Network Anti-Spoofing with SDN Data plane

Yehuda Afek
Tel-Aviv University
Email: afek@post.tau.ac.il

Anat Bremler-Barr
The Interdisciplinary Center Herzliya
Email: bremler@idc.ac.il

Lior Shafir
Tel-Aviv University
Email: lior.shafir@gmail.com

***Abstract*—Traditional DDoS anti-spoofing scrubbers require dedicated middleboxes thus adding CAPEX, latency and complexity in the network. This paper starts by showing that the current SDN match-and-action model is rich enough to implement a collection of anti-spoofing methods. Secondly we develop and utilize advance methods for dynamic resource sharing to distribute the required mitigation resources over a network of switches.**

None of the earlier attempts to implement anti-spoofing in SDN actually directly exploited the match and action power of the switch data plane. They required additional functionalities on top of the match-and-action model, and are not implementable on an SDN switch as is. Our method builds on the premise that an SDN data path is a very fast and efficient engine to perform low level primitive operations at wire speed. The solution requires a number of flow-table rules and switch-controller messages proportional to the *legitimate* traffic. To scale when protecting multiple large servers the flow tables of multiple switches are harnessed in a distributed and dynamic network based solution.

We have fully implemented all our methods in either OpenFlow1.5 in Open-vSwitch and in P4. The system mitigates spoofed attacks on either the SDN infrastructure itself or on downstream servers.

I. Introduction

The new networking technology of Software Defined Networking (SDN) presents new opportunities to provide a flexible and powerful defense system against DDoS attacks. Recently a new phase in the development of SDN technologies has emerged, where the software running over the SDN infrastructure becomes richer, more flexible, and capable of expressing more powerful network operations. New SDN high level languages such as P4, and the corresponding appearance of SDN programmable switch like Intel FlexPipe [5] and RMT [15] enable SDN switches to carry out advanced packet modifications, including a richer set of key action primitives necessary to manipulate packet headers.

In this paper we take the first step in the design of an SDN network based system for the mitigation of denial of service attacks. Specifically, we implement anti-spoofing techniques in the SDN data-plane. The new approach uses the SDN switches and controllers as the platform thus avoiding any CAPEX expenditure, and provides flexible software like solution. We believe this approach will be extended in the future to implement other functionalities of middleboxes as software on the SDN platform.

While DDoS attacks remain a top threat that is growing in size and frequency of reported incidents [2], [4], [9], with the spoofed SYN attacks remaining a major vector, SDN opens the door for yet new vulnerabilities to these DDoS attacks [10], [13], [25]. In a control plane saturation attack, for example, in a reactive mode SDN system, for each new connection (SYN packet) a message is sent from the switch to the controller and back (to install a rule for the corresponding flow) as noticed by [26], [30]. In Open vSwitch [7] for example, our experiments show that the performance degrades dramatically under spoofed SYN flood attack not only in reactive mode as shown in [30] but also in the proactive mode due to the data-path caching mechanism (each new SYN results in switching to user mode). We note that recent state-full functionalities and implementations in SDN such as the P4 cross-flow registers [11], OpenState.p4 and connection tracking (conntrack) [10] in Open vSwitch, all store a flow-state in the data-path and thus are again vulnerable to Syn-Attack. Furthermore, spoofed SYN floods (a very old DDoS attack vector) remain a substantial part of current DDoS attacks due to their simplicity and effectiveness. At the same time SDN data-path (in the switches) presents a powerful high throughput packet processing and filtering capabilities which can be instrumental in the scrubbing of DDoS attacks. In this paper we harness these capabilities to design a powerful and flexible and dynamic SDN based network anti-DDoS system. Our system provides protection from spoofed attacks for both the SDN infrastructure (saturation and cache miss attacks) and for down stream servers.

At a high level we distinguish between two classes of DDoS attack vectors, the volumetric network based attacks and the application level attacks. According to this partition we believe the protection from DDoS attacks should be divided between upstream scrubbers in the network that deal with the volumetric network based attacks, and closer to the edge scrubbers that deal with the more sophisticated application level attacks that have lower throughput. In this paper we address the spoofed attacks which are part of the volumetric network based attacks, which are still very popular attacks [2], [4].

The main approach taken to stop spoofed Syn floods and other spoofed attacks is that of a state-less challenge response. An ACK that includes a hash generated cookie is sent in response to each SYN packet, and only if the client responds with the correct cookie back, it is authenticated (it might still be malicious, but authentic) [3], [23], [28], [29]. Other approaches that use TTL correlation will be integrated in future work.

In this paper we present a two stage solution for the state-less challenge response in SDN networks. First (in Section IV) we show how a single SDN switch is programmed to be an efficient spoofed SYN flood mitigator. We have

978-1-5090-5336-0/17/$31.00 ©2017 IEEE

implemented four anti-Syn-spoofing methods [3], [23], [28], [29] and one anti-DNS spoofing method [23], as outlined in Section III. Our implementations exploit the match and action SDN model using only supported and existing P4 features and using OpenFlow 1.5 with its actions applied to additional fields in vein of an already existing actions. This is in contrast to previous work, AVANT-GUARD [26], that reverted to extensive unsupported changes to the switch in order to implement one of the methods from our anti-spoofing techniques in SDN. A complete realistic prototype of this step is implemented here using OpenFlow1.5 in Open-vSwitch and P4 behavioral model in Mininet. In P4, our solution can be directly implemented as is on any P4 target. In OpenFlow 1.5 based switches one can easily tell whether our implementation can be implemented or not (e.g., it can run on Intel FlexPipes [5], OpenFlow 1.5 switches with match buffer that covers all the TCP header such as Noviflow [6]). In line with the SYN-cookie technique the space complexity (number of flow table entries) of our algorithms is only a function of the number of authenticated connections, independent of the spoofed attack size. The prototype is presented and evaluated in Sections VI and VII.

In the second stage (Section V) the network distributes the space complexity of the solution over several switches. The system adapts dynamically to the attack size and resources consumption supporting the addition of spare switches if necessary. We believe this is a first step in a more general SDN methodology of using the SDN infrastructure to provide by software only (Switch rules type of software) additional network functions on demand, such as full DDoS scrubber (see Section VIII), firewalls, and load balancers.

The contributions of this paper are:

1. Programming and implementing a collection of anti-spoofing techniques completely in OpenFlow1.5 and P4 match and action rules, without controller involvement during the authentication phase.

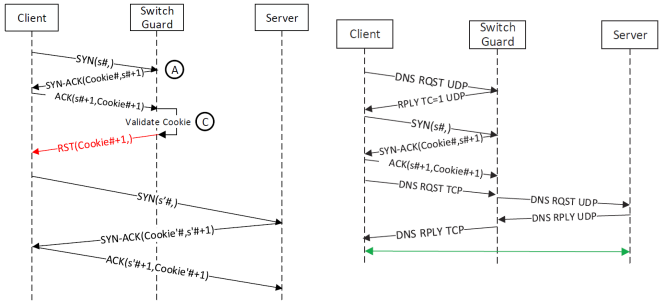
2. Verified the implementation in a setup prototype in (i) Open vSwitch using OpenFlow 1.5 (ii) P4 using a behavioral model. In both the setup includes a web-site with legitimate users and attackers.

3. Programming experience comparison: OpenFlow and P4. We use our case to compare the effort and design complexity required to program novel applications that operate with new header fields in both OpenFlow 1.5 using OVS, and P4.

4. Developed a set of algorithms and methods to dynamically and adaptively distribute the required rule space (which is a function of the number of legitimate connections) over a network of switches, when a single switch is not enough. We present algorithms that automatically redistribute the required rules over the switches in a network while minimizing the controller - switch traffic.

II. Related Work

Usually there are two aspects of security discussed in the context of SDN: First is the use of SDN to facilitate and improve security services and solutions [16], second are the security vulnerabilities that have been introduced by the SDN architecture as noticed by [18], [25], [26]. Notice that our proposed mitigation techniques use the first to solve the second as well as protecting other targets of attack.



(a) TCP Reset (b) DNS spoofed attack mitigation

Fig. 1: SYN and DNS spoofed attack mitigation methods. A complete description of other methods that we use (HTTP, Safe-Reset and Proxy) appears in the technical report [12]

Wang et. al. [30] analyze and study the behavior of SDN network under a spoofed SYN flood attack. They specifically show and address the switch - controller communication problem under a SYN flood attack, more specifically, the open flow agent on the switch is swamped and becomes a bottleneck. Their proposed solution is not to mitigate the attack but rather to simply increase the SDN capacity by scaling up the control plane with a pool of vSwitches distributed across the network.

AVANT-GUARD [26] describes a *control plane saturation attack* (as in [30]). They continue and provide one of the methods in the first stage of our solution (TCP-proxy anti-spoofing mechanism), however their proxy anti-spoofing mechanism requires changes in the data plane; specifically adding data-structures, packet parser, arbiter and packet editor modifications. In [21] Mekky et. al. also perform TCP handshake in the data-plane, however DDoS was not their concern and their implementation is not robust against a spoofed SYN flood attack. Moreover, their implementation of the TCP handshake relies on additional logic (extended architecture) and connections state table.

Some work [17], [20] has been done to exploit the controllability of an SDN network in the context of diverting DDoS attack traffic upon anomaly detection: An enhancement to the Remote Triggered Black-Hole (RTBH) routing approach is described in [20]. Bohatei [17] proposes an elastic scaling of defense for several DDoS attacks by using NFV.

III. Challenge Response Methods

In this paper, four different SYN anti-spoofing methods (TCP Proxy, HTTP redirect, TCP safe reset, TCP Reset [3], [28], [29]) and one DNS anti-spoofing method [23] are implemented over SDN using OpenFlow 1.5 and P4. Due to space limitations we only describe here the HTTP redirect, and TCP Reset. The implementation of the other methods follows

the same principles and the details are in the technical report [12].

A. Spoofed SYN Flooding Mitigation

Floods of spoofed SYN packets are an effective DDoS attack on servers and on the SDN infrastructure. On the server it quickly exhausts the TCP (Transmission Control Blocks) and in an SDN switch it exhausts the switch flow-table and the communication channel between the switch and the controller.

The most effective mitigation methods against spoofed SYN flood attacks are different variations of the SYN Cookie method [3], which is basically a challenge response technique that does not maintain any state on the server (mitigator) side. Instead, see Fig. 1a, the state is encoded into the sequence number in the response SYN-ACK. A legitimate client would respond with an ACK with the correct corresponding ACK number, and the mitigator would allow it to proceed. We explain here two variations:

HTTP Redirect: In [29], after being authenticated the mitigating device installs a pinhole by recording the source IP of the connection as legitimate, then responds to the http get request with a corresponding http-redirect response, and finally closes the authenticated connection. Sending a redirect response with the same original server address causes the client to re-establish the connection. This time it will go directly to the server through the installed pinhole.

TCP Reset: This method is similar to the HTTP-Redirect method except that a RST packet is sent to the client instead of an HTTP-Redirect response upon handshake completion, see Fig. 1a. This technique is suitable not only for HTTP protocols but also SMTP and others.

B. Spoofed DNS Flooding Mitigation

Following [23] the techniques of the last subsection can be applied to mitigate a spoofed DNS request flood in SDN. Although the majority of DNS traffic is carried over UDP, DNS may also be carried over TCP. When the switch receives a DNS UDP request from a source it has not yet authenticated, it forces the client to repeat its request over TCP, and then performs a TCP authentication using a cookie as described in §III-A. In order to force the client to repeat its UDP request in TCP the *TC* bit (Truncated - Indicates that the UDP response is too long to be carried by UDP) in the DNS header is set in the response, see Fig. 1b. After the TCP handshake is completed, it first installs the corresponding rules (pinhole) to allow future UDP based requests from this source to pass through, and second it forwards the TCP DNS request to the controller that acts as a TCP/UDP proxy between the client and server only for this request.

IV. Single Switch Solution

Our approach is to break the Syn-cookie methods into primitive steps and implement each primitive step as an action in the SDN data plane. In this way, the authentication phase is completed without any switch-controller communication. Essentially the received SYN packet is turned around and sent back to the sender as a SYN-ACK packet with a cookie (Phase A in Fig. 1). To achieve this, the following primitive steps are performed on received and not yet authenticated SYN packets (see Figure 2):

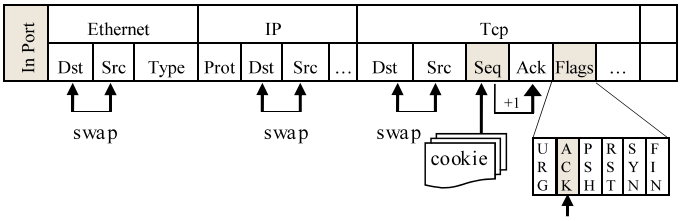


Fig. 2: Modifying a SYN packet to become a SYN-ACK packet

1) Exchange the source and destination IP addresses.
2) Exchange the source and destination Ethernet addresses.
3) Exchange the source and destination TCP ports.
4) Set the ACK bit in the TCP flags, to turn the SYN packet into a SYN-ACK packet.
5) Increment the client-SEQ number field by one.
6) Copy the incremented SEQ number field to the ACK number field.
7) Write a pre-generated cookie to the SEQ number field. (See §IV-A).
8) Recalculate the IP/TCP checksum values (in P4).
9) Send the packet back on the incoming port on which the SYN packet was received.

The *http redirect*, and *TCP-reset* anti-spoofing methods each requires all the nine primitive steps above.

All the above 9 steps are in the spirit of the SDN match-and-action model. In the P4 [14] SDN abstraction, these steps are supported as is, not requiring any modifications. OpenFlow 1.5 SDN standard also supports these action types, however few of the actions are restricted to particular fields, hence some of the methods require to extend them to additional fields (i.e., SEQ, ACK fields).

A. Pre-generated cookies

Here we explain how the SYN cookies are generated without communication with the controller during the actual authentication. A key requirement from the cookie generation is that the attacker would not be able to predict or learn the sequence number values, i.e., the cookie should appear to be random, and on the other hand should be a function of some of the SYN packet socket parameters so the SYN cookie method is stateless.

To this end, we partition the space spanned by the source IP address, and TCP source port into 256 partitions and associate a different 32 bit random number with each partition.

The partition is on arbitrary selection of 8 bits out of the 48 bits that represent the source IP and the TCP source port, see Table I. To prevent easy learning of the cookies by the attackers, the partition is based on the *low-order* bits of the above 48 bits. The controller periodically updates the 512 rules in the switch table (256 for SYN and 256 for the corresponding ACK) to use a set of different random numbers. In addition,

	Match					Action		Time-out	Num. Rules
	Prot.	flags	src-ip	src-port	ack	↔ - swap fields			
1	TCP	All	$\in Pinholes$	*		Forward		t	$Pinholes$
2	TCP	SYN	**.**.*0.**	**0*		seq← $Seq\#_0$	src-ip ↔ dst-ip, src-port ↔ dst-port src-eth ↔ dst-eth, tcp-flags ← $0x18$ ack ← seq + 1 , send to in_port		256
			...	...		...			
			..*0.**	**F*		seq← $Seq\#_{15}$			
			..*1.**	**0*		seq← $Seq\#_{16}$			
			...	...		...			
			..*F.**	**F*		seq← $Seq\#_{255}$			
3	TCP	ACK	**.**.*0.**	**0*	$Seq\#_0$+1	send to controller **or** (in Open-vSwitch, *TCP-Reset* method) use *learn-action* to locally install a pinhole and turn ACK into RST (See technical report [12].)			256
			...	...	...				
			..*F.**	**F*	$Seq\#_{255}$+1				
4	TCP	All	All	*		Drop			1

TABLE I: The rules installed on a switch to implement a redirect-based (*HTTP* and *TCP-Reset*) anti-spoofing methods.

to prevent the attacker from easily learning the bits on which we partition, the controller periodically changes the 8 bits which span the partition. The controller monitors the amount and frequency of successful handshakes per each partition (the match counters associated with the ACK rules), and may decide to perform the cookies updates at a different frequency or further refine a busy partition.

If the 3rd ACK packet is matched with the correct cookie, the connection is authenticated and a corresponding pinhole is installed. In the *TCP-Reset* technique (one of the four methods), the ACK is forwarded to the controller that sends a RST packet back to the client. (or optionally, the ACK packet is transformed into a RST packet and a new pinhole is installed locally on the switch using the *learn* action). To transform an ACK packet into a RST packet a set of primitive steps similar to those used to transform a SYN into a SYN-ACK are performed[1] This allows us to implement the *TCP-Reset* technique without any controller intervention, by using the Open vSwitch `learn` action, as we demonstrate in the example in the technical report [12]. If the 3rd ACK (or RST) packet does not match any rule, i.e., has an incorrect cookie, it is dropped. This happens when there is an ACK attack concurrently with the SYN attack.

The methods described above install pinholes for authenticated sources. The numbers of installed rules on a switch might grow proportionally to the legitimate traffic. Each pinhole rule is installed with a timeout value (both hard and idle timeouts) to avoid the number of rules from growing without bound. Table I summarizes the total number of rules used by our mitigation solution. While recent SDN switches have large TCAM size and may contain millions of flow entries [6], [8], (e.g., Noviswitch supports up to 1 million entries in wild card match flow tables [6]) there are still many hardware and software based switches with only tens of thousands of rules space.

V. Distributed Network Solution

The single switch solution works well as long as the switch resources, e.g., flow table size, and processing power, are not exhausted under an attack. The flow table may be exhausted by the amount of pinholes created for the authenticated sources. The processing power may become a bottleneck due to the extra processing of the SYN and ACK packets (mostly in software switches like Open vSwitch, in a hardware based switch we expect the extra processing overhead to be negligible). Here we orchestrate multiple switches to adaptively and dynamically distribute the attack mitigation among them to constitute a coordinated mitigation system against multiple large spoofed attacks. The distribution allows to harness both the flow table entries and the processing power of the additional switches in the mitigation of an attack.

[1]the ACK number should be zero, the SEQ number should be the same as the incoming ACK number and only the RST flag should be set (ACK flag should be unset).

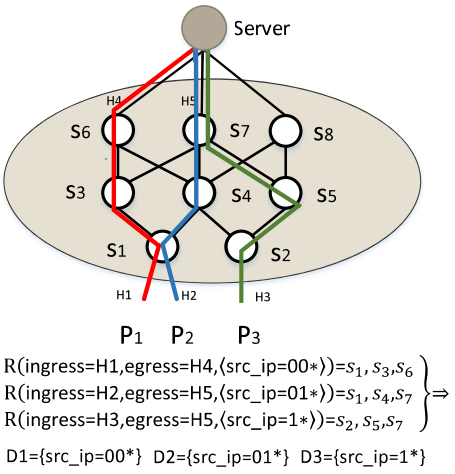


Fig. 3: An example decomposition of the network into paths. Flow space partition is computed from routing policy R.

As a first step the resources of the switches paths along which the incoming traffic flows are harnessed to distribute the mitigation load among them. This is the *vertical distribution* described in detail in the next few subsections.

The routing policy associates each path P_i with a sub flow space D_i. The routing policy R is a function $R(ingress, dst, header fields) = P_i = \{s_{i_1}, s_{i_2}, ..., s_{i_k}\}$ where $s_{i_1}, s_{i_2}, ..., s_{i_k}$ are the switches along which the corresponding packets are forwarded. R induces a set of τ paths $P_1, P_2, ..., P_\tau$, see an example in Fig. 3. We assume that any two sub-flow spaces D_i and D_j are disjoint $\forall i, j\ i \neq j$.

Flow Table 1				
	Match			Action
	Prot.	Flags	SIP	
1	TCP	All	$src\_ip \in D_{i_\ell}$	Redirect to Flow Table 2 for authentiaction
2	TCP	All		Forward

Flow Table 2 - (Similar to Table I in §IV - Single switch solution)

Fig. 4: The rules types with decreasing priority in a single switch associated with sub flow space D_{i_ℓ}.

A. Vertical Distribution

Here each sub space D_i associated with path P_i is further partitioned into smaller disjoint portions, $\{D_{i_1}, D_{i_2}, \ldots, D_{i_k}\}$, each portion associated with a switch along path P_i. Each switch s_{i_ℓ} authenticates flows belonging to D_{i_ℓ} and acts as a relay for all other flows that do not belong to D_{i_ℓ}.

The *vertical distribution* dynamically modifies the assignment of portions along a path to adapt to the changes in mitigation resources usage in the different switches. The rules used for vertical distribution on switch s_{i_ℓ} are given in Figure 4 in decreasing order of priority. The flow space partition is done on the *src_ip* field.

As the assignment of portions dynamically changes, we minimize the number of rules being updated (and thus also the number of controller messages) during such changes. To this end, we use a pipeline of two tables. The first table in Figure 4 redirects flows that are in D_{i_ℓ} to be authenticated in the next flow table, and forwards (acts as relay) all other packets. The second flow table in Figure 4 is the mitigation table that contains the same rules as in Table I in §IV. This approach (1) minimizes flow entries changes during reassignment of portions because only the first table is modified. (2) minimizes the number of rules that match D_{i_ℓ} allowing our solution to separate between the mitigation logic and the network redistribution logic.

1) Flow Space Partition and Allocation: The initial division algorithm differs from the reassignment algorithm during an update since the traffic and pinholes distribution are already known during an update, while initially the algorithm assumes uniform distribution over the portions.

In both cases the input to the algorithm is the available rules capacity at each switch along the path. For switches shared by only one path, the rules capacity is simply the number of available rules $C_{s,path} = C_s$, but for switches that share more than one path, the initial allocation for each path is proportional to the paths associated sub spaces. Formally, given a switch s and a set of m paths $P_1, P_2, ..., P_m$ passing through the switch, assuming the mitigation rules capacity in s is C_s, the allocated rules per each path is $C_{s_{path}} = C_s \times (|D_{path}| / \sum_{j=1}^{m} |D_j|)$. Note that the allocation above is used to estimate the portions size that are allocated to the switch for each one of the paths, however, during the attack mitigation, the rules capacity of the switch is shared by all paths to install pinholes with no per path bound.

Given k switches along path P_i and given the space for rules allocated on each switch along the path, $c_{i_1}, .., c_{i_k}$,

Division of P_3 sub-space $D_3 = 1*$ into portions using 5 bits accuracy					
Switch	Total # of Rules	Paths Shared	Alloc. for P_3	Weight	Portions
S_2	1000	P_3	1000	0.23	1000* 10010* 100110*
S_5	2000	P_3	2000	0.46	100111* 101* 1100* 11010*
S_5	2000	P_2, P_3	1334	0.31	11011* 111*

Fig. 5: Initial division of P_3 (from Fig. 3) into portions according to the switches rules capacity allocated for this path. The *wildcard* rules generation is done using the load balancing algorithm of [31]

our goal is to divide the sub space D_i into k portions and generate *wildcard* rules that match these portions: $D_{i_1}, .., D_{i_k}$. The SDN load balancing method presented in [31] provides a generic and accurate traffic-splitting scheme to compute a minimal set of *wildcard* rules given a set of weights as an input, rather than using ECMP or WCMP [19], [27] which partition the space assuming equal traffic load. We implemented and used the algorithm of [31] to divide the path subspace into smaller portions. In our case D_i is the input flow space, and $\{w_{i_1}, .., w_{i_k}\}$ is the input weights vector ($w_{i_j} = c_{i_j} / \sum_{\ell=1}^{k} c_{i_\ell}$).

An example of such sub space division into portions appears in Figure 5.

As the attack progresses, some portions may contain more pinholes than others due to the actual legitimate traffic distribution. A switch associated with an overloaded portion may become saturated. To avoid switch overload by the number of rules or processing capacity, we define configurable thresholds (e.g., 80%) for each of these two workload parameters (rules capacity, processing power). When the controller detects that one of these parameters exceeds its threshold on a particular switch, it changes the portions division by moving pieces of portions from the busy switch to lightly loaded switches along the path. Here we describe how to shift load from one switch to another along the path.

During a vertical update, an algorithm different than the initial distribution algorithm is used to take into account the pinholes distribution within the path that is associated with the saturated switch. The algorithm checks which path caused the heavy load on the busy switch, and redistribute all the portions associated with that path over the switches along the path. Our goal is to redistribute the portions to achieve fair allocation of existing pinholes along the path. Let P_i be the problematic path. To compute the available rules on each switch along path P_i, the algorithm updates $c_{i_1}, .., c_{i_k}$ on switches in P_i that share more than one path by calculating: $C_{i_j} = C_{avail} \times (PH_i / \sum_{\ell=1}^{m} PH_\ell)$ where PH_ℓ denotes the number of pinholes installed for path ℓ on switch j, and m is the number of paths that share switch j. The algorithm then computes how many pinholes to allocate for each switch along

the path to failry share the load among them.

Here the problem is more complicated than the initial division since the pinholes distribution over D_{i_j} is known and taken into account. Moreover, during such incremental update, we would like to minimize the number of controller messages, i.e., the number of pinholes that are being transferred between the switches. Given $c_{i_1}, .., c_{i_k}$ along the path, we use the following algorithm to redistribute the path portions with respect to the pinholes distribution.

First, for each switch $s_{i_\ell} \in P_i$ the algorithm calculates $\alpha_\ell = c_{i_\ell} / \sum_{m=1}^{k} c_{i_m}$ as the fraction of pinholes that will be assigned to s_{i_ℓ} out of the path pinholes that are redistributed. Then, we calculate for each switch how many pinholes it should release/receive to reach its fair allocation. For each switch that has to release pinholes, the pinholes in its portions are sorted (by source IP address) and partitioned into two sets of pieces such that one set contains the amount of pinholes that it has to keep, and the other set contains the amount of pinholes that it has to release.

In the partition above, there may be a large gap between the largest source IP pinhole of one piece, and the smallest IP pinhole of the next piece. The border line between two consecutive pieces can be placed anywhere in the gap. Our algorithm places the borderline in a way that minimizes the number of *wildcard* rules necessary to specify the pieces matching rules.

B. Vertical Dynamic Updates

The pinholes migration described above is an update problem of states. Usually the update should be consistent. A *consistent update*[24] is a mechanism for SDN that ensures key invariants hold during a transition from policy A to policy B. More specifically, in our case, a consistent update from policy A where a portion D_{i_j} is associated with switch S_a to policy B where the same portion D_{i_j} is associated with switch S_b ensures that every packet traversing the network is authenticated (and forwarded) exclusively by S_a or exclusively by S_b. For that there are known algorithm in the literature: *two phase commit* [24] or *TimeFlip* [22]. However, they come with extra cost of version numbers and tagging. In our case we can do a simpler solution, because a flow can go through anti-spoofing switch more than once, but eventually an authenticated connection is established with the server. Due to space limitations we describe our simpler migration algorithm in the technical report [12].

VI. Implementation

We implemented and tested all the variations of our solution in several environments: (1) OpenFlow using Open vSwitch 2.3.1 and POX controller, and (2) P4 using a behavioral model, simulator and a Mininet plugin.

A detailed running example of one of the anti-spoofing methods, the TCP-Reset appears in the technical report [12].

A. Open vSwitch Modifications

We chose to implement our solution in Open vSwitch 2.3.1 that contains most of the latest OpenFlow 1.5 extensions.

Design Considerations: In Open vSwitch, the packet may be directed through user-space daemon main component or through *datapath-kernel module*. When a first packet of a flow has a kernel cache miss, it is directed to the user-space component using an upcall, the user-space forwarding actions for this packet are being cached in the kernel for subsequent packets of the same flow. We implemented the new actions to allow efficient execution in the kernel-data-path. However, we saw in our experiments that a random spoofed ip-address attack is not cached efficiently in the kernel (we will show this in §VII). Additional implementation details about Open vSwitch and POX appear in the technical report [12].

B. P4 Programming Language

P4 (Programming Protocol-Independent Packet Processors) [14] is a new programming language that appears to be the next step in the evolution of OpenFlow. One of the main goals of P4 is to propose a protocol independent abstraction, that is, allowing the developer to program network applications without being tied to a specific protocol and a fixed header format. The P4 abstraction level fits exactly to our solution requirements. Several key principles in P4 provide a natural API for our solution, easing its implementation, thus generally demonstrating how more sophisticated data plane customizations can be adopted quickly using this approach. Here we briefly describe which key aspects of P4 are required by our task.

Configurable Packet Parser. P4 does not assume a fixed parser. The developer can define any custom subset of header formats. It is often the case that an SDN switch can inspect and write to these fields but this capability is not exposed due to limitations of existing protocols. P4 however allows us to easily interact with any field within the TCP header without being restricted by standardization issues. Specifically, we can easily define TCP flags and SEQ/ACK matching and writing rules using P4.

General Packet Processing Primitives. In P4, the processing instructions are not a fixed set of actions. The developer can construct generic actions that can be built out of simple operations such as add, modify, remove, increment, etc. For example, the following are primitive actions from the latest P4 specification [11]:

```
add_to_field  - Add a value to a field.
copy_header   - Copy one header instance
                to  another.
modify_field  - Set the value of a field.
```

These examples reflect the protocol-independence property. Unlike fixed protocol-dependent actions such as `PUSH_MPLS`, `POP_VLAN`, and `SET_NW_TTL` in existing protocols, the proposed P4 primitive actions are more natural and suitable for applications such as our solution that manipulate the packet to swap between header fields, increment the ACK field, and modify other fields in the TCP header as described in §IV.

P4 Implementation description, intended for readers that are familiar with P4 appears in the technical report [12].

C. OpenFlow and P4 - Our experience

The main difference we observed between OpenFlow and P4 is related to protocol-dependent issues. When starting to examine the feasibility of our solution in OpenFlow 1.5, we had to start bottom-up. Since protocol standardization and details are strongly tied with OpenFlow's level of abstraction, we had to focus in fields and bits specifications to find out which is supported in which version. Indeed, adding match/modify of only few fields inside the Open-vSwitch code required plumbing of code into dozens of different files, although the new actions were exactly the same as other actions that modify similar but different fields in the same header. Implementing the same behavior in P4 enabled us to focus on our main task of defining our flow, in a more natural way, mapping our techniques into the relevant logical tables and actions. The resulted P4 program verbosity did reflect the actual task we were trying to perform, thereby, was at the appropriate abstraction level (and expressiveness level). P4 supported a natural top-down networking design.

VII. Evaluation

We have evaluated our solutions in two setups: single switch and network of multiple switches. In the first we focus on evaluating our implementation of the anti-spoofing methods under massive attacks, including throughput and latency analysis. In the distributed network setup, we conduct an event-driven simulation to examine the network sub-spaces and portions assignment accuracy, scalability and update efficiency.

A. Anti-Spoofing performance on single switch

We implented the anti-spoofing methods using Open vSwitch 2.3.1, with a POX controller, a client and an HTTP server. The Open vSwitch is running on a server with Intel E3-1270v3 CPU with 32GB ram (4 cores and 8 threads). To simulate a SYN flood attack we use *hping3* [1] to generate SYN packets with different variants of random source ports, source IP addresses and pps (packets per second) rates.

Open vSwitch under DDoS attack: In Fig. 6 we compare two scenarios under a SYN flood attack: (1) normal OpenFlow behavior (2) As before, but with one of our mitigation techniques enabled on the switch by installing the appropriate rules. In this attack the client generates TCP SYN packets and send them to the web server in different rates (random source IP addresses) to saturate the switch and the control path. At the same time, the client sends HTTP get requests to the web server in order to retrieve a web page. In the graph we measure and show the fraction of successful http-get requests at different rates of attack. Under the first setting, without any mitigation method, the web page requests fail when the attack rate exceeds 2.7K pps. When we enable our TCP-Reset anti-spoofing methods in the Open vSwitch, the controller is not notified of any of the attack packets. In this case, the client successfully completes web page http-get operations upto attack rates of 206K pps.

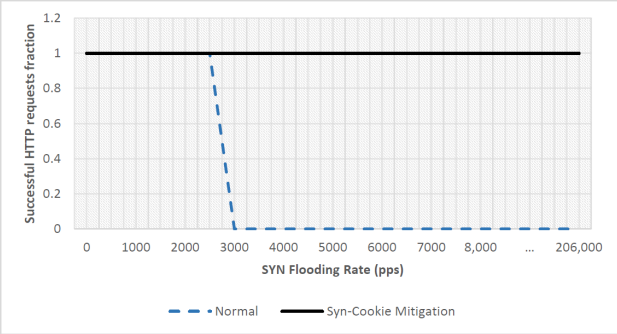


Fig. 6: Fraction of successful legitimate web page requests in reactive mode under SYN flooding attack

Open vSwitch Throughput Analysis (Figure 7): Here we analyze the impact of activating different subsets of the primitive operations on the throughput at which packets can be processed in the Open vSwitch. In the experiment a SYN flood is applied on the Open vSwitch and the switch is configured each time with a different set of primitive operations out of the 9 from §IV. The action in the 'forwarding' in all the cases is to return the packet back on the IN_PORT on which it has arrived, so as to avoid the saturation effect, which would disable the experiment. In the base case no operation is applied on the packet and the maximum throughput before starting to drop packets is 290Kpps. Then in each experiment we add different primitive operations from the list of nine given in §IV. The following sets were tested: (1) No operation (2) Exchange the Ethernet source and destination addresses - ETH swap (3) ETH swap + Write a static TCP source port (4) ETH swap + TCP ports swap (5) ETH swap + IP addresses swap + TCP swap (6) The previous + Writing SEQ, ACK and Flags fields. Note that the Forwarding reference has also relative low throughput (290Kpps), comparing to regular throughput of OVS (which is much higher). This is due to the Open vSwitch architecture which works poorly under SYN-attack even in proactive mode. Note, that in reactive mode, as discussed earlier, the switch stops responding at 2700pps. Open vSwitch has data-path (kernel) microflows/megaflows caching. The decision taken for the first packet is cached, allowing subsequent packets decision in the kernel space without up-calls to the user space. However, during a spoofed SYN attack, since the source IP addresses are all different, almost every packet is up-called to the user-space (in proactive mode). The CPU usage of the `ovs-vswitch` (user-space daemon) process is very high during such an attack. In [30] Wang et.al. show that an OpenFlow network is vulnerable to DDoS attacks in reactive mode due to saturation of the switch control channel. Our experiments as discussed here show that OVS performance degrades in spoofed SYN attacks even in *proactive mode*, due to the caching mechanism.

Figure 7 summarizes the throughput results we obtained by generating high-rate SYN flooding attacks, and detecting the rate in which the vSwitch starts to loose packets. The results

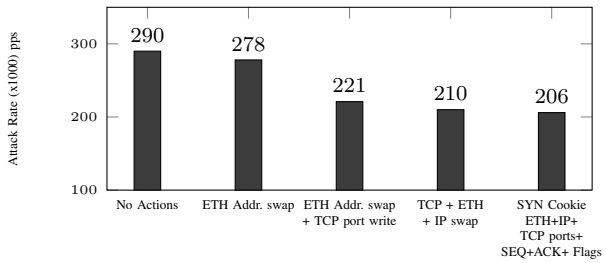


Fig. 7: Open vSwitch throughput in proactive mode under SYN attack with different actions sets

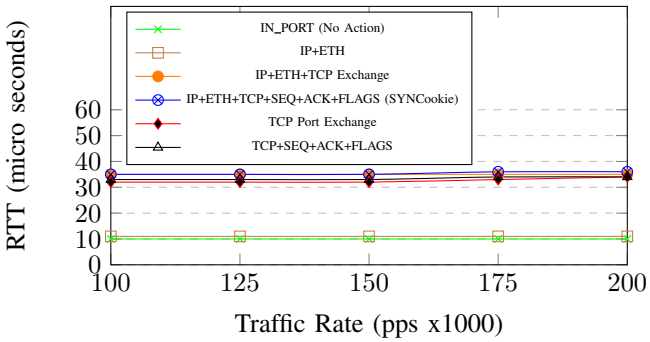


Fig. 8: Data plane RTT for different actions sets

show that the operations have a moderate impact on the throughput, mostly due to the write on the TCP ports and swapping the ETH fields.

Actions Latency Overhead: Here we measure the *latency* introduced by our operations (with similar experiment as in [21]). In this setup, the client sends SYN packets in high rate (100-200K pps). The OVS always returns the packets back to the IN_PORT applying different sets of operations. We measured round trip time of a TCP ping. We compared between the latencies when different sets of operations are applied on the header. As can be seen in Figure 8, the only difference among different sets of operations is between operations that do not include a write to the TCP header (IN_PORT, and IP+ETH exchange) which have 10 micro seconds latency, and operation sets that do write to the TCP header and have 33 micro seconds latency.

B. Distributed Network Simulation

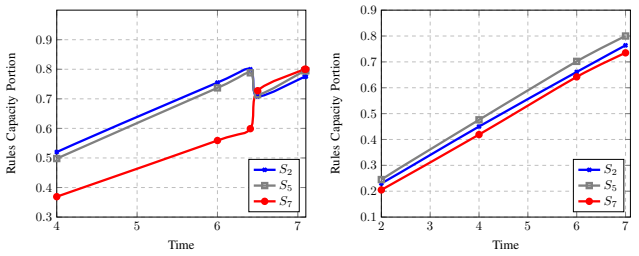
In this setup we demonstrate the dynamic allocation of sub spaces and portions in the network that appears in Fig 3. We perform a simulation in which we measure how the entire flow-space is distributed over the different paths in the network, as well as the distribution among the switches along each path. We first apply the initial allocation and load-balancing algorithms (including an implementation of [31]) assuming uniform traffic distribution. As described in §V-A1 the portions and *wildcard* rules computation is done on source IP prefixes. We use a configurable parameter that indicates how many bits are used to represent a sub space prefix. After we allocate the portions to the switches in the network, we run both attack and legitimate traffic to measure the load of pinholes that are created upon successful authentication on each switch over the different paths in the network. Fig 9 shows the distribution of pinholes as rules in a specific path (Path 3 in Fig. 3) under uniform legitimate traffic distribution. As can be seen in Fig 9b, when using 5 bits to perform the initial allocation in the path, the accuracy of the assignment produces a fair distribution of pinholes under equal traffic distribution, thus under high amount of legitimate traffic (we did not set a timeout for the flow-entries in this simulation) all the switches in the path reach their threshold together. However, when performing initial allocation using 3 bits (portions are not bigger than $1/8$ of the 1* path subspace i.e., 1001*, 1111* etc.), S_2 is saturated before the other switches, and a vertical update occurs (See Fig. 9a). During the vertical update, the path portions are re-divided: sub-portions are released from S_2, S_5 and assigned to S_7. As a result, the allocation of pinholes is spread fairly over the switches. Moreover, since the new assignment is using more bits (additional 4 bits in each vertical update) to divide portions into smaller ones, the redistribution becomes more accurate after each update.

Next, we evaluate the vertical update algorithms accuracy using a non-uniform legitimate traffic distribution. Here we send many more connections to be matched by a single portion of S_7. Fig. 10a shows the pinholes distribution over time before and after a vertical update occurs. Note that here, besides that the algorithm is fairly splitting the active pinholes over all the switches in the path with high accuracy (4 bits), it also adjusts the load among the switches to become fair for additional traffic that arrives after the vertical update. This result is because the loaded portions are divided by counting the number of pinholes in the portion rather than just splitting them as in [31]. By doing that (as described in §V-A1), the actual traffic distribution is taken into account, and produces more accurate portions with concise *wildcard* rules that adaptively fit the actual non-uniform distribution.

Fig. 10b presents the number of controller messages over time in the scenario of Fig 10a. Here we show that even under extreme non-uniform traffic, the number of controller messages during a vertical update is moderate and much lower than the total number of pinholes in the path (In this case, less than half). The number of controller messages is exactly the number of pinholes that are being transferred between the switches (which is the minimal number required to perform a fair allocation).

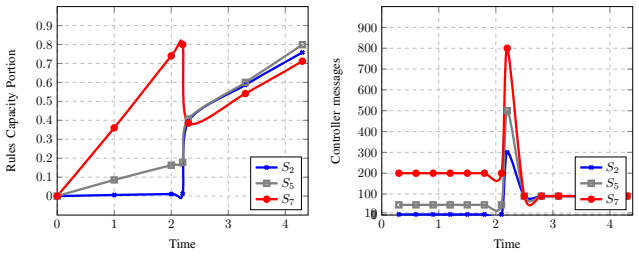
VIII. Conclusions

The concept of turning an SDN switch into a simple middlebox is very appealing. It enables further utilization of the network resources - i.e., the switches, and opens the possibilities for scale out and scale in when the middlebox functionality is required on demand. As we show in this paper the overhead of implementing the anti-spoofing mechanisms on a SDN switch is small. Moreover this functionality protects not only the downstream servers and other middleboxes, it also protects the SDN switch and controller from saturation



(a) Initial allocation using 3 bits accuracy (b) Initial allocation using 5 bits accuracy

Fig. 9: Switches rules capacity during a vertical update on a single path - Uniform traffic distribution



(a) Rules capacity (b) Controller messages

Fig. 10: Resources consumption - Non uniform traffic

and cache misses attacks. An important capability to keep the SDN network operational despite being attacked by a SYN flood.

An interesting question is to understand how far can we push the concept of implementing middlebox functionalities over SDN switches with minimal controller support. While it seems that complex boxes such as a general DPI or session reassembly cannot be implemented due to the high memory and processing requirement, it is not so obvious if all other type of middleboxes functionalities can be implemented using the basic SDN model. This paper shows that at least for the case of anti-spoofing the answer is positive.

Acknowledgments

The authors would like to thank Dan Touitou for helpful discussions about anti-spoofing over the years. This research was supported by the European Research Council under the European Unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259085, and by the Neptune Consortium, administered by the Office of the Chief Scientist of the Israeli Ministry of Industry, Trade, and Labor.

References

[1] http://linux.die.net/man/8/hping3/.
[2] "akamai's security Q3 2015 report". https://www.stateoftheinternet.com/downloads/pdfs/2015-cloud-security-report-q3.pdf/.
[3] D. J. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html.
[4] "DDoS Threat Landscape Report 2013-2014 Security Alliance". https://www.incapsula.com/blog/wp-content/uploads/2015/08/2013-14_ddos_threat_landscape.pdf.
[5] Intel Ethernet Switch Silicon FM6000. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switchfm6000-sdn-paper.pdf.
[6] Noviflow 2122 white pages. http://noviflow.com/products/noviswitch/.
[7] Open vSwitch. http://openvswitch.org.
[8] Pica8 P-5401 white pages. http://www.pica8.com/wp-content/themes/twelve-theme/documents/pica8-datasheet-32x40gbe-p5401.pdf.
[9] "Radware 2014-2015 Global Application and Network Security Report". https://www.radware.com/ert-report-2014/.
[10] "Stateful Connection Tracking in Open vSwitch". http://openvswitch.org/support/ovscon2014/17/1030-conntrack_nat.pdf/.
[11] The P4 Language specification. http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf.
[12] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. Technical report: Network Anti-Spoofing with SDN Data plane. http://sdn-anti-spoofing.net.
[13] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, 44(2):44–51, 2014.
[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.
[15] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.
[16] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. Sane: A protection architecture for enterprise networks. In *Proceedings of USENIX-SS'06*.
[17] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 817–832, August.
[18] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An api for application control of sdns. *SIGCOMM Comput. Commun. Rev.*, 43(4):327–338, August 2013.
[19] Aaron Gember, Aditya Akella, Ashok Anand, Theophilus Benson Benson, and Robert Grandl. Stratos: Virtual middleboxes as first-class entities. *Tech. Rep. TR1771, University of Wisconsin-Madison, 2012*.
[20] Kostas Giotis, Georgios Androulidakis, and Vasilis Maglaris. Leveraging sdn for efficient anomaly detection and mitigation on legacy networks. In *Proceedings of EWSDN '14*, pages 85–90.
[21] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and T.V. Lakshman. Application-aware data plane processing in sdn. In *Proceedings of HotSDN '14*, pages 13–18.
[22] Tal Mizrahi, Ori Rottenstreich, and Yoram Moses. Timeflip: Scheduling network updates with timestamp-based tcam ranges. 2015.
[23] G. Pazi, D. Touitou, A. Golan, and Y. Afek. Protecting against spoofed dns messages, April 10 2003. US Patent App. 10/251,912.
[24] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012*, pages 323–334.
[25] Seungwon Shin and Guofei Gu. Attacking software-defined networks: A first feasibility study. In *Proceedings of HotSDN '13*, pages 165–166.
[26] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of CCS '13*, pages 413–424.
[27] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection, 2000.
[28] D. Touitou, G. Pazi, Y. Shtein, and R. Tzadikario. Using TCP to authenticate IP source addresses, January 27 2005. US Patent App. 10/792,653.
[29] D. Touitou and R. Zadikario. Upper-level protocol authentication, May 19 2009. US Patent 7,536,552.
[30] An Wang, Yang Guo, Fang Hao, T.V. Lakshman, and Songqing Chen. Scotch: Elastically scaling up sdn control-plane using vswitch based overlay. In *Proceedings of CoNEXT '14*, pages 403–414.
[31] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Proceedings of Hot-ICE'11*, pages 12–12.