FlowCloak: Defeating Middlebox-Bypass Attacks in Software-Defined Networking

Kai Bu*, Yutian Yang*, Zixuan Guo*, Yuanyuan Yang•, Xing Li*, Shigeng Zhang°
*College of Computer Science and Technology, Zhejiang University
•Stony Brook University, °Central South University

***Abstract*—Software-Defined Networking (SDN) greatly simplifies middlebox policy enforcement. Middleboxes need tag packet headers to avoid forwarding ambiguity on SDN switches. In this paper, we present a new attack, called middlebox-bypass attack, to breach SDN-based middlebox policy enforcement. Such an attack manipulates a compromised switch to locally tag attacking packets without handing them over to the attached middlebox for inspection. Existing SDN security solutions, however, cannot detect the middlebox-bypass attack under practical constraints of efficiency, robustness, and applicability. We design and implement FlowCloak, the first protocol for per-packet real-time detection and prevention of middlebox-bypass attacks. FlowCloak enables middleboxes to generate tags that are probabilistically unknown to an attacker and confines it to only random guessing. We propose a multi-tag verification technique to address the tradeoff between FlowCloak robustness and TCAM usage by tag verification rules on the egress switch. Experiment results show that dozens of verification rules can confine the attacking probability under 0.1%. FlowCloak imposes only a 0.3 ms packet processing delay on middleboxes and no obvious delay on the egress switch.**

I. Introduction

Since the rise of middleboxes, the intricacy of enforcing middlebox policies never ceases [1]. Middleboxes are specialized appliances providing services like traffic engineering and packet inspection beyond the capability of routing and forwarding devices. A survey in 2012 shows that middleboxes account for a third of devices in the 57 surveyed enterprise networks [2]. It also reveals the high cost of managing distributed heterogeneous middleboxes—a mere 10 middleboxes may demand a management team of 6-25 personnel. Furthermore, distributively configuring middleboxes is complex and error prone—over 50% of the interviewed administrators regard misconfiguration as the most common cause of middlebox failures. Sherry *et al.* pioneer the idea of reducing middlebox management cost by outsourcing middlebox functions (except internal firewalls) to the cloud [3]. A feasible outsourcing solution should overcome various challenges regarding consistency [4], security [5], and privacy [6]. Another innovation for easing middlebox management is to consolidate middlebox services into a commodity server by decomposing middleboxes and eliminating redundant processing modules [7]–[9].

Without radical infrastructure changes like outsourcing and consolidating middleboxes, SDN also offers great flexibility in enforcing middlebox policies [4], [8]–[12]. Such flexibility stems from an SDN controller's global network view of topology, routing, and statistics. This information is reported by a number of distributed SDN switches. To enforce network policies, the controller—with the help of hosted management applications—translates the policies into rules and installs them on switches. The controller can use crafted rules to direct specific packets through a chain of middleboxes, without wrestling with the complexity of middlebox deployment otherwise necessary in prior-SDN networks [10], [12]. Since SDN rules match with packets against specified header fields, forwarding ambiguity arises when a packet goes through a switch more than once, each time toward a different next hop. This forwarding ambiguity is further exacerbated by middleboxes like NAT. Without their modification strategies as a priori, it is hard to pre-configure rules for modified packets. Stateful processing against forwarding ambiguity requires middleboxes to tag packet headers [11]. The tagging scheme is lightweight and augments only tens of lines of code to middleboxes.

In this paper, we identify a new attack called *middlebox-bypass attack* against SDN-based middlebox policy enforcement. A middlebox-bypass attack occurs when a compromised switch locally tags packets without forwarding them to its attached middlebox. Such a policy breach leads to potential performance degradation and especially security threats. Countermeasures are solicited as compromising SDN switches is no longer a hypothesis. An attacker can compromise a switch by exploiting its vulnerabilities or reconfiguring it after getting the access [13]–[15]. Compromised servers and switches were exploited to carry out 55% of all attacks monitored by IBM in 2014 [16]. Emerging software switches like Open vSwitch are easier to compromise than traditional physical switches [17]. Although verifying rule correctness [18] and effectiveness [19] is fully investigated, detecting rule violation by compromised switches draws only limited attention. One may inject probe packets to the data plane [20], [21]. If a switch does not process a probe packet as expected, the switch is suspected as compromised. One may also use flow-statistics [17], [21]–[23]. When a compromised switch injects, drops, or reroutes packets, flow statistics of switches along a path may fluctuate.

The middlebox-bypass attack, however, can evade both probe- or statistics-based detection because it neither reroutes packets from switches nor incurs flow statistics fluctuation across switches. One may enable middleboxes to report probe results and flow statistics. However, probe-based detection promises per-packet detection but lacks detection accuracy. A compromised switch likely misbehaves for only target production packets. Flow-statistics can more comprehensively track switch forwarding behaviors, but it is hard to estimate the exact statistics of each flow path. Statistics-based detection methods convict a flow fluctuation that exceeds a prede-

fined threshold; this limits detection precision. Furthermore, statistics emerge after packets flee. They thus discourage real time detection of suspicious packets, let alone quarantining them from end hosts. Besides, both solutions are limited in efficiency as they compete with conventional management operations (e.g., topology discovery and rule update) for scarce control channel bandwidth [24].

We present the design and implementation of FlowCloak, the first protocol for per-packet real-time prevention and detection against middlebox-bypass attacks. FlowCloak enables middleboxes to mark packet headers with packet-specific tags that are deterministically synchronized across middleboxes and the egress switch, yet probabilistically unknown to other switches. Middleboxes have an intrinsically secure design because they contain sufficient hardware and software resources for affording advanced security solutions. If the egress switch is compromised, there is no next-hop switch to check its forwarding correctness and thus leaves open the possibility for its any injection, dropping, or reroute attack. The integrity of the egress switch is therefore imperative [17], [20]–[23]; but its adopted security solution might be as expensive as infeasible to augment to all switches. Tags generated by a middlebox will be verified by another middlebox or the egress switch. Since an attacker is unaware of the tagging strategy, it can hardly forge correct tags that the bypassed middlebox would use for specific packets. This not only lowers attack probability but also raises detection probability.

A major challenge is practically efficient tag verification on the egress switch. SDN switches follow controller-designated rules to process packets at the line speed and do not support the complex computation over packet headers supported by middleboxes. While middleboxes can use any computation to generate tags as long as they fit the size of unused bits in packet headers, the egress switch has to emulate the computation using simple tag verification rules. Consider for example that middleboxes compute t-bit tags over h header bits. The egress switch needs 2^h rules to verify the h-to-t bit mapping. A larger h improves inference resistance but increases the number of rules. However, SDN switches install rules in expensive and power-hungry Ternary Content Addressable Memory (TCAM), which can accommodate only thousands of OpenFlow rules [24]. One may work around the TCAM constraint by mounting an additional middlebox to the egress switch. The induced delay by packets traveling through the additional middlebox, however, may not be affordable. Even a delay of hundreds of milliseconds per transaction can deprive Internet service providers of million-dollar profits [25].

We propose a multi-tag verification technique to make the number of tag verification rules practically affordable to the egress switch. The idea is to divide the preceding h-to-t mapping process into subprocesses of h_i-to-t_i mapping, where we have $\Sigma h_i = h$ and $\Sigma t_i = t$. This enables tag verification on the egress switch to leverage flow table pipeline, which has been supported since OpenFlow Switch Specification Version 1.1.0 in 2011 [26]. That is, each table verifies one h_i-to-t_i mapping and only passing all tables succeeds tag verification.

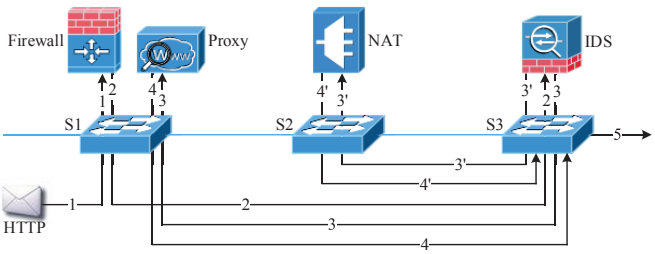

Fig. 1. Stateful packet processing against forwarding ambiguity in SDN-based middlebox policy enforcement [10], [11]. Example policy chains Firewall-IDS-Proxy (i.e., packet instances of 1-2-3-4-5) and Firewall-IDS-NAT (i.e., packet instances of 1-2-3′-4′-5) need middlebox-added tags in packet headers to differentiate processing states, that is, prior or post which middlebox.

The number of tag verification rules now shrinks to $\Sigma 2^{h_i}$, instead of a much larger $2^h = 2^{\Sigma h_i}$. Moreover, each of the $\Sigma 2^{h_i}$ rules is shorter than original rules. We thus can leverage variable-length flow tables to save more TCAM space [27]. To guarantee robustness, FlowCloak may map different packet headers to the same tag while the same packet header to different tags. This confines an attacker to random guessing.

In summary, we make the following contributions to securing SDN-based middlebox policy enforcement.

- Identify a new attack called middlebox-bypass attack that breaches middlebox policies in SDN (Section II).
- Propose FlowCloak as the first protocol to not only detect but also prevent middlebox-bypass attacks (Section III). Our measurement study demonstrates the feasibility of generating packet-specific tags toward per-packet real-time attack detection (Section VII-A).
- Propose a multi-tag verification technique to address the tradeoff between FlowCloak robustness and TCAM constraint on the egress switch (Sections IV-V).
- Implement FlowCloak using Snort, the most widely deployed IDS/IPS (Section VI). Our modification takes only 387 lines of C code over Snort's 293K lines. Experiment results (Section VII) show that a 10-bit tag suffices to limit attacking probability within 0.1%; it requires only dozens of rules on the egress switch. FlowCloak imposes only a 0.3 ms packet processing delay on a middlebox and no obvious delay on the egress switch.

II. Problem

In this section, we define the middlebox-bypass attack in SDN and explore the goals and challenges for countermeasure design. We find adapting existing malicious-switch countermeasures limits efficacy, efficiency, security, and applicability.

A. Middlebox Meets SDN

To address forwarding ambiguity for enforcing middlebox policies, middleboxes have incentives to enable stateful packet processing albeit this might impose minor software update [11]. Simply translating intricate off-packet topology into control logic is not SDN friendly [10], [11]. Fayazbakhsh *et al.* [11] propose that middleboxes add tags in packet headers to track their processing states. The tagging scheme is lightweight and effective yet without switch modification.

TABLE I

TAG-AUGMENTED RULES FOR ENFORCING POLICY CHAINS FIREWALL-IDS-PROXY (I.E., PACKET INSTANCES OF 1-2-3-4-5) AND FIREWALL-IDS-NAT (I.E., PACKET INSTANCES OF 1-2-3'-4'-5) IN FIG. 1. (*pkt ins: Packet Instance.*)

Switch	Tag-augmented Rule		
	matching	action	*pkt ins*
S1	protocol=http, tag=null	fwd to Firewall	1
	protocol=http, tag=Firewall	fwd to S3	2
	protocol=http, tag=IDS	fwd to Proxy	3
	protocol=http, tag=Proxy	fwd to S3	4
S3	protocol=http, tag=Firewall	fwd to IDS	2
	protocol=http, tag=IDS	fwd to S1	3
	protocol=http, tag=Proxy	tag=null, fwd out	5
	src=voter, tag=IDS	*fwd to S2*	$3'$
	tag=NAT	*tag=null, fwd out*	5
S2	*src=voter, tag=IDS*	*fwd to NAT*	$3'$
	tag=NAT	*fwd to S3*	$4'$

Take the policy chain S1-Firewall-S1-S3-IDS-S3-S1-Proxy-S1-S3 in Fig. 1 as an example. When S1 first receives a non-tagged HTTP packet, it knows that the packet just entered the network and should be directed to Firewall. To enable stateful packet processing, Firewall adds a tag in unused header fields such as VLAN tags and MPLS labels [10]. When S1 receives the packet with tag Firewall, S1 knows that the packet has been processed by Firewall. S1 then forwards it to switch S3. S3 and subsequent switches follow a similar process. All their forwarding decisions are enforced by tag-augmented rules dictated by the controller. Table I illustrates rules on S1 and S3 for enforcing policy chain Firewall-IDS-Proxy. Note that egress switch S3 needs to untag the packet before sending it out; this avoids affecting checksum verification on end hosts.

Another benefit of such a tagging scheme is to address the challenge raised by header-modifying middleboxes. Consider a policy chain Firewall-IDS-NAT (i.e., packet instances of 1-2-3′-4′-5) in Fig. 1 for example. Assume that it aims to protect source privacy of a voter during an anonymous electronic voting. NAT rewrites packet headers before packets go outside the network. Although the modified packets are agnostic to the controller, it can use the tag NAT to configure rules for processing NAT-processed packets. Corresponding rules on S3 and S2 are also illustrated (in *italic*) in Table I.

B. Middlebox-Bypass Attack

We find that middlebox-bypass attacks remain the missing piece in the rising battle against compromised switches. In a **middlebox-bypass attack**, a compromised switch does not forward likely suspicious packets to its attached middlebox for a security check. Instead, it directly tags these packets as if they were verified by the bypassed middlebox. The switch can easily recognize expected tags in local tag-augmented rules. As illustrated in Fig. 2, the compromised switch S2 tags a packet with IDS (i.e., packet instance $3'$) without actually handing over the packet to IDS for inspection. Next-hop switch S3, however, deems the packet inspected based on tag IDS in the packet header. The middlebox-bypass attack thus unleashes suspicious packets and breaches security policies.

Adversary Model. We assume that an attacker can compromise any switches on a middlebox policy chain except the egress switch (Section I). The integrity of the egress switch is

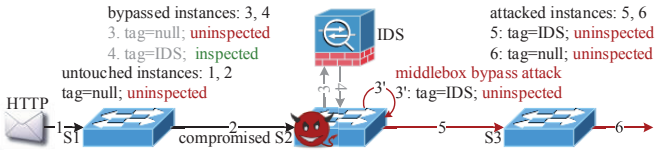

Fig. 2. Middlebox-bypass attack example. When enforcing the policy chain S1-S2-IDS-S2-S3 (i.e., packet instances of 1-2-3-4-5-6), the compromised switch S2 does not forward the packet to IDS. But S2 locally tags it with IDS, which makes subsequent S3 regard the packet as inspected by IDS. The actually enforced policy chain thus becomes S1-S2-S3 (i.e., packet instances of 1-2-3′-5-6), with IDS bypassed.

specified or implied as necessary in existing compromised-switch detection solutions [17], [20]–[23]. Compromised switches may bypass middleboxes in the way of a coward attack [28], that is, launching an attack only when they cannot be caught by, for example, probing packets and statistics analysis. Under the control of an attacker, compromised switches may adopt various strategies to infer the countermeasure principle. They may also collude to increase the probability of evasion. For ease of discussion, we assume that compromised switches misbehave only through bypassing middleboxes. Existing solutions [17], [20]–[23] can be adopted to combat other switch misbehaviors such as injection and drop. We consider our work complementary to these solutions; together they secure network policy enforcement.

Assumptions. Although switches may be compromised, we assume that the controller and middleboxes are trusted. A trusted controller is critical for correct functioning of the network [17]. Trusted middleboxes are also crucial as they play decisive roles for network performance (e.g., load balancer and proxy) and security (e.g., Firewall and IDS) [3]. Therefore, controllers and middleboxes are more complex and resource-sufficient by design. They are thus more likely to afford expensive security mechanisms than are switches with merely forwarding functions. For ease of understanding, we assume a network with a single ingress switch and a single egress switch. It is straightforward to apply our definition and solution to multi-ingress/multi-egress–switch scenarios.

C. Solution Goals and Challenges

We expect that a middlebox-bypass attack countermeasure fulfills the following performance goals regarding granularity, timeliness, efficiency, effectiveness, robustness, and applicability. These goals face respective challenges, which can hardly be jointly addressed by merely adapting existing malicious-switch countermeasures.

Granularity: Per-packet detection. It is most likely that middlebox-bypass attacks aim to evade security middleboxes. Even one evaded attacking packet may cause various security and privacy issues to the target end host. We expect a solution to detect the middlebox-bypass attack at a packet level.

Timeliness: Real-time detection. Given the potential severity of attacking packets, we also expect to throttle them before they exit the network. Per-packet real-time detection, however, cannot simply report the processing result of every packet to the controller [29]. Massive traffic generated by such report fatigues the control channel and thus degenerates network performance like lower throughput and longer delay [24].

TABLE II
COMPARISON OF FLOWCLOAK WITH STRAWMAN SOLUTIONS BASED ON EXISTING MALICIOUS-SWITCH COUNTERMEASURES.

Legend: PPD: Per-packet detection; RTD: Real-time detection; DPD: Data-plane detection; BD: Beyond detection; CAR: Coward-attack resistance; IC: Infrastructure computability;						
Solution	PPD	RTD	DPD	BD	CAR	IC
probe [20], [21]	✓	✓	✗	✗	✗	✓
statistics [17], [21]–[23]	✗	✗	✗	✗	✗	✓
path verification [30]	✓	✓	✗	✓	✓	✗
FlowCloak	✓	✓	✓	✓	✓	✓

Efficiency: Data-plane detection. Toward efficient detection without impacting other performance measures, we strive to constrain detection to the data plane. Instead of pushing packets' processing results [20], [21] or statistics [17], [21]–[23] to the centralized controller, distributed data-plane devices like middleboxes and switches should locally check whether a packet is middlebox-verified. A challenge arises here is that switches are resource constrained and even compromised.

Effectiveness: Beyond detection. Responses to detected attacks usually lead to performance oscillations. For example, after detecting a middlebox-bypass attack, the corresponding data-plane device must report it to the controller. The controller may configure a new flow path for the attacked flow and reuse the original path when the compromised switch is fixed. To accomplish this, certain actions are required from both control and data planes. To minimize the number of such actions, we expect to actively reduce the probability of a compromised switch inferring the correct tags used by the target middlebox. This challenge is not handled by existing detection-based malicious-switch countermeasures.

Robustness: Coward-attack resistance. This would be a side product of per-packet detection. It is not guaranteed by probe [20], [21] or statistics [17], [21]–[23] based solutions.

Applicability: Infrastructure compatibility. From a functional perspective, path verification schemes [30] would be feasible for middlebox-bypass attack detection. Such schemes let each forwarding device append a cryptographic footprint to the header of each passing packet. The cryptographic footprint is generated using a key shared between the controller and device. The egress switch will report packets to the controller for verification. Specifically, the controller knows the reported packet's expected path and keys of enroute devices. The controller can then generate the expected cryptographic footprint and compare it with that of the reported packet. Although promising a per-packet detection, such schemes rely on the centralized controller for verification. Furthermore, they require cryptographic computation, which is not supported by current switches or even middleboxes yet.

As demonstrated in Table II, FlowCloak as presented in this paper is the first protocol against middlebox-bypass attacks with all the preceding goals and challenges addressed.

III. OVERVIEW

In this section, we present FlowCloak to achieve per-packet real-time detection of middlebox-bypass attacks. It enables a middlebox to mark packets with packet-specific tags that are random to an attacker yet deterministic to a subsequent middlebox or the egress switch, which verifies the tags. FlowCloak functions beyond simple detection because the random tagging scheme lowers the probability of a successful attack. As with prior solutions for SDN-based middlebox policy enforcement [11], FlowCloak requires minor software modification of middleboxes but no modification of switches.

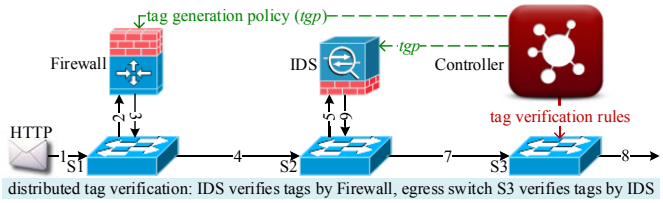

Fig. 3. FlowCloak architecture toward per-packet real-time detection of middlebox-bypass attacks. Besides deterministic tags for tracking processing states, middleboxes add probabilistic tags in packet headers for the subsequent middlebox or egress switch to verify packets.

A. Methodology

FlowCloak defeats the middlebox-bypass attack by randomizing tags for different packets in the same flow. To this end, besides the conventional deterministic tag (*dtag*) for tracking a packet's processing states [11], middleboxes introduce an additional packet-specific probabilistic tag (*ptag*) in the packet header[1]. Although *dtags* are known to middlebox-adjacent switches, FlowCloak synchronizes *ptags* from a middlebox with a downstream verification device (i.e., another middlebox or the egress switch). The verification device triggers an alarm whenever it receives a packet with a different tag from the synchronized one. A satisfactory tagging scheme not only lowers attack probability but also raises detection probability. This way, FlowCloak makes a middlebox's connecting switch harder to infer the correct tag for a packet and thus prevents middlebox-bypass attacks. In other words, FlowCloak is not simply a passive detection solution that endures likely persistent attacks. It actively interferes with the attacking process to discourage the attacker as much as possible.

B. Architecture

FlowCloak enforces a distributed packet verification architecture based on intrinsically secure and readily available elements. As shown in Fig. 3, FlowCloak requires each verification device to verify only its previous hop middlebox. Take the illustrated policy chain S1-Firewall-S1-S2-IDS-S2-S3 as an example. IDS verifies Firewall-tagged packets while egress switch S3 verifies IDS-tagged packets. Such a step-wise verification may make the last middlebox in a policy chain the weakest point. In the preceding example policy chain, as long as an attacker can successfully bypass IDS, it breaches the entire security policy no matter whether it can bypass Firewall or not. The attacker, however, needs the knowledge of network-wide rules to identify last-hop middleboxes. We consider this beyond a practical attacking capability because rule update is via secure controller-switch communication and

[1]Based on the processing result, a middlebox may assign different *dtags* to a packet [11]. Fayazbakhsh *et al.* propose reusing a *dtag* when, for example, corresponding flow expires or corresponding policies take non-joint paths [11]. For ease of presentation, we thus assume only one *dtag* per middlebox.

topological information alone is not sufficient for identifying last-hop middleboxes. As shown in Fig. 1, Proxy is located next to the ingress switch but serves as the last-hop middlebox of the policy chain Firewall-IDS-Proxy.

The controller coordinates tag generation and verification across middleboxes and the egress switch. For each middlebox, the controller provides it with tag generation policies. How a middlebox generates tags should take into account the computing capability of its downstream verification device. If it is another middlebox, tag generation can freely choose feasible computation over packet headers as long as the result fits the size of unused header bits. To verify a packet, the downstream middlebox follows the same tag computation and verifies whether the result matches the tag carried in the packet header. However, it is more challenging if the downstream verification device is the egress switch. Unlike middleboxes, switches cannot compute tags over packet headers upon line speed forwarding. They simply follow rules to match against packet headers. Therefore, we generate tags for the egress switch to verify by a secret mapping from certain header bits.

The preceding mapping is accordingly implemented through tag verification rules, which are populated to the egress switch by the controller. Consider, for example, when we simply use the first bit of a packet header as the probabilistic tag. Then corresponding verification rules can be configured as [Priority: high; Matching: FirstBit = 0, *ptag* = 0; Action: pass], [Priority: high; Matching: FirstBit = 1, *ptag* = 1; Action: pass], and [Priority: low; Matching: FirstBit = *, *ptag* = *; Action: alarm]. With the single-bit probabilistic tag, we expect that an attacker can forge a correct tag with a probability of $\frac{1}{2}$. If the attacker knows the mapping principle, it can deduce which bit is in use and how it is mapped by statistical inference. For example, given two packets with the same probabilistic tag, bits with the same index but different values can be filtered. The attacker may quickly pinpoint the bit for mapping the probabilistic tag. Therefore, the mapping strategy should be sufficiently complex to combat statistical inference. Meanwhile, it should be practically efficient in that the number of tag verification rules fits TCAM capacity on the egress switch.

We propose a multi-tag verification technique to address the tradeoff between inference resistance and TCAM usage. Section V details this technique alongside the design of tag generation and verification between a middlebox and the egress switch. Prior to that, Section IV first presents the design of tag generation and verification between two middleboxes.

IV. Design: Middlebox versus Middlebox

In this section, we present the packet processing logic of FlowCloak-enhanced middleboxes. As discussed in Section III-B, tag generation and verification are highly coupled. Both of them depend on whether the operation is between middleboxes or between a middlebox and the egress switch. This section focuses on between-middlebox operations based on hashing.

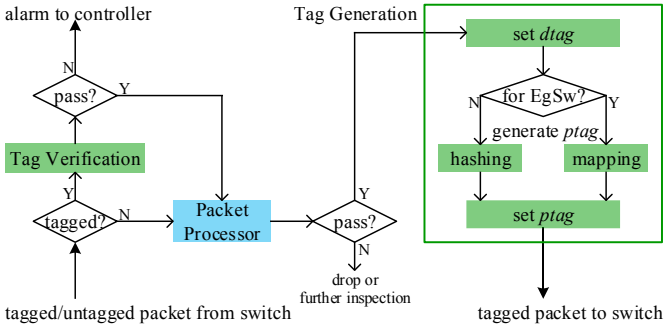

Fig. 4. FlowCloak-enhanced middlebox's packet processing logic.

A. Middlebox Packet Processing Logic

FlowCloak augments a middlebox with two major components, that is, tag generation and tag verification. As shown in Fig. 4, only middlebox-processed (i.e., tagged) packets need to undergo tag verification. To differentiate tagged packets from untagged ones, we can leverage the property that unused bits are set to zero by default. Untagged packets are directly fed to the packet processor (i.e., the original middlebox function) while tagged packets are treated so only if they pass tag verification. (A potential attack to bypass a middlebox by setting $dtag$ as zeros is addressed in Section IV-C.) We steer packets to the packet processor prior to tag generation because some middleboxes like NAT may modify packet headers. If we direct packets to tag generation first, the downstream verification device may get a different packet header than that used for tag generation and thus fails tag verification.

B. Tag Generation

A FlowCloak tag consists of a deterministic tag (*dtag*) and a probabilistic tag (*ptag*). The *dtag* is pre-assigned by the controller and is unique across middleboxes for tracking packet processing states [11]. It should be non-zero for differentiating tagged and untagged packets. The *ptag* is computed by hashing if the downstream detection device is a middlebox or is chosen by mapping if the downstream detection device is the egress switch. To enable a middlebox to differentiate the two cases, the controller preloads to it a map of its downstream detection devices and corresponding flows. In this section, we focus on the middlebox case. Since we first tag a packet with the *dtag* that varies across middleboxes (Fig. 4), we avoid generating *ptags* over the same packet header at different middleboxes. We thus can use identical hashing parameters on middleboxes for ease of configuration. It is straightforward to tailor these parameters for middleboxes toward higher robustness; we omit investigating such enhancement hereafter. Formally speaking, let $PktHdr^{dtag}$ denote the packet header with $dtag$ embedded and $\mathrm{Sample}(PktHdr^{dtag})$ the sampled bits therein. A middlebox generates $ptag$ for another middlebox to verify as:

$$ptag = \mathrm{Hash}(\mathrm{Sample}(PktHdr^{dtag})), \quad (1)$$

where $\mathrm{Hash}(\cdot)$ is the adopted hash function that hashes an input to an output uniformly at random across the output space and is hard to reverse engineer using input-output pairs.

C. Tag Verification

On-middlebox verification passes a packet if it satisfies two conditions. First, the packet comes from the correct previous-hop middlebox or it has not previously visited a middlebox. This should be enforced in the "tagged?" component in Fig. 4. Second, if the packet comes from a middlebox, the packet should carry the correct probabilistic tag generated by that middlebox.

The first condition is necessary against two types of indirect middlebox-bypass attacks. Take the policy chain S1-S2-Firewall-S2-S3-IPS-S3-S4-Proxy-S4-S5 as an example. In the first attack, compromised S2 may make a packet bypass Firewall and set its $dtag$ to zeros without forging $ptag$. Without precaution, the next-hop middlebox IPS regards the packet unnecessary to verify as if it were the first encountering middlebox and proceeds to subsequent processing. In the second attack, compromised S3 may relay a packet tagged by Firewall to S4 and then Proxy while bypassing IPS. If Proxy simply verifies the packet's $ptag$ as if it were generated by Firewall, the verification will succeed and so will the attack. To enable a middlebox to detect the preceding two attacks, the controller preloads to it a map of its previous-hop middleboxes and corresponding flows. Mismatch of the $dtag$ and that of the corresponding previous-hop middlebox fails tag verification and triggers an alarm.

After the first condition is verified, the middlebox proceeds to verify the second condition, that is, the correctness of $ptag$. Following the synchronized tag generation policy, the middlebox first generates a tag $ptag'$ by Equation 1. If $ptag'$ is equal to the $ptag$ carried in the packet header, tag verification succeeds and the middlebox directs the packet to subsequent processing components (e.g., Packet Processor and Tag Generation in Fig. 4). Otherwise, tag verification fails and the middlebox triggers an alarm.

V. Design: Middlebox versus Egress Switch

In this section, we present tag generation and verification design between a middlebox and the egress switch. Two challenges arise from the egress switch's capacity constraint regarding computation and storage. Since SDN switches support only simple matching rather than more complex computation on packet headers, we adopt a mapping scheme that maps certain bits in a packet header to a probabilistic tag. The map is preloaded on middleboxes and emulated via tag verification rules on the egress switch. As discussed in Section III-B, more verification rules enhance inference resistance but aggravate TCAM scarcity. We propose a multi-tag verification technique to guarantee inference resistance with a practically affordable number of rules. This technique enables the egress switch to leverage pipelined flow tables, which has been supported since OpenFlow Switch Specification Version 1.1.0 in 2011 [26].

A. Motivation: Multi-tag Verification

The multi-tag verification technique addresses the tradeoff between inference resistance and TCAM usage by breaking tag verification rules into a pipeline of flow tables. A major benefit

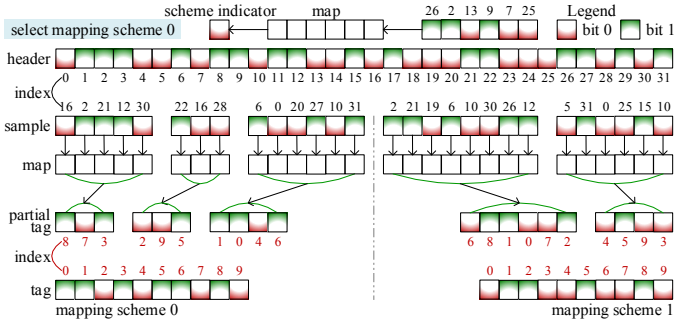

Fig. 5. Multi-tag generation using preloaded maps on a middlebox.

of this technique is drastically shrinking the number of rules. Formally speaking, take a number of resulting flow tables T_i, each with a number $|T_i|$ of tag verification rules. The multi-tag verification technique can decrease rule count from $\mathcal{O}(\Pi|T_i|)$ to $\mathcal{O}(\Sigma|T_i|)$. Consider an example with $2^{10} = 1{,}024$ rules for mapping 10 bits in a packet header to a 10-bit probabilistic tag. If we divide the mapping process to two 5-to-5 mapping subprocesses, we need only two $2^5 = 32$-rule flow tables. Then the total number of rules drops to $32+32 = 64$ instead of much larger $32 \times 32 = 1{,}024$. Another benefit is that pipelined flow tables enable variable rule lengths [27]. The preceding two 5-to-5 mapping flow tables cost $(5+5) \times 32 \times 2 = 640$ bits of matching fields; this approximates the size of only a number $\frac{640}{288} \approx 3$ of 288-bit 10-tuple OpenFlow rules [24]. Without the multi-tag verification technique, the 10-to-10 mapping flow table costs $(10+10) \times 1024 = 20{,}480$ bits, which approximate the size of $\frac{20480}{288} \approx 72$ OpenFlow rules. Furthermore, if we simply place the 1,024 rules in the original flow table, they would occupy the space of 1,024 OpenFlow rules. Our design therefore promises an affordable overhead of TCAM usage.

B. Tag Generation on Middlebox

Design. Preloaded with multiple mapping schemes, a middlebox generates a probabilistic tag $ptag$ via multiple partial tags. For each partial tag, we choose non-consecutive and shuffled header bits and map them to non-consecutive and shuffled bits in $ptag$. As shown in Fig. 5, we first decide which mapping scheme to use by an indicator generated by chosen bits in the packet header. An x-bit indicator supports switching among 2^x mapping schemes. We compute an x-bit indicator using sliding-window XOR over x or more chosen header bits. By sliding-window XOR, the window size is x bits and the sliding size is one bit. In the illustrated example, we generate a 10-bit $ptag$ out of a 32-bit toy packet header. Since two mapping schemes are given, we need only a single-bit indicator for deciding which scheme to use. Six bits are used for computing the indicator using sliding-window XOR. The indicator result is 0; so mapping scheme 0 is selected. Under mapping scheme 0, the middlebox generates the 10-bit $ptag$ by mapping 5, 3, and 6 bits to three partial tags of 3, 3, and 4 bits, respectively. The middlebox then merges these partial tags to compose $ptag$.

Speed. We achieve fast tag generation by generating partial tags in parallel and enabling fast searching over maps. First, once a mapping scheme is selected, generating partial tags

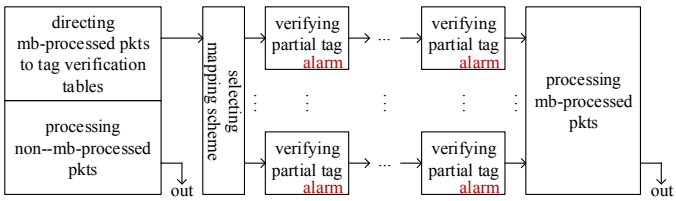

Fig. 6. FlowCloak's flow table pipeline logic on the egress switch.

based on different maps can run in parallel. This upper bounds the generation time by $\mathcal{O}(\max(t_i))$ instead of $\mathcal{O}(\Sigma t_i)$, where t_i denotes the partial-tag generation time over the i-th map. Second, the partial-tag generation process over a map resembles a search process. Consider for example a map with m-bit inputs. If we simply store the map as an array, then each round of the search process compares two m-bit strings; it invokes $\mathcal{O}(2^m)$ rounds of such comparison in the worst case. To minimize search time, we can store the map as an m-level tree. The internal nodes correspond to bits in each entry of the map; the leaf nodes correspond to partial tags. Based on such a tree, partial-tag generation requires a constant number of $\mathcal{O}(m)$ rounds of single-bit comparison. Another leap of search speed takes place when the number of entries is small. Consider when $m = 5$ for example. Now we have $2^5 = 32$ entries. We can sort the entries in an ascending order of the 5-bit inputs and store the map as an array with each item indexed by a 5-bit input and assigned as the corresponding partial tag. In this case, we enjoy a constant $\mathcal{O}(1)$ search speed.

Robustness. Our tag generation mitigates statistical inference in four directions. ❶ Using more input bits than tag bits leads to intrinsic collisions where two different sets of input bits map to the same tag. Given two different packet headers with the same tag, the attacker can hardly know whether it is due to intrinsic collision or due to the set of chosen bits are with the same values in both packet headers. This makes it pointless for the attacker to simply filter different-value bits. ❷ Several possible mapping schemes to use make same-value bits map to different tags. Moreover, as we use XOR to compute the indicator, even one flipping bit in the ones for computation might generate a different indicator. Therefore, it is also pointless for the attacker to simply filter same-value bits upon different tags. ❸ The attacker may not know whether a middlebox's next hop is another middlebox or the egress switch (Section III-B). This makes inference even more infeasible. ❹ Given that thousands of rule updates per second are supported by current switches [31], we can periodically refresh the mapping schemes on middleboxes and tag verification rules on the egress switch. Different mapping schemes make the same packet header map to different tags. This significantly confuses the attacker.

C. Tag Verification on Egress Switch

Our multi-tag verification technique enables the egress switch to leverage a flow table pipeline to save TCAM space. Since traversing multiple flow tables introduces delay, a goal for configuring the pipeline is that such delay impacts only middlebox-processed packets. In other words, we expect that non–middlebox-processed packets still go through rule-matching only once. To achieve this goal, we lead the pipeline with the original flow table. As shown in Fig. 6, we assign rules matching expectant middlebox-processed packets with higher priorities than we assign to rules matching non–middlebox-processed packets. For middlebox related rules, we modify their actions to direct the matching packets to the next flow table (i.e., Goto-Table instruction [26]).

For packets arriving at the second table, tag verification is enforced via a pipeline of flow tables. As Fig. 6 shows, the second table selects the mapping scheme. Given, for example, two mapping schemes in use and six bits for computing the indicator (Fig. 5), the second table contains 2^6 rules. All their matching fields enumerate all instances of a six-bit string. For each rule, if the XOR result of the six bits is zero, then its action is go to the table corresponding to mapping scheme 0. Otherwise, the action is go to the table corresponding to mapping scheme 1. Each mapping scheme consists of a pipeline of flow tables, with each verifies a partial tag. Consider such a flow table that verifies p-bit partial tags mapped from h header bits. It contains 2^h rules. The format of a verification rule therein is as follows.

- The matching field is an instance of the h-bit string and the corresponding p-bit partial tag.
- The action is go to the next table for verifying another partial tag.

Only when a packet matches with a verification rule can it pass verification. For non-matching packets, we use an all-wildcard rule to trigger an alarm to the controller. A packet passes tag verification only if it passes all partial-tag verification.

After tag verification, packets enter the ending flow table that contain rules for processing verified middlebox-processed packets. It should be a subset of the original flow table. We extend their action fields such that the egress switch zeros all tag fields before it forwards the packets.

VI. Implementation

We implement FlowCloak using Snort [32], the most widely deployed IDS/IPS with over 500,000 registered users. As a security middlebox, Snort detects emerging threats based on various techniques (e.g., packet logging and real-time traffic analysis) and filters attacking traffic. Given that Fayazbakhshthe *et al.* [11] have already demonstrated the feasibility and limited overhead of modifying middleboxes to support packet tagging, we adopt only Snort as a use case and focus more on how to implement our tagging scheme that can combat the newly identified middlebox-bypass attack.

Our modification empowers Snort to generate and verify probabilistic tags (Fig. 4). Both generation and verification functions are called in the file /src/snort.c. The generation function is called at the beginning of ProcessPacket$(\cdot)$, by which Snort starts processing an incoming packet. The verification function is called at the end of PacketCallback$(\cdot)$, by which Snort directs the processed packet to subsequent components. The implementation of the generation and verification functions mainly include hashing, bit-wise operation,

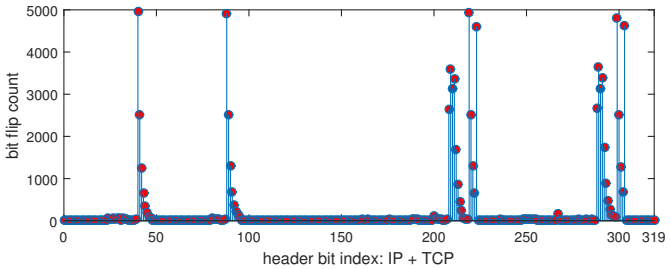

Fig. 7. Header dynamics over 5,000 locally sniffed co-flow packets.

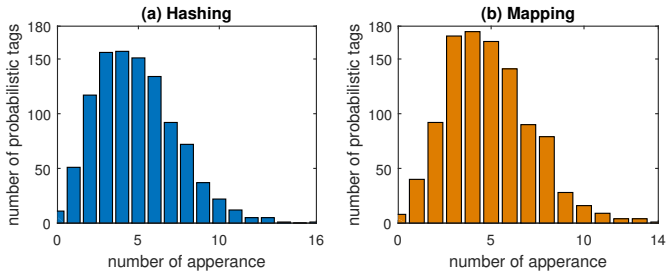

Fig. 8. Frequency distribution of probabilistic tags generated by (a) hashing and (b) mapping over 5,000 locally sniffed co-flow packets.

and map searching over packet headers. We adopt the FNV-1a hash function [33] because it well randomizes the hash results of similar inputs. We encode preloaded maps in a way that an $\mathcal{O}(1)$ search complexity is achieved (Section V-B). Our modification accounts for 387 lines of C code while the original code base of Snort contains over 293,000 lines. Besides, a 250-line library from [11] is used for supporting controller-middlebox communication.

We build FlowCloak using ODL Carbon as the controller, OVS v2.5.3 (with long-term support) as switches, and Snort 2.9.9.0 [32] as middleboxes. We interconnect the preceding elements into a network using Mininet 2.2.2.

VII. Evaluation

In this section, we evaluate the performance and robustness of FlowCloak. We first conduct a trace-driven measurement of packet-header dynamics. The results show that sufficient frequently-changed bits in the headers of co-flow packets can be sampled to generate probabilistic tags. The generated probabilistic tags follow a uniform distribution and leave an attacker with no advantage over random guessing. Limiting the success rate of random guessing under 0.1%, a 10-bit probabilistic tag requires memory overhead of only dozens of rules on the egress switch. Limited packet processing delay is imposed by FlowCloak on middleboxes and the egress switch.

Experiment setup. Our FlowCloak prototype runs on a Dell PowerEdge R730 server with 30 MB cache, 24 2.3-GHz Intel(R) Xeon(R) CPUs (E5-2670 v3), and 128 GB memory. Since we use only Snort as a use case (Section VI), we create multiple Snort instances to emulate a multi-middlebox policy. Each Snort instance runs in a virtual machine and allocated with 8 GB memory and 2 CPUs.

A. Feasibility: Packet Header Dynamics

The dynamics of header fields/bits are key to randomize probabilistic tags, especially for those to be verified by the egress switch. We conduct a trace-driven measurement using packets with various transport protocols. The measuring methodology counts packet-wise bit flips for each co-flow packet header. We initialize the count of a bit as zero; we increment it by one if the bit flips in two consecutive packets and keep it intact otherwise. Due to space constraint, we report only TCP results because TCP represents 90% of fixed Internet traffic and 96% for mobile Internet traffic [34].

Fig. 7 reports header dynamics over locally sniffed 5,000 co-flow packets. We concentrate on IP and TCP headers, which are matched by most SDN rules. 53.8% of the 320 bits demonstrates bit flips among consecutive packets. 6 bits change more than 4,600 times while the most frequent one hits 4,957. 18 bits change over 50% of the packet-wise comparisons. 24 bits change more that 1,000 times; they represent part of Identifier and Header Checksum in IP header and part of Sequence Number and Checksum in TCP header. Given that less varying bits and even static bits (e.g., source/destination IP addresses) can also be useful (Section V), we have sufficient choices of header bits to map them to, say, 10-bit probabilistic tags.

B. Efficacy and Robustness: Probabilistic Tag Distribution

While eliminating the concern of various inference attacks in Section V-B, we further investigate the distribution of probabilistic tags to make sure that an attacker can obtain no advantage over random guessing. If the distribution is biased toward certain tags, the attacker may use such tags to increase success rate. Our results show that probabilistic tags generated by FlowCloak resemble a uniform distribution, which well confines the attacker to random guessing.

Fig. 8 reports the distribution of probabilistic tags generated by (a) hashing and (b) mapping. Hashing-based tags are for middleboxes to verify while mapping-based tags are for the egress switch to verify. Given a 10-bit probabilistic tag in use and 5,000 co-flow packets to consider, we expect that the average number for a tag to appear is $\frac{5000}{2^{10}} = 4.9$. Satisfying this expectation, both distributions peak when the number of appearance is 4. For the mapping scheme, we test different settings and the results are consistent.

Another sweet spot of robustness is that FlowCloak does not passively endure the attacker to keep guessing and issuing incorrect tags. It triggers an alarm whenever an incorrectly-tagged packet is detected. Furthermore, refreshing mapping schemes helps nullify the attacker's inference efforts.

C. Overhead: Memory and Latency

TCAM overhead by tag verification rules on the egress switch. We conduct extensive experiments on generating 10-bit probabilistic tags using various mapping schemes. Dozens of tag verification rules suffice to emulate all test mapping schemes and satisfy uniform tag distribution. For example, when we map 20 bits to a 10-bit tag. We can use a pipeline of 5 flow tables, each for a 4-to-2 mapping. Then the number of rules is $2^4 \times 5 = 80$.

Latency by tag generation/verification on middlebox. FlowCloak-enhanced Snort performs additional tag generation/verification over packet headers. Fig. 9(a) compares its packet processing latency with that of the original Snort. The original Snort takes 4.6 ms in average to process a packet

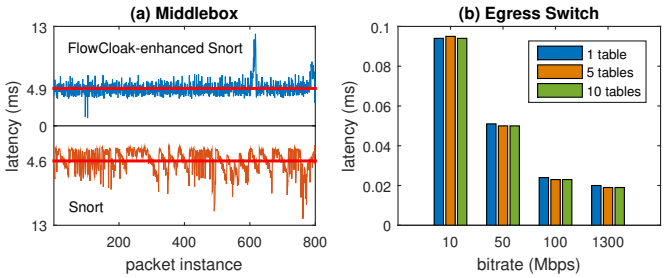

Fig. 9. Packet processing delay for (a) tag generation/verification on a middlebox and (b) tag verification on the egress switch.

while FlowCloak-enhanced Snort 4.9 ms. A delay of as low as 0.3 ms demonstrates FlowCloak's high efficiency.

Latency by tag verification on egress switch. Middlebox-processed packets go through a pipeline of flow tables for verification on the egress switch. We compare in Fig. 9(b) the packet processing latency using different numbers of flow tables. To eliminate the impact of other network elements, we create a network with only one switch (as both ingress and egress) and two end hosts. The only switch is configured with multiple tables containing tag verification rules. We craft test (1,500-byte TCP) packets in advance by feeding them into Snort for tagging. Then we send the tagged packets via one end host at different bitrates. To our surprise, no obvious delay is observed upon the use of pipeline processing. We consider this reasonable because traversing several flow tables requires only several more map searches of high speed.

VIII. Conclusion

We have presented our newly identified middlebox-bypass attack to breach SDN-based middlebox policy enforcement and proposed FlowCloak solution. FlowCloak enables middleboxes to mark packet headers with tags probabilistically unknown to an attacker. The tagging strategy, however, is synchronized across middleboxes and the egress switch for packet verification. If an attacker manipulates a compromised switch to bypass attacking traffic from the attached middlebox, it has to forge correct tags to evade detection. FlowCloak confines the attacker to only random guessing. We implement FlowCloak using Snort. Experiment results show that FlowCloak can achieve accurate detection with negligible overhead.

Acknowledgment

This work is supported in part by the National Key R&D Program of China under Grant No. 2017YFB0801703, the National Science Foundation of China under Grant No. 61402404, 61772559, and 61602093, and U.S. National Science Foundation under Grant No. CCF 1320044 and CNS-1513719. We also thank Avery Laird and INFOCOM 2018 Chairs and Reviewers for their helpful feedback.

References

[1] M. Walfish, J. Stribling, M. N. Krohn, H. Balakrishnan, R. Morris, and S. Shenker, "Middleboxes no longer considered harmful." in *OSDI*, 2004.

[2] J. Sherry and S. Ratnasamy, "A survey of enterprise middlebox deployments," UC Berkeley., Tech. Rep. UCB/EECS-2012-24, 2012.

[3] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *SIGCOMM*, 2012.

[4] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *SIGCOMM*, 2014, pp. 163–174.

[5] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: securely outsourcing middleboxes to the cloud," in *NSDI*, 2016.

[6] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *SIGCOMM*, 2015.

[7] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *NSDI*, 2012.

[8] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *SOSP*, 2015.

[9] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *SIGCOMM*, 2016, pp. 511–524.

[10] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *SIGCOMM*, 2013.

[11] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *NSDI*, 2014.

[12] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan *et al.*, "Steering: A software-defined networking for inline service chaining," in *ICNP*, 2013, pp. 1–10.

[13] SDN switches aren't hard to compromise, researcher says,. [Online]. Available: https://tinyurl.com/ydy5xjl4

[14] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an sdn with a compromised openflow switch," in *NordSec*, 2014, pp. 229–244.

[15] SDN & Security: Why Take Over the Hosts When You Can Take Over the Network. [Online]. Available: https://tinyurl.com/ybfwlsrp

[16] IBM security services 2015 cyber security intelligence index. [Online]. Available: https://tinyurl.com/ydglh5z5

[17] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks," in *NDSS*, 2015.

[18] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "Pga: Using graphs to express and automatically reconcile network policies," *SIGCOMM*, 2015.

[19] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," in *INFOCOM*, 2016.

[20] Y.-C. Chiu and P.-C. Lin, "Rapid detection of disobedient forwarding on compromised openflow switches," in *ICNC*, 2017, pp. 672–677.

[21] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *NetSoft*, 2015, pp. 1–6.

[22] A. Kamisiński and C. Fung, "Flowmon: Detecting malicious switches in software-defined networks," in *SafeConfig*, 2015, pp. 39–45.

[23] C. Pang, Y. Jiang, and Q. Li, "Fade: Detecting forwarding anomaly in software-defined networks," in *ICC*, 2016, pp. 1–6.

[24] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *SIGCOMM*, 2011, pp. 254–265.

[25] I. N. Bozkurt, A. Aguirre, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. Maggs, and A. Singla, "Why is the internet so slow?!" in *PAM*, 2017, pp. 173–187.

[26] OpenFlow Switch Specification Version 1.1.0 Implemented (Wire Protocol 0x02). [Online]. Available: https://tinyurl.com/qbe8ekp

[27] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane specialization for high-performance openflow software switching," in *SIGCOMM*, 2016.

[28] B. Liu, J. T. Chiang, J. J. Haas, and Y.-C. Hu, "Coward attacks in vehicular networks," *MC2R*, vol. 14, no. 3, pp. 34–36, 2010.

[29] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI*, 2014, pp. 71–85.

[30] P. Zhang, "Towards rule enforcement verification for software defined networks," in *INFOCOM*, 2017.

[31] M. Kuzniar, P. Peresini, and D. Kostic, "What you need to know about sdn flow tables," in *PAM*, 2015.

[32] Snort. [Online]. Available: http://www.snort.org/

[33] Fowerler-Noll-Vo hash function. [Online]. Available: https://tinyurl.com/qxcqjhk

[34] TCP Optimization: Opportunities, KPIs, and Considerations. [Online]. Available: http://tinyurl.com/ybahn3uy