Network Anti-Spoofing with SDN Data plane - Technical Report

I. Challenge Response Methods - Extended Version

Four different anti-spoofing methods [1], [8], [11], [12] are implemented in this paper over SDN using OpenFlow 1.5 and P4.

A. Spoofed SYN Flooding Mitigation

SYN flood DDoS attacks exhaust the server resources by flooding it with new connection requests (SYN packet for TCP), for each of which the server allocates a new connection record structure (called *Transmission Control Blocks* or TCB) [4]. More recently a similar attack, called *control plane saturation attack*, uses the same SYN packets flood [10], [13], which now exhausts the bandwidth of the control channel between an SDN switch and a controller, as well as exhausting the switch flow-tables entries space. Such flooding attacks are usually spoofed, making their mitigation more difficult.

The most effective mitigation methods against spoofed SYN flood attacks are different variations and combinations of the SYN Cookie method [1]. The SYN cookie is basically a challenge response technique that does not maintain any state on the server (mitigator) side. Instead, see Fig. 1, 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. The mitigation methods we implement are:

TCP Proxy In this method [1], see Fig. 1a, after being authenticated the mitigating device performs a hand-shake with the server and from here on acts as a proxy between the client and server for the entire duration of the connection. For each packet, in both directions, it has to adjust its server side sequence number as part of the TCP splicing. This is the main disadvantage of this method; the mitigation device has to see and process all the traffic (at least of the first connection from that client) in both directions, which may present a significant overhead.

HTTP Redirect Here [12], 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 because of the installed pinhole.

TCP Safe Reset In this weak version of the SYN cookie [11], see Fig. 1c, the switch responds to the initial SYN message with an SYN-ACK containing a bogus ACK number. If the client is compliant with RFC 793 (TCP) [5], it immediately responds to the bogus ACK number with a RST packet containing the bogus ACK number as its sequence number (thus being authenticated). Then, 1 second after this, the client initiates a new connection by sending a new SYN request. This technique assumes no application-level awareness and requires only TCP compliance, thus is suitable not only for HTTP protocols but also SMTP and others.

TCP Reset This method is similar to the HTTP Redirect method except that a RST packet is sent from the mitigating device to the client instead of an HTTP-Redirect response upon handshake completion, see Fig. 1b. This technique do require application awareness, thus suitable with all TCP connections that retry to connect when a RST message is received.

B. DNS Flooding Mitigation

Following [8] the techniques of the last subsection can be applied to mitigate a spoofed DNS request flood in SDN. A DNS spoofed attack exhausts the server bandwidth, CPU, and other resources by sending a large amount of spoofed DNS requests (over UDP port 53). Although the majority of DNS traffic is carried over UDP, DNS may also be carried over TCP. Thus, when the mitigating device receives a DNS UDP request from a source it has not authenticated before, it can force the client to repeat its request over TCP, and then perform a TCP authentication using a cookie as described in §I-A. In order to force the client to repeat its first 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 should be set in the response. This method is similar to the *HTTP-Redirect* method. As shown in Figure 2, the switch responds to the original UDP request by setting the *TC* bit. Then after the TCP handshake is completed, it forwards the TCP DNS request to the controller that translates the TCP to UDP both for the request and response from the DNS server. Finally, the controller installs rules (pinhole) to allow future DNS UDP requests from this authenticated source. Notice this method requires in the case of OpenFlow 1.5 an extension to allow setting the *TC* bit in the DNS response packet.

II. Example: TCP-RESET implementation

Open vSwitch Code Changes: We added and modified 390 code lines in 26 files in Open vSwitch 2.3.1 to support the following missing matching and additional actions that are used by our methods:

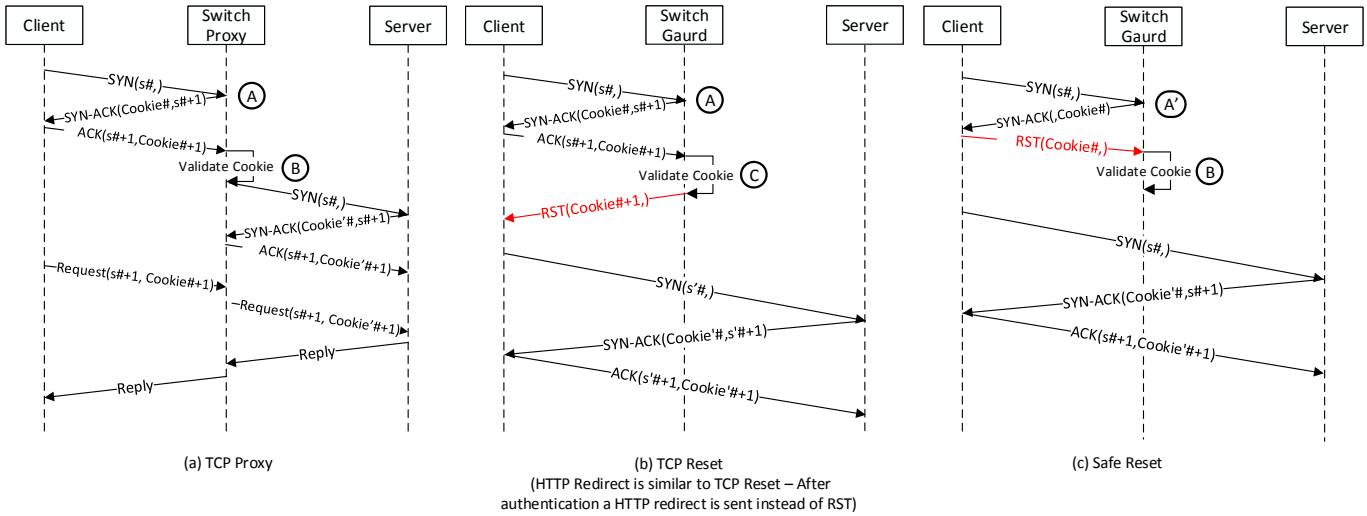

Fig. 1: SYN Anti-Spoofing Methods

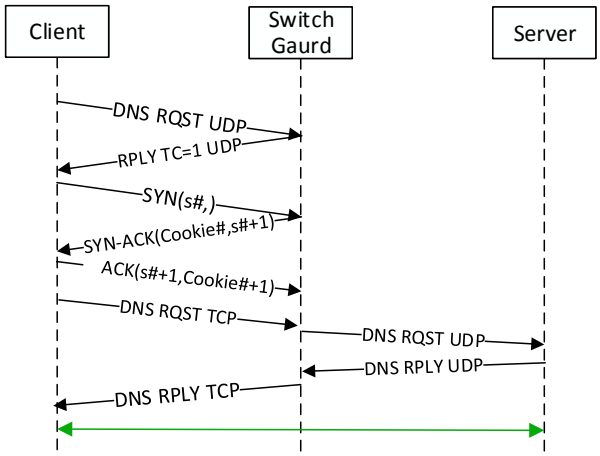

Fig. 2: DNS spoofed attack mitigation

For `TCP-Reset` and `HTTP-Redirect`:

- `write to SEQ number field`
- `match/write to ACK number field`
- `increment SEQ number field by one.`
- `write to TCP-Flags field (the field is supported in OF 1.5 but write is not implemented in OVS)`

For `Safe Reset`:

- `write to ACK number field`
- `match SEQ number field`
- `write to TCP-Flags field`

For `TCP Proxy`:

- `write to SEQ number field`
- `match/write to ACK number field`
- `write to TCP-Flags`
- `add a fixed value to SEQ/ACK number fields`

We demonstrate the details of the implementation in Open vSwitch on one of the anti-spoofing methods, the TCP-Reset.

After the controller has allocated cookies for each partition, (in this example the partition is done using subnets for simplicity) we show in Figure 3a an example of two rules that should be set in the switch flow-table in order to implement the the TCP-Reset method. The first rule (Fig. 3a lines 2-6) matches SYN packets, write the cookie to the SEQ number field, and transform the SYN packet to a SYN-ACK packet, for each matched packet. The second rule (Fig. 3a lines 8-14) matches an ACK packet with the correct cookie, then transforms it to a RST packet, and finally installs (using the `learn` action) another rule to allow traffic from the authenticated client. These two rules are for a single partition; Similar pairs of rules must be installed for each partition. We use *ovs-ofctl* tool [2] to show the rules as a flow-table dump:

First Rule: In our example, as appears in Figure 3a (lines 2-6), all the SYN packets from `10.0.0.0/8` should be answered by replies with SEQ = 347834245. The rule contains the following actions:

```
move:NXM_OF_IP_DST[]->NXM_NX_REG0[]
move:NXM_OF_IP_SRC[]->NXM_OF_IP_DST[]
move:NXM_NX_REG0[]->NXM_OF_IP_SRC[],
//Actions to swap the IP addresses

move:NXM_OF_TCP_SRC[]->NXM_OF_TCP_DST[]
mod_tp_src:80
//These actions handle the TCP ports exchange,
//We assumed http, but a full swap can be used.

move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[]
mod_dl_src:00:00:00:00:00:02
//These actions swap the ETH addresses.

mod_tcp_seq:347834245
//This action sets 347834245 in the SEQ field

mod_tcp_ack:1
//This action responsible to set SEQ+1 in
//the ACK field (1 is a reserved number
//for this operation)

mod_tcp_flags:18
//sets 0x12 in the TCP-flags field
//Raise the ACK in addition to the SYN flag
```

Fig. 3: TCP Reset anti-spoofing implementation in Open vSwitch example

```
0   $ ovs-ofctl dump-flows s1
1   NXST_FLOW reply (xid=0x4):
2   cookie=0x0, duration=26.274s, table=0, n_packets=1, n_bytes=74, idle_timeout=6000, hard_timeout=6000, idle_age=17,
3   priority=65534,tcp,in_port=1,nw_src=10.0.0.0/8,tcp_flags=+syn
4   actions=move:NXM_OF_IP_DST[]->NXM_NX_REG0[],move:NXM_OF_IP_SRC[]->NXM_OF_IP_DST[],move:NXM_NX_REG0[]->
5   NXM_OF_IP_SRC[],move:NXM_OF_TCP_SRC[]->NXM_OF_TCP_DST[],move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[],
6   mod_dl_src:00:00:00:00:00:02,mod_tp_src:80,mod_tcp_seq:347834245,mod_tcp_ack:1, mod_tcp_flags:18,IN_PORT
7
8   cookie=0x0, duration=31.370s, table=0, n_packets=1, n_bytes=66, idle_timeout=6000, hard_timeout=6000, idle_age=17,
9   priority=65534,tcp,in_port=1,nw_src=10.0.0.0/8,tcp_ack=347834246,tcp_flags=+ack
10  actions=load:0x2->NXM_NX_REG1[1..2],learn(table=0,hard_timeout=60,priority=65535, eth_type=0x800, NXM_OF_IP_SRC[],
11  NXM_OF_IP_DST[],output:NXM_NX_REG1[1..2]),move:NXM_OF_IP_DST[]->NXM_NX_REG0[],move:NXM_OF_IP_SRC[]->
12  NXM_OF_IP_DST[],move:NXM_NX_REG0[]->NXM_OF_IP_SRC[],move:NXM_OF_TCP_SRC[]->NXM_OF_TCP_DST[],
13  move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[],mod_dl_src:00:00:00:00:00:02,mod_tp_src:80,mod_tcp_seq:347834246,
14  mod_tcp_ack:0, mod_ tcp_flags:4, IN_PORT
```

(a) Open vSwitch installed-rules example to implement TCP-Reset anti-spoofing method for a single SEQ number (subnet). **Lines #2-6** represent the first rule to match SYN packets from 10.0.0.0/8, transform them into SYN-ACK packets, and write 347834245 as a SEQ number. **Lines #8-14** represent the second rule to match a correct ACK packet (with ACK#=347834246) and transform it into an RST packet. The second rule also use learn action to "allow" locally this flow upon authentication. The bolded actions show our modifications in OVS 2.3.1 to write to additional TCP fields.

No.	Time	Source	Destination	Protocol	Length	Info
4	0.785791000	10.0.0.1	10.0.0.2	TCP	74	38676 > http [SYN] Seq=2379872098 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=130291462 TSecr=0 WS
5	0.785865000	10.0.0.2	10.0.0.1	TCP	74	http > 38676 [SYN, ACK] Seq=347834245 Ack=2379872099 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=
6	0.785894000	10.0.0.1	10.0.0.2	TCP	66	38676 > http [ACK] Seq=2379872099 Ack=347834246 Win=29696 Len=0 TSval=130291466 TSecr=130291462
7	0.785960000	10.0.0.2	10.0.0.1	TCP	66	http > 38676 [RST] Seq=347834246 Win=29696 Len=0 TSval=130291466 TSecr=130291462
8	0.786221000	10.0.0.1	10.0.0.2	TCP	74	38677 > http [SYN] Seq=107207042 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=130291466 TSecr=0 WS=
9	0.787769000	10.0.0.2	10.0.0.1	TCP	74	http > 38677 [SYN, ACK] Seq=2310708400 Ack=107207043 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=
10	0.787793000	10.0.0.1	10.0.0.2	TCP	66	38677 > http [ACK] Seq=107207043 Ack=2310708401 Win=29696 Len=0 TSval=130291467 TSecr=130291466
11	0.787845000	10.0.0.1	10.0.0.2	HTTP	172	GET / HTTP/1.1
12	0.787911000	10.0.0.2	10.0.0.1	TCP	66	http > 38677 [ACK] Seq=2310708401 Ack=107207149 Win=29184 Len=0 TSval=130291467 TSecr=130291467
13	0.792910000	10.0.0.2	10.0.0.1	TCP	83	[TCP segment of a reassembled PDU]
14	0.792925000	10.0.0.1	10.0.0.2	TCP	66	38677 > http [ACK] Seq=107207149 Ack=2310708418 Win=29696 Len=0 TSval=130291468 TSecr=130291468
15	0.792948000	10.0.0.2	10.0.0.1	TCP	204	[TCP segment of a reassembled PDU]
16	0.792953000	10.0.0.1	10.0.0.2	TCP	66	38677 > http [ACK] Seq=107207149 Ack=2310708556 Win=30720 Len=0 TSval=130291468 TSecr=130291468

(b) A Wireshark snapshot of a client webpage request following the rules (figure a) that implement the TCP-Reset anti-spoofing method for a single subnet (10.0.0.0/8 – SEQ=347834245). **Packets #4-#6** represent the TCP authentication between the client and the OVS switch using a SYN cookie according to the rules that appear above. The first OVS rule is modifying the SYN packet (packet #4) into a SYN-ACK packet (packet #5) with SEQ#=347834245. The second OVS rule is modifying the ACK packet (packet #6) into a RST packet (packet #7) and allowing the flow to be forwarded directly to the server. Then **packets #8-#10** represent the TCP authentication between the client and the real HTTP server after the client has received the RST packet.

```
IN_PORT
//return the SYN-ACK 'cooked'
//packet back to the sender.
```

Second Rule: This rule matches and validates ACK packets from `10.0.0.0/8` that have the correct ACK (347834246) number, and convert it into an RST packet that is sent back to the sender. As appears in Figure 3, it contains the appropriate match attributes (nw_src=10.0.0.0/8, +ack, ack_num = 347834246) and contains similar actions as in the first rule (swap the IP, ETH addresses, and TCP ports). In this rule we also demonstrate how authenticated flows can be enabled in the switch without notifying the controller. We use the *learn* action (nx_action_learn - Nicira extensions) which causes the generation of a new table entry triggered by the `current packet` but for the `future packets`. The learn action allows us to generate a new table entry that allows traffic from the authenticated source IP to the server, locally on the switch. If such a function is not supported, the authenticated ACK should be sent to the controller, that will reactively set a similar pinhole flow-entry. Formally, the actions that appear in this rule and that differ from those in the first rule are:

```
learn(table=0,hard_timeout=60,priority=65535,
eth_type=0x800,NXM_OF_IP_SRC[],NXM_OF_IP_DST[],
output:NXM_NX_REG1[1..2])
//The nx_action_learn as described above

mod_tcp_seq:347834246
```

```
//The SEQ number of the RST packet must be
//the same as the ACK number (347834246).

mod_tcp_flags:4
//Raise RST flag.
```

In Figure 3b, we show a TCP handshake (a Wireshark snapshot) in action, based on the rules we described above. The first packet is a SYN packet (#4) sent by the client (10.0.0.1) to the server. The switch replies by an SYN-ACK message (#5-generated by the first rule) encapsulating the corresponding cookie (SEQ number = 347834245). The client responds with an ACK message (#6 - with ACK number = 347834246) to complete the authentication. Then the switch replies by an RST message (#7 - generated by the second rule). In addition, a new 'allow' rule is being installed by the second rule for the authenticated flow. The client then performs another TCP handshake with the real server (#8-#10) and complete a successful HTTP request.

III. Additional Implementation Details

Open vSwitch Internal Changes The new TCP fields that our new actions deal with were added to `ovs_key_tcp` in order to fit exactly to the same API and structures of other similar field actions. We used `inet_proto_csum_replace2(/4)` to calculate the checksum exactly in the same way that TCP ports modifications is performed in the kernel space.

POX Controller: We used POX [3] controller, and made only few modifications to support the installation of rules that include the new actions we added to the Open vSwitch. We slightly modified `nicira.py` and `libopenflow_01.py` to insert three new actions: `OFPAT_SET_TCP_FLAGS`, `OFPAT_SET_TCP_SEQ`, and `OFPAT_SET_TCP_ACK` conform with the structure of the existing actions such as `OFPAT_SET_TP_SRC` and `OFPAT_SET_TP_DST`

IV. Implementation under the P4 Environment

P4 program consists of header, parser, table/actions and a control flow. First we added a TCP header definition to allow reading and matching based on the TCP fields. Since we modified the TCP fields, a TCP checksum recalculation should also be defined in the parser and must be explicitly declared within a P4 program. Then we defined logical tables that are matched based on the partitions that we described in §IV, and applied the relevant actions that modify the SYN and ACK packets including writing the corresponding SEQ number into the packet. As appears here in the code (represents the SYN matching table and the set cookie action), writing a P4 action reflects our intention in a very simple way:

```
action set_seq(seq_num, port) {
    modify_field(standard_metadata.egress_spec,
                 port);
    modify_field(r_metadata.src_ipv4,
                 ipv4.srcAddr);
    modify_field(r_metadata.src_mac,
                 ethernet.srcAddr);
    modify_field(r_metadata.dst_port,
                 tcp.dstPort);
    modify_field(ipv4.srcAddr, ipv4.dstAddr);
    modify_field(ethernet.srcAddr,
                 ethernet.dstAddr);
    modify_field(tcp.dstPort, tcp.srcPort);
    add_to_field(tcp.seqNum, 1);
    modify_field(tcp.ackNum, tcp.seqNum);
    modify_field(tcp.seqNum, seq_num);
    add_to_field(ipv4.ttl, -1);
}

table handle_syn {
    reads {
        ipv4.srcAddr : ternary;
        ipv4.dstAddr : exact;
        tcp.srcPort : ternary;
        tcp.tcpFlags1 : exact;
    }
    actions {
        set_seq;
        _drop;
    }
    size: 512;
}
```

The P4 experimental platforms are developing these days. We used a behavioral model that has a software simulation with a Mininet [6] plugin. The model compiles a P4 program into an executable simulation component, and also generates the API to be used to populate the flow tables' rules during run-time. This allowed us to test the behavior of our packet modifications using Mininet, a TCP client (*wget*) and *tcpdump*.

V. Vertical Dynamic Updates - Extended Version

The pinholes migration described above is an update problem of states. Usually the update should be consistent. A *consistent update*[9] 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* [9] or *TimeFlip* [7]. However, they come with extra cost of version numbers and tagging. In our case we can do a simpler solution, by relaxing the consistent requirement.

Here we describe a simpler sub portion migration. Three properties must be maintained during this update: (i) unauthenticated SYN packets must not reach the target. (ii) authenticated connections must not be disrupted. (iii) new legitimate connections are not blocked during the update.

If a certain range of flows is covered and goes through anti-spoofing by more than one switch along the path, the corresponding connection is simply authenticated multiple times, but eventually an authenticated connection is established with the server. Note that simply installing the pinholes and mitigation rules on an additional switch and then removing them from the original switch may introduce hazardous scenarios in which new legitimate connections as well as packets of old pinholes are erroneously dropped. For example, if we

first copy a sub portion pinholes from switch s_{j_1} to s_{j_2} and then install new rules on s_{j_2} to associate it with a new sub-portion that is associated with s_{j_1}, the new pinholes that were created meanwhile (after the copy) may be disrupted by the new ACK rules since a new pinhole has not been updated on s_{j_2}.

To allow a migration mechanism that satisfies the properties defined above, we make sure that during the update every new authentication completion in the migrated sub portion is installing pinholes on both s_{j_1} and s_{j_2}. This requirement is enforced by the controller that first installs the pinholes and then sends the redirect message to avoid race conditions (as described in §IV).

The following update sequence that satisfies our requirements is used to migrate redirection-based mitigation sub portion from switch s_{j_1} to s_{j_2}:

1	Configure the Controller to install new $D_{i_{j_1}}$ pinholes both on s_{j_1} and s_{j_2}.
2	Copy all $D_{i_{j_1}}$ pinholes from s_{j_1} to s_{j_2}.
3	Install $D_{i_{j_1}}$ matching rule (#1 on §IV Table I) on s_{j_2}.
4	Remove $D_{i_{j_1}}$ matching rule from s_{j_1}.
5	Remove $D_{i_{j_1}}$ pinholes from s_{j_1}.

Note that the rules we use to match a portion D_{i_j} are not using subtractive terms, hence migrating a set of rules that represent a sub portion can not affect any other disjoint portion.

When migrating *proxy-based* mitigation rules along the path, we can not use the above sequence. *Proxy* methods are installing *splicing rules* rather than pinholes. A packet of an authenticated connection can pass multiple pinholes, but can not pass more than exactly one set (in both directions) of *splicing rules*. Thus, to migrate *proxy-based* policy we will use the generic *two-phase commit* mechanism [9].

The above sequence migrating a portion D_{i_ℓ} from one switch to another is performed by the SDN controller. Note that although the switch-to-controller path might become a bottleneck in some cases, here the number of rules moved during a migration is proportional to legitimate traffic rather than attack volume. Furthermore, a migration of a big portion may be carried out piece wise.

REFERENCES

[1] D. J. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html. Online; accessed 2015-08-25.

[2] Open vSwitch ovs-ofctl - administer OpenFlow switches. http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt. Online; accessed 2015-09-08.

[3] POX - SDN Controller. http://www.noxrepo.org/. Online; accessed 2015-09-08.

[4] RFC 4987 - "TCP SYN Flooding Attacks and Common Mitigations". https://tools.ietf.org/html/rfc4987. Online; accessed 2015-08-25.

[5] RFC 793 TCP - "Transmission Control Protocol". https://tools.ietf.org/html/rfc793. Online; accessed 2015-08-25.

[6] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 19:1–19:6.

[7] MIZRAHI, T., ROTTENSTREICH, O., AND MOSES, Y. Timeflip: Scheduling network updates with timestamp-based tcam ranges.

[8] PAZI, G., TOUITOU, D., GOLAN, A., AND AFEK, Y. Protecting against spoofed dns messages, Apr. 10 2003. US Patent App. 10/251,912.

[9] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012*, pp. 323–334.

[10] SHIN, S., AND GU, G. Attacking software-defined networks: A first feasibility study. In *Proceedings of HotSDN '13*, pp. 165–166.

[11] TOUITOU, D., PAZI, G., SHTEIN, Y., AND TZADIKARIO, R. Using TCP to authenticate IP source addresses, Jan. 27 2005. US Patent App. 10/792,653.

[12] TOUITOU, D., AND ZADIKARIO, R. Upper-level protocol authentication, May 19 2009. US Patent 7,536,552.

[13] WANG, A., GUO, Y., HAO, F., LAKSHMAN, T., AND CHEN, S. Scotch: Elastically scaling up sdn control-plane using vswitch based overlay. In *Proceedings of CoNEXT '14*, pp. 403–414.