating the others as wildcards. If one header or more headers is in the set with different values, multiple rules have to be installed. For this the existing rule with the singular headers from P has to be copied and for each value of one header a rule has to be created. This could for example be the case if the attacker leverages a bot net and as a result more than one IP-Source address has to be blocked. When installed as a low prioritized dropping rule in the switch it is now able to handle further attacking packets at line rate and without additional interaction with the controller. The impact on the network is negligible since only regular hosts falling in the detected flow characteristics are affected by the mitigation which is considered unlikely concerning our attack model.

IV. Evaluation

This section describes the abstracted simulation method which was used to validate the detection algorithm. Besides the simulation results, an analytic evaluation of the *false positive* and *false negative* probabilities is provided which can be used to determine the correct parameters of the detection algorithm.

A. Abstracted Simulation

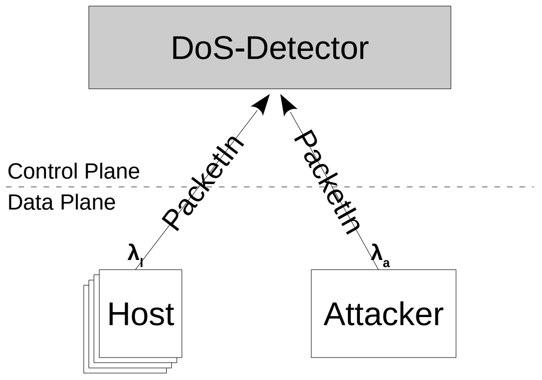

Fig. 3: Abstracted simulation of the detection system.

In order to evaluate the detection method we built a simulation based on the widely used OMNeT++ framework [16]. One feature of our simulation is that we did not simulate on a data plane level, but only on a control plane level. The abstracted view is shown in Figure 3: In the simulated system a host causes a new PacketIn when a new connection is started, i.e. with the first packet. Afterwards, all packets which belong to the same flow would be handled in hardware (i.e. the data plane) in the real system and are not simulated. This greatly reduces the number of simulated packets, while retaining all important effects of the attack. In our simulation we used for the legitimate users negative exponentially distributed arrivals with an expected mean inter arrival time T_l, which correspond to an arrival rate $\lambda_l = \frac{1}{T_l}$. The attacker arrival rate is called λ_a. Other important parameters are the window size t_W, i.e. the time between two consecutive runs of the detection algorithm and the detection threshold θ_m. The parameters are also summarized in Table II.

Normal Traffic Arrival Rate	λ_l
Attack Arrival Rate	λ_a
Window time	t_W
Number of Hosts	H
Max value threshold	θ_m

TABLE II: Parameters of simulation and Analysis

B. Analytic Evaluation of the Detection Performance

The detection mechanism of our approach labels the state of the system as "under attack" if the table maximum $m_T = max(T_{S,h}) \forall S, h$ exceeds some threshold θ_m. The threshold should be low enough to detect attacks, but it should not raise an alarm if no attack is attempted. As usual, we call these false alarms *false positives* while undetected attacks are *false negatives*. The threshold could be set empirically by just trying different thresholds and measuring the effects.

In this section, we try to give a more systematic approach for determining the threshold. The expected m_T corresponds with the maximum expected collisions of a header value. For example, if all headers of the incoming connections of one host are uniformly distributed, except for the source IP, this results in a high value in the corresponding table entry s, h and therefore this entry $T(s,h)$ dominates $m_s = max(T_s)$, the maximum of row s. For this system, we can compute the probability $P_{m_s}(n)$ of the maximum of row T_s with the help of the Erlang distribution: The Erlang CDF $F_{n,\lambda}(x)$ describes the probability of n events occurring in a certain time interval x with $0 \leq X \leq x$, if the events are exponentially distributed in time with a rate λ.

$$F_{n,\lambda}(x) = 1 - e^{-\lambda \cdot x} \sum_{i=0}^{n-1} \frac{(\lambda \cdot x)^i}{i!}$$

For our case the interval is always $0 \leq X \leq t_W$. For a fixed event rate, the probability of more than n events is:

$$P_{m_s}(n) = 1 - F_{n,\lambda}(t_W) = e^{-\lambda \cdot t_W} \sum_{i=0}^{n-1} \frac{(\lambda \cdot t_W)^i}{i!}$$

The probability of n events is then:

$$p_{m_s}(n) = P_{m_s}(n+1) - P_{m_s}(n)$$

$p_{m_i}(n)$ corresponds with the probability of a value of n for row m_s if the entry s, t of the counter table is hit by the repeated header field, which is the source IP in our case. Now, our algorithm determines maximum of the table m_T. The network consists of H hosts which generate flows statistically independent, therefore the cumulative probability of m_T is:

$$P_{m_T}(n) = (P_{m_s}(n))^H = (1 - F_{n,\lambda}(t_W))^H$$

and the corresponding probability density is:

$$p_{m_T}(n) = P_{m_T}(n+1) - P_{m_T)}(n)$$

From this density the expected maximum can be derived with:

$$E(m_T) = \sum_{n=0}^{\infty} n \cdot p_{m_T}(n)$$

The results shown in Figure 4 support this theoretic model, the simulation fits the theoretical results very well, i.e. the analytical expected value matches the simulated mean.

With the help of this probability we can also get the *false positive* (FP) probability for a given threshold θ_m, as this is the probability to reach a value of more than θ_m:

$$P_{FP} = 1 - P_{m_T}(\theta_m) = 1 - (1 - F_{\theta_m,\lambda_l}(t_W))^H$$

On the other hand, an attacker with the rate λ_a is not detected with the probability indicated by the *false negative* (FN) rate:

$$P_{FN} = P_{m_T}(\theta_m) = 1 - F_{\theta_m,\lambda_a}(t_W)$$

As we assume only one attacker $H = 1$ for P_{FN}.

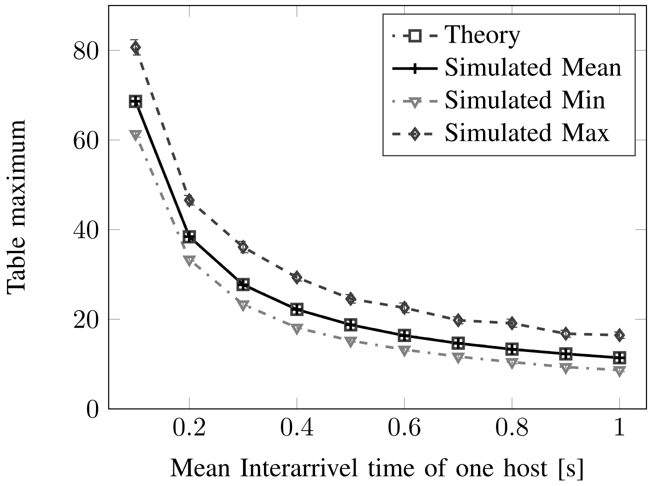

Fig. 4: Results of the maximum table value for different inter arrival times.

C. Results

We evaluated our system with a network of legitimate $H = 100$ hosts. We simulated 10 repetitions for each setting with a duration of 1000 s per run.

	Normal Traffic	**10% Rate Attack**
Arrival Rate	$\lambda_l = 1\,s^{-1}$	$\lambda_a = 10\,s^{-1}$
θ_m	20.00	20.00
H	100	100
	Simulation	
Mean max	11.40	52.58
Maximum max	18.00	84.00
	Theory	
Expected max	11.41	50.00
FP Propability	0.035%	0.035%
FN Propability	-	0.000048 %

TABLE III: Exemplary simulation results for a network with 100 Hosts

Table III shows a comparison of the behavior without and with attack, in this simulation the maximum did not exceed a value of 18 so we did not have any false positives for the given threshold. As can be seen from the results, if an attack adds only 10% additional load to the system, it can be detected easily. Even with this relatively small additional load, the expected maximum value is about five times higher than the normal traffic. This large difference results in very small FN and FP probabilities. A 10% increase means that the switch's table is filled 10% faster than usual, or if we take timeouts into account the table has 10% more entries. Usually, this comparably small increase should not affect the system's behavior.

Figure 4 shows the behavior of the max value for different traffic intensities. The simulation results match almost exactly the theoretic forecast. It can be observed that the table maximum is highly dependent on the traffic rate of one host. Therefore, it can be necessary to set the detection threshold according to the specific network environment.

Figure 5 shows the sum of the false negative and false positive probabilities. When the sum is close to 0 (darker) the algorithm is performing well. The lighter upper right region

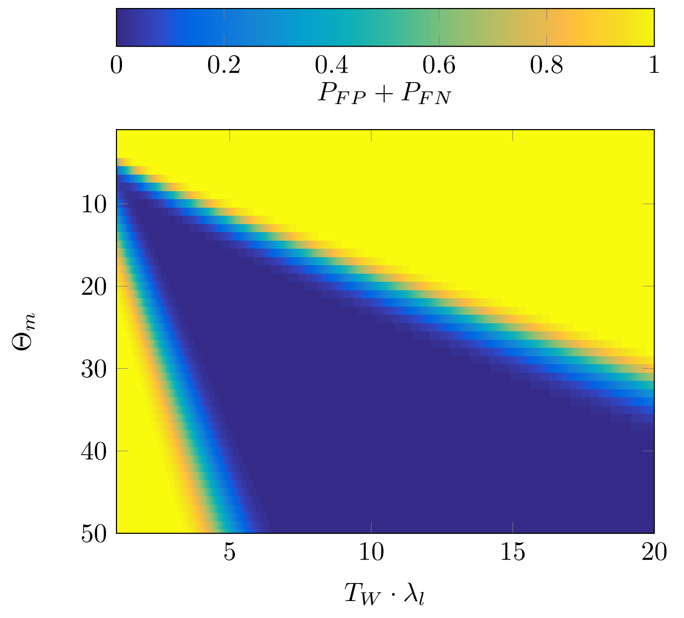

Fig. 5: Working region of detection algorithm, with $\lambda_a = 10 \cdot \lambda_l$ and $H = 100$

is caused by a high false positive probability the lower left by false negatives. For a given arrival rate a small detection window is beneficial as this improves the detection speed. Although if the product of window time and rate of legitimate hosts is big we have a very broad detection range, i.e. we have a big region where we can choose a good threshold while sacrificing the detection speed. On the other hand for a small product and consequently a small threshold the system is very sensitive for changes in the arrival rate of the users as this can cause false positives.

V. Conclusion

In this work we introduced a novel detection and mitigation algorithm for Denial of Service Attacks in Software Defined Networks. Our work concentrates on attacks which aim to overflow the hardware tables of SDN switches in cloud environments. The attacker causes a high number of PacketIn messages by changing header fields. Our proposed detection approach is based on the observation that an attacker cannot change all header fields. This allows us to identify the attack. For example, in order to keep the connectivity with the network for OpenStack spoofing the source IP address is not possible. Our approach uses a table with the header fields as columns and hashes of the header fields as rows. If an attack occurs the fields corresponding to the unchanged fields grow tremendously. After identifying the attack, we propose to use an OpenFlow rule which drops further attack packets.

Our evaluation shows that the algorithm can detect attacks reliably and with low false positive probability with the correct parameters. Using the proposed formulas it is analytically possible to determine a good choice of these parameters.

One drawback is, that our approach cannot detect attackers, that can change all header fields simultaneously. This could be the case for an attacker that controls a big bot-net.

In the future we want to enhance the detection with additional metrics such as the difference between two subsequent maxima. Finally, we want to further extend the mitigation of the attack. The mitigation could also lead to a relatively big set of table entries and deteriorate performance, though fewer rules than without our approach are used. Especially, possible side effects on the legitimate network traffic should be investigated.

Acknowledgment

This work has been performed in the framework of the KMU-Innovativ project SarDiNe, and it is partly funded by the BMBF (Project ID 16BP12308). The authors alone are responsible for the content of the paper.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks, (LCN)*, 2010.

[3] J. M. Dover, "A denial of service attack against the Open Floodlight SDN controller," Dover Networks, Tech. Rep., 2013.

[4] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *2015 International Conference on Computing, Networking and Communications (ICNC)*, 2015.

[5] H. Wang, L. Xu, and G. Gu, "FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[6] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[7] R. Bifulco and A. Matsiuk, "Towards scalable sdn switches: Enabling faster flow table entries installation," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[8] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch." in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[9] L. Wei and C. J. Fung, "FlowRanger: A request prioritizing algorithm for controller DoS attacks in Software Defined Networks," in *IEEE International Conference on Communications (ICC)*, 2015.

[10] R. Kandoi and M. Antikainen, "Denial-of-service attacks in OpenFlow SDN networks," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015.

[11] H. Wang, L. Xu, and G. Gu, "OF-Guard: A DoS Attack Prevention Extension in Software-Defined Networks," Texas A&M University, Tech. Rep., 2014.

[12] Y. Qian, W. You, and K. Qian, "Openflow flow table overflow attacks and countermeasures," in *2016 European Conference on Networks and Communications (EuCNC)*, 2016.

[13] D. Kreutz, F. M. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*, 2013.

[14] R. Klöti, V. Kotronis, and P. Smith, *Proceedings - International Conference on Network Protocols (ICNP)*, 2013.

[15] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen, "The FNV Non-Cryptographic Hash Algorithm," https://tools.ietf.org/html/draft-eastlake-fnv-11.

[16] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proceedings of the European simulation multiconference (ESM)*, 2001.