PSI: Precise Security Instrumentation for Enterprise Networks

Tianlong Yu†, Seyed K. Fayaz†, Michael Collins♭, Vyas Sekar†, Srinivasan Seshan†
†Carnegie Mellon University, ♭RedJack

***Abstract*—Despite soaring investments in IT infrastructure, the state of operational network security continues to be abysmal. We argue that this is because existing enterprise security approaches fundamentally lack precision in one or more dimensions: (1) *isolation* to ensure that the enforcement mechanism does not induce interference across different principals; (2) *context* to customize policies for different devices; and (3) *agility* to rapidly change the security posture in response to events. To address these shortcomings, we present PSI, a new enterprise network security architecture that addresses these pain points. PSI enables fine-grained and dynamic security postures for different network devices. These are implemented in isolated enclaves and thus provides precise instrumentation on these above dimensions by construction. To this end, PSI leverages recent advances in software-defined networking (SDN) and network functions virtualization (NFV). We design expressive policy abstractions and scalable orchestration mechanisms to implement the security postures. We implement PSI using an industry-grade SDN controller (OpenDaylight) and integrate several commonly used enforcement tools (e.g., Snort, Bro, Squid). We show that PSI is scalable and is an enabler for new detection and prevention capabilities that would be difficult to realize with existing solutions.**

I. Introduction

Despite dramatic escalation in cost (e.g., 7.3 billion dollars/year for the US Government [47]) the state of operational network security is still abysmal. We continue to hear about high-profile breaches and failures of existing network security infrastructures [33], [37], [56]. In many ways, these indicate the collective failure of traditional network security approaches for enterprises including perimeter-defenses [65], distributed firewalls [41], Security Information and Event Management (SIEM) systems [21], network management products [13], [71], among others.

As a well-known fact in the operational security community [1], [26], current solutions do not and cannot effectively implement *precise* defenses along three key dimensions: *isolation, context,* and *agility* (§II):

- **Isolation:** First, a defense system must ensure that security policies do not interfere with each other and cause collateral damage. Due to cost and network management limitations, existing approaches enforce policies at topological "choke points" [41]. This induces both logical and performance interference. For example, reconfiguring a firewall to block a specific user can unintentionally block others (logical interference), or processing traffic may overload the firewall, cause dropped packets (performance interference) and may lead to the shutting down of advanced security functions [16].
- **Context:** Second, a defense system must be able to enforce customized policies for individual network devices based on the context — all the security-related device attributes and states. For example, a firewall protecting a database should allow a http server with `OpenSSL 1.0.1g` to access the database (heartbleed patched), but should deny the access of the other http server with `OpenSSL 1.0.1f` (vulnerable to heartbleed), even if both servers are exposing the same IP and ports to the firewall (via NAT). The state of the practice relies on traffic attributes such as IP addresses or netblocks. Unfortunately, this induces significant "blind spots" as the relevant contextual attributes may be obscured due to topological artifacts; e.g., the device origin may be hidden behind a NAT [36].
- **Agility:** Finally, a defense system must be able to change policy at fine-grained timescales. We know that attackers dynamically alter their strategies (*e.g.*, moving laterally inside the perimeter, switching to different exploit kits at different stage [55]). Ideally, we should be able to dynamically change our posture when specific internal hosts appear to be engaging in suspicious activities. Unfortunately, today's mechanisms are derived from static abstractions (e.g., ACLs, signatures) and cannot express and implement such dynamic capabilities.

To address these limitations, let us consider a hypothetical design point as shown in Figure 1 where we can: (1) physically and logically isolate the processing applied to the traffic to/from one device from traffic to/from other devices; (2) ensure fine-grained customization based on the relevant context (*e.g.*, this traffic is from a device with heartbleed vulnerability); and (3) dynamically instantiate the necessary security processing (*e.g.*, if a BYOD device suddenly tries to connect to an irrelevant internal server (suspicious), subject it to deep inspection or quarantine it). By construction, this design addresses the above limitations—it has deep context into every packet, the processing can be dynamically adjusted, and it guarantees zero interference. Based on the current trajectory of enterprise security solutions (*i.e.*, relying on statically configured hardware appliances deployed at designated network chokepoints), however, this design may seem hopelessly elusive in terms of cost, complexity, and deployability.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23200

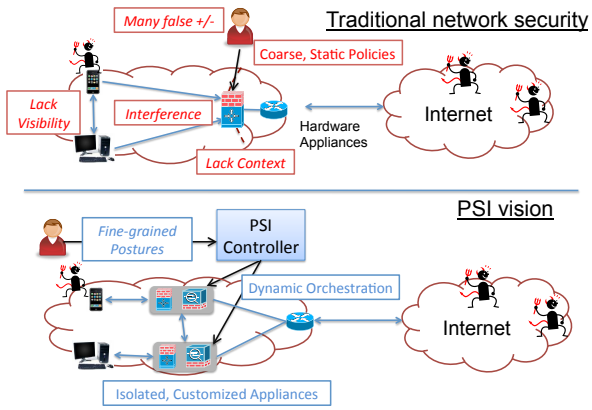


Fig. 1: Contrasting today's approaches vs. PSI

In this paper, we present Precise Security Instrumentation (**PSI**), which serves as a proof-by-construction realization of this hypothetical design (§III). PSI uses recent advances in Network Functions Virtualization (NFV) to launch virtualized security functions (*e.g.*, virtual NIDS) on demand [30] inside an on-premise cluster and SDN capabilities to route the traffic to the desired virtual appliances. Thus, PSI can serve as an enabler for new precise security detection and prevention capabilities that would be exceedingly difficult, if not impossible, with existing mechanisms.

Contributions: Our goal in this paper is to design the technical foundations for PSI, rather than develop new detection and prevention algorithms. To this end, we make three key contributions:

- **Expressive Polices (§IV):** We design a PSI policy abstraction that can express agile and contextual traffic processing. This allows us to express rich multi-stage security-relevant processing mapped to a security-relevant state for each device; *e.g.*, a host in normal state is subject to simple IDS-followed-by-firewall but in "suspicious" state may be subject to additional on-demand exfiltration detection modules. We also provide mechanisms to incorporate legacy policies that need to be applied to a group of devices.
- **Scalability and Responsiveness (§V):** Naively applying SDN/NFV mechanisms in security context is problematic and can introduce new avenues for DoS attacks [63]. We develop a scalable orchestration platform by synthesizing three key ideas: (a) proactive forwarding schemes based on logical tags that do not need to involve the controller; (b) effective techniques for horizontally scaling the controller infrastructure; and (c) prefetching future enforcement states to improve responsiveness.
- **Practical Implementation (§VI–§VII):** We prototype PSI in an industry-grade SDN control platform (`OpenDaylight`) [17]. We extend a range of widely used open source network security tools (*e.g.*, Snort, Squid, iptables) and integrate them within PSI. We show that PSI can coordinate complex policies on networks of up to 100,000 hosts without significant performance overload. Finally, we demonstrate use cases showing how PSI can enable new security capabilities (e.g., IoT patch).

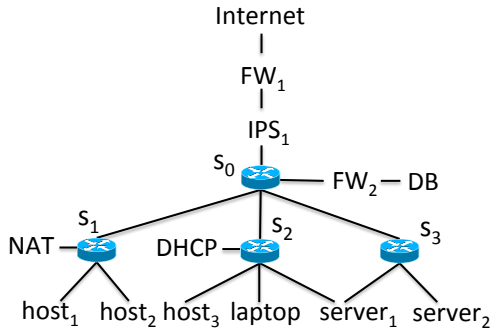


Fig. 2: An example enterprise network from an enterprise intrusion incident [10].

II. Motivation

In this section, we motivate the need for isolation, context, and agility using a simple but realistic enterprise network topology (§II-A). Then, we discuss our threat model, highlighting the attackers' goals, capabilities, and potential strategies in §II-B. Given this setup, §II-C presents attack scenarios that highlight key shortcomings in current mechanisms with respect to isolation, context, and agility.

A. Problem setting

Network Description: Figure 2 depicts a small enterprise network composed of multiple switches[1] and middleboxes (*e.g.*, firewalls, NATs, proxies, IPSes). This network connects multiple *devices* such as hosts, databases and servers. The devices are protected by a collection of security middleboxes (e.g., FW_1, FW_2 and IPS_1). Like most enterprise networks, the depicted network has some more complex subcomponents: a NAT changes IP and port addresses of packets; DHCP dynamically assigns IP to some devices and $server_1$ is connected to both s_2 and s_3 with different IP addresses to support failure recovery or high throughput [20].

Security intent: The *intent* of the operator is to enforce security postures as a function of network devices and their context (device attributes and security states), namely, $intent = function(device, context_{device})$. For example, a device may be a HTTP server, and the context may be it has n-day Heartbleed vulnerability.

B. Threat Model

The goal of attackers is to compromise devices, exfiltrate data, or disrupt services. To achieve these goals, attackers need to evade the detection and mitigation of current defense system. There are four general strategies the attackers can use: being stealthy, being dynamic, causing collateral damage, and overloading the defense:

- **Being stealthy:** Attackers can use "blind spots"—devices or traffic routes not visible to the defense system, to avoid detection. For example, an attacker can exploit a BYOD device and use it to launch internal attacks that can avoid the detection of IPS at the department gateway, as seen in the Shady RAT exploit [10].

[1]We use the terms router and switch interchangeably.

- **Being dynamic:** Attackers can adjust their attack postures to achieve their goals. For instance, we commonly see multi-stage and multi-vector attacks such as using a zero-day attack to bypass the defense and compromise an initial device [5], then using multiple exploit toolkits [11] to compromise others from private network.
- **Causing collateral damage:** Attackers can force the defense system to act on innocent users or not act at all. For example, an attacker behind a NAT can force an IPS to enforce deep packet inspection (DPI) for all traffic at the gateway, which can degrade performance for legitimate traffic [16].
- **Overloading the defense:** Attackers can overload the defense systems or the administrator. For example, an attacker can increase the traffic volume to overload an IPS or use malformed packets to generate a large scale of alerts to overload the alerting systems or the administrator.

Our threat model assumes that the attacker cannot directly compromise the defense system (*e.g.*, infect the IDS or the SIEM system).

C. Motivating Scenarios

Next we walk through several scenarios to highlight how the attackers can evade existing mechanisms such as perimeter defense [24], distributed Firewall/IPS [12], [41], [51], [58], SIEM (Security Information & Event Management) [6], [21], and network segmentation (*e.g.*, vLAN [13]). These scenarios highlight the importance of *isolation*, *context*, and *agility* in mitigating the attacker's strategies above.

Isolation: Our example network (Figure 2) lacks isolation in two ways: *performance interference* and *logical interference*. Attackers can exploit this interference to cause collateral damage. Performance interference results from the need to process traffic through narrow enforcement points. For example, in Figure 2, IPS_1 is shared across all devices' traffic from/to the Internet. Suppose, an attacker is exploiting $server_1$, which has massive inbound/outbound traffic, and the administrator uses a DPI module at IPS_1 to stop the exploit. Here IPS_1 can be overloaded, and the throughput of other devices (e.g., $host_3$) will also decrease. Thus, the attack causes collateral damage to $host_3$. Logical interference results from the fact that common enforcement points often result in policy specifications that are coarse-grained or prone to misconfiguration [75]. For example, suppose an attacker has compromised $host_1$ and use it to access to the database server DB. To stop the attack, the administrator updates the firewall policy at FW_2 to block the IP of $host_1$. Due to the NAT, $host_1$ and $host_2$ are exposing the same IP and the updated policy can unintentionally interfere with $host_2$'s access to DB. This kind of problem is commonly reported on operational forums [3], [49]. In terms of the security intent we defined in §II-A, interference conceptually means that the same device can be affected with conflicting intents.

Current approaches: existing isolation mechanisms are either too coarse-gained or too costly to resolve all the logical/performance interferences. For example, network segmentation mechanisms (*e.g.*vLAN) cannot support fine-grained isolation at a subnet level [73], e.g., $host_1$ and $host_2$ in Figure 2 cannot be isolation as they share the same subnet.

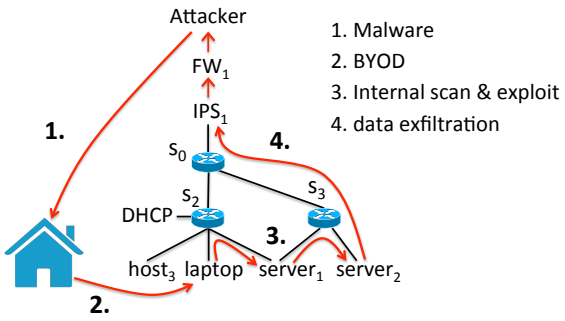


Fig. 3: A typical APT attack: firewall and IPS lacks the context and visibility to prevent internal exploit and data exfiltration.

Then to provide isolation using distributed Firewall/IPS techniques would require costly distributed Firewall/IPS hardware deployed for each device and rewiring the network.

Context: By definition, enforcing security intents as discussed in §II-A requires taking into account the *context* of devices. To illustrate why this is challenging in practice, Figure 3 shows an advanced persistent threat (APT) [10] in the example network. The goal of the attacker is to compromise $server_2$ and exfiltrate sensitive data. To do so, the attacker can use a stealthy strategy: exploiting several *blind spots* in the defense (FW_1, IPS_1 and vLAN at s_2 and s_3). In the first step, the attacker compromises a laptop in a loosely protected home network with malware. Then, the attacker uses the laptop (as a BYOD device) to access other enterprise network devices. Unfortunately, the context that BYOD laptop is accessing $server_1$ is not visible to FW_1 and IPS_1 for enforcing more stringent policies. Next, the attacker compromises $server_1$ from the laptop with internal scan and exploits, uses it to get through the segmented vLANs between s_2 and s_3, and exfiltrate data from $server_1$.

Current approaches: Perimeter defenses (FW_1, IPS_1) lack visibility and context about devices inside the network. Similarly, vLAN are coarse and lack context about $server_1$ across two subnets. Even if we use distributed Firewall/IPS and deploy two IPSes at s_2 and s_3, the fact that $server_1$ is accessed by a BYOD laptop is hidden from IPS at s_2 (could be access from $host_3$?). This is because DHCP may dynamically assigns IPs to laptop and $host_3$, and the IPS cannot distinguish them by IP address (i.e., hidden context); Ideally, we want the enforcement mechanism to be *logically deep* inside the network and have fine-grained visibility into the security relevant *context* of individual devices.

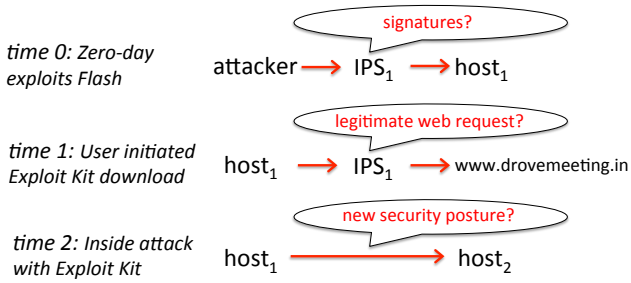


Fig. 4: Does defense system have the agility to mitigate a multistage attack [5]?

Agility: Security postures need to be updated as the *context* of

a device may change. Figure 4 describes a multi-stage attack on our example network similar to those seen in recent incidents [5]. At time 0, $host_1$ receives a phishing mail containing a zero-day attack exploiting a `Flash Player` vulnerability. At time 1, the $host_1$ contacts a drive-by-download website and install a powerful toolkit `Magnitude EK` [11]. The infected $host_1$ initiates the download (so the defender does not block the outgoing request), the attacker uses a short-lived name for the download (so the site is not blacklisted). The (dynamic) context here is $host_1$ is accessing a suspicious website and getting infected. At time 2, $host_1$ contacts and infects $host_2$ using the toolkit, bringing it under the botnet's control. Note that the attacker dynamically adjusted the attack posture by using different techniques at each step, as described in § II-B, which makes it hard for static defenses to mitigate.

Current approaches: To address such dynamic attacks, we need to change the network defenses across different stages. For example, from time 0 to time 1, the administrator would need to reconfigure the defense mechanisms from checking for malicious destination IPs in packet headers to checking for suspicious file download with DPI. From time 1 to time 2, having determined that $host_1$ might have been compromised, the administrator would want to subject $host_1$'s traffic to heavier intrusion prevention to prevent potential exploits against critical resources such as $host_2$. However, adjusting the behavior of existing systems, such as distributed Firewall/IPS, is difficult as they are constrained by topology; *e.g.*, placing an IPS filter in inline mode on demand requires rewiring the network topology and routing. In addition, changes need to happen at the time scale of minutes or seconds. Unfortunately, current approaches are not designed to evolve at such fine timescales.

In summary, these motivating scenarios highlight that isolation, context, and agility are fundamental requirements that any enterprise network security strategy should provide. However, existing approaches exhibit key shortcomings on one or all of these dimensions, as they are constrained by the network topology, the use of fixed hardware-based defenses, and by static policy abstractions. Our goal in designing PSI is to address these pain points.

III. System Overview

In this section, we begin with an idealized approach to address the above problems and highlight why realizing it with current techniques may incur high cost and complexity. Then, we describe the PSI architecture and show how PSI leverages recent advances in SDN/NFV to realize this ideal approach. We conclude by highlighting key technical challenges that we address in designing PSI.

Idealized solution: Let us consider an ideal solution that can provide the desired isolation, context, and agility. We start with a collection of integrated "omnipotent" security appliances that can perform any of the necessary security functions; e.g., Layer 3 firewall, DPI, anti-virus, and application-level firewalling. We ensure that each device has one of these appliances physically connected as its immediate "next hop". Finally, we employ a global policy enforcement algorithm that can dynamically configure these appliances in real-time to add/drop processing modules as needed and change the configuration; e.g., invoking

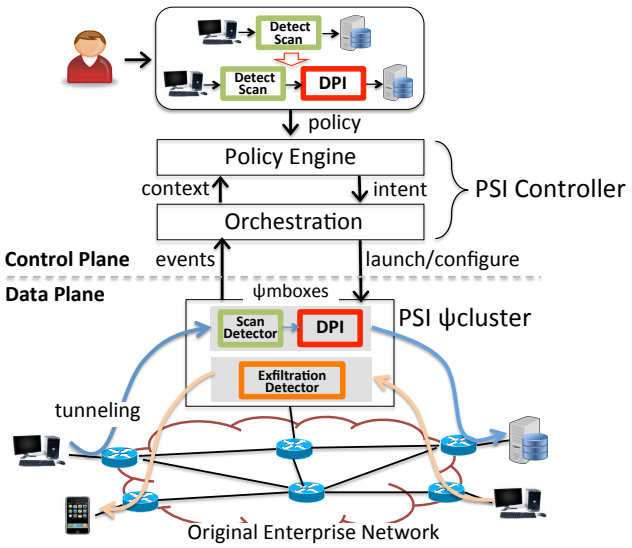


Fig. 5: A high-level view of the PSI.

the DPI module on-demand for TCP flows to suspicious destinations.

By construction, this design addresses the pain points mentioned earlier. First, since we have a dedicated processing appliance per-device, the policy applied is logically isolated and there is no cross-device performance interference due to multiplexing. Second, because this appliance is directly connected to the device it is protecting, it has all the relevant processing context needed to apply a given security posture. Finally, the rules and modules can be added/dropped dynamically to provide the necessary agility to change the posture.

Challenges with the idealized solution: Unfortunately, the approach described above is impractical on several fronts. First, in terms of deployment complexity, we need a dedicated hardware appliance attached to every device. Even ignoring the complexity of rewiring the network, this is an uphill task given the number of wireless, mobile, and virtualized devices. The second issue is cost; adding as many physical hardware appliances as there are devices is a non-starter for medium-to-large scale enterprise networks with tens of thousands of devices. Third, such an omnipotent device that can dynamically reconfigure its traffic processing does not exist today. While future solutions (e.g., [59]) may offer such a consolidated appliance, we have to embrace the practical concern that security functions are fragmented across different vendors with diverse capabilities. Finally, even if we had the appliances, the policy abstractions offered by current frameworks are not expressive enough to capture agile security postures.

The obvious question then is whether we can realize the ideal solution with low cost, without requiring changes to the existing network topology, and by using existing appliance capabilities? In a nutshell, this is the practical problem that PSI solves.

PSI approach: To address this problem, PSI decouples the deployment of security appliances from topological constraints by *tunneling* a device's traffic to a server cluster $\psi cluster$.[2] This $\psi cluster$ can provide an appropriate appliance for any

[2]PSI read as the Greek symbol ψ.

traffic, on demand. To address the cost issue, PSI leverages NFV to build "tiny" virtualized appliances ($\psi mbox$ in Figure 5) to share commodity hardware and reduce cost. Note that one commodity server can support up to hundreds of such appliances [46]. To address the lack of dynamic omnipotent appliances, PSI uses SDN capabilities to compose existing appliances (*e.g.*, Snort or Bro) to dynamically steer the traffic within the $\psi cluster$.

Incremental deployability: To deploy PSI, the enterprise needs to add a pool of commodity server machines. Each device's first-hop edge switch is configured to *tunnel* packets to/from the device to the gateway switch of the $\psi cluster$. Note that this tunneling capability is supported even in commodity non-SDN switches. Note that we need SDN and NFV capabilities only inside the $\psi cluster$, which is easy to deploy [14].

PSI walkthrough: Next, we conceptually walk through the various components of PSI. The PSI PolicyEngine takes as input the high-level security posture from the administrator and translates these into per-device intents. The Orchestration module enforces these intents by launching/configuring $\psi mboxes$ and switches. Each device's traffic is tunneled to/from, and processed, in the $\psi cluster$. Event from the $\psi mbox$ or switch are sent to the Orchestration module and passed to Policy Engine. The logic in PolicyEngine will update the intent based on the context, and the Orchestration module dynamically launches and/or reconfigures the $\psi mboxes$ (and switches) based on the updated intent.

Isolation, context and agility provided by PSI: Now let's briefly explain how PSI framework provides the isolation, context and agility desired. For isolation, the Orchestration module will assign a number of dedicated $\psi mboxes$ to each device, and by default allocate a fixed amount of CPU cores and memory to each $\psi mbox$. To isolate the logic, each device has dedicated policies enforced by its dedicated $\psi mboxes$ so that updating the policies will not effect the other devices. For context, PSI attaches the detection and mitigation $\psi mboxes$ to the device's next-hop switches with tunneling, so the traffic from/to the device can not bypass the security enforcements. Therefore, PSI has full visibility to all the context of a device and there is no "blind spot" that attackers can exploit. For agility, PSI simplifies and speeds up the procedure to update security postures. Deploying new security function is as simple as launching a few virtualized instances, avoiding the high cost of deploying hardwares. Similarly, the network configuration is simplified as PSI automatically translates the security policy to context-based forwarding on switches and $\psi mboxes$.

Challenges: Given this overview, two key challenges remain:

- *Expressive policy abstractions (§IV):* Traditional policy abstractions (e.g., in firewall, IDS, ACL) that rely on a simple if-match-then-action paradigm are not sufficient for the agile and context-aware defenses we envision. To this end, we develop an expressive policy abstraction based on an intuitive combination of finite state machines to capture the security state and directed acyclic graphs to capture context-based security actions. We design a Policy Engine that interpret the policy abstraction and computes the real-time security intent updates for each device based on the current context.
- *Scalable and responsive orchestration (§V):* Naive orchestration schemes to realize PSI would cause several scalability and responsiveness problems. For instance, naively sending the first packet of every context change to a central controller introduces new control plane attacks [29], [64]. Naive scheme to launch new $\psi mboxes$ instances at every context change wastes time and resources (CPU/memory). Finally, if the orchestration cannot scale out, the attacker can easily overload the system of PSI. To address these issues, we develop a scalable orchestration platform using a combination of proactive orchestration, pre-fetched installation of $\psi mboxes$, and horizontal scaling.

IV. PSI Policy Abstraction

In this section, we identify key expressiveness requirements that PSI policies should satisfy and then describe our solution.

A. Requirements

We begin by highlighting three key requirements that the PSI policy abstraction should meet to help the administrator to express intents that govern how the devices' traffic should be processed/forwarded.:

- *Context-based forwarding & processing:* The administrator should be allowed to define a set of context for each device, and express forwarding & processing based on the context. For example, if a host is sending oversized DNS packets (detected by header checker), the administrator should be allowed to define a context to note that a host is suspicious for data exfiltration (DNS-based), and specifies that the oversized DNS packets should be forwarded to a DPI for payload check to prevent potential data exfiltration.
- *Agile intent evolution:* The administrator's intent over a device's traffic may evolve over time as the context changes. For instance, if a host initiated a HTTP connection to a suspicious website, the context is changed from host is normal to host is suspicious, and the inbound traffic to this host should be put under deeper scrutiny (*e.g.*, DPI) to stop malicious downloads.
- *Supporting legacy aggregate policies:* Given that existing deployments use security functions that monitor many devices, there may be legacy policies that an administrator may want to define over *aggregated* views of the traffic, *e.g.*, per department policies or global policies [18]. The PSI policy abstraction should also be able to express such legacy policies.

A natural question then is whether prior policy abstractions can satisfy the requirements. We evaluated a number of existing approaches including distributed Firewall/IPS configurations (Cisco's PIX [9],Bro [51]) and more recent SDN policy languages (Kinetic [43], Merlin [67], Group-based Policies (GBP) [18], PGA [52]), and found that they cannot meet the first two requirements. For distributed Firewall/IPS configurations, the context and agility that can be expressed is limited to one type of tools, *e.g.*, a firewall cannot tell a oversized packet to be forwarded to a IPS for DPI. We qualitatively evaluated distributed Firewall/IPS configurations in the coverage test in Section VII-A, and the result shows that the context-based and agile policies prevented 35% more potential attacks than distributed Firewall/IPS configurations.

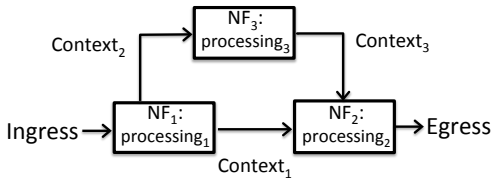


Fig. 6: Express context-based intent with ψDAG.

Similarly, SDN policy languages are largely limited to switch contexts (packet header attributes), and their agility is limited to static forwarding capabilities.

Our goal in designing the PSI policy abstraction is not to claim a novel theoretical or programming language contribution (*e.g.*, [31], [54]). Rather, it is a practical abstraction for capturing context-dependent and agile processing.

B. High-Level Ideas

Context-based processing: To enable administrator to express traffic forwarding and processing based on context, PSI uses a Directed Acyclic Graph called ψDAG, as shown in Figure 6. Formally, this is a two tuple $\psi DAG = \langle NFInstances, NextHops \rangle$. Each vertex in ψDAG represents a processing function (*e.g.*, IPS) denoted as $NFInstance$ that can: (1) tag the traffic based on the processing outcome (*e.g.*, tag packet with "oversized" as context); (2) apply custom process based on the tags (*e.g.*, check the payload for packet with "oversized" context). The edge relation $NextHops$ specifies the intended sequence of traversal through processing tools with respect to context; *e.g.*, packet with "oversized" context is forwarded to a IPS (to check payload).

Supporting intent evolution with ψFSM: While the ψDAG abstraction captures context-dependent processing, it does not capture the evolution of security intent. For instance, the operator may have an inkling of future "states" of the host (contexts at different time) and may want to proactively express the intent for these subsequent likely states as well. Similarly, the intent may evolve as new information arrives; *e.g.*, new vulnerabilities or new alerts from external sources. To capture such intent evolution, we introduce the ψFSM abstraction. Formally, the ψFSM is a Moore machine $\psi FSM := \langle S, s_{start}, \mathcal{E}, \mathcal{F}, T, O \rangle$ that maps the traffic's state to a specific ψDAG via the output function O ($S \rightarrow \mathcal{F}$). In the simplest case, we have a single global state for a specific class with a single ψDAG. More generally, we can define state *transitions* based on different events $E_j \in \mathcal{E}$ and the ψDAG will depend on the current state.

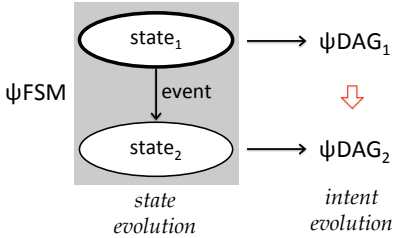


Fig. 7: PSI agile intent evolution.

Note that from Section III, our policy is isolated at a per device/traffic granularity. Therefore, each device/traffic is assigned with a ψFSM and a ψDAG, as shown in Figure 8; the ψFSM captures current state and future state evolution; and the ψDAG captures the current intent with respect to the current state.

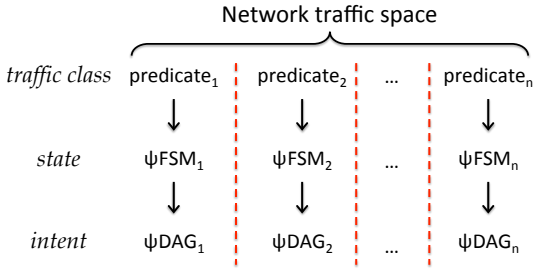


Fig. 8: PSI policy at per device/traffic granularity.

Expressing aggregate policies by scoping: Next we discuss how we can support legacy policies expressed in terms of traffic aggregates, *e.g.*, a device belongs to multiple logical groups [18]. To this end, we add the notion of *scope* to denote the range of devices that a ψFSM and a $\psi mbox$ would apply to. For example, local $\psi mboxes$ denoted as $X_L : NFInstance$ is dedicated to a device, while global $\psi mboxes$ denoted as $X_G : NFInstance$ apply to all devices. For each device, we compose its local $X_L : \psi FSM$ and global $X_G : \psi FSM$ as a new ψFSM. Then for each state in the composed ψFSM, we merge the local $X_L : \psi DAG$ and global $X_G : \psi DAG$. Following the above process, it's easy to convert legacy group-based policies to per-device PSI policies. For example, using the scope notion, we port 348 distributed Firewall/IPS policies to PSI policies in the logical interference part in SectionVII-A.

Illustrative Example: Let's consider an end-to-end example to put the ideas together and see how PSI policy abstraction enables us to express effective security postures. Figure 9 illustrates a dynamic policy that can filter out spurious alarms and precisely identify a multistage attack in real-time. The multi-stage attack includes two stages: 1) Tempting a user to click a link to suspicious websites (unknown IP, initial sign); 2) Device initiated downloads of malicious payloads (worms, Trojans, Exploit Kits). Currently, the initial sign of the attack is hidden in a deluge of alerts (accessing a unknown website can be normal user activity) and always checking the payload will cause collateral damage to the network. With the PSI policy language, the administrator can enforce a dynamic scrutiny policy [60] to react to the initial sign of the attack. As shown in Figure 9, the administrator can define the traffic of a device (IP 10.2.0.1) by set the *sip* field of a predicate to 10.2.0.1. Then the administrator uses the corresponding ψDAG to specify the possible security states (normal, suspicious and malicious) and state transitions of the traffic. By setting a ψDAG for each state, the administrator means that: when the traffic is at normal state, the traffic will be processed by a light weight IPS (L-IPS) performing only header check (e.g., checking for access to unknown IP); if L-IPS detects access to unknown IP, the context will be updated with unknown IP event and triggers the traffic to transit from normal to suspicious, and L-IPS will forward the packet to a heavy payload check (H-IPS); if exploit payload is detected, then the multi-stage attack is confirmed, the traffic state will be changed to malicious, and corresponding packets will be dropped.

Once we have the abstract policy, the PSI Policy Engine will interpret the abstract policy and compute the real-time se-

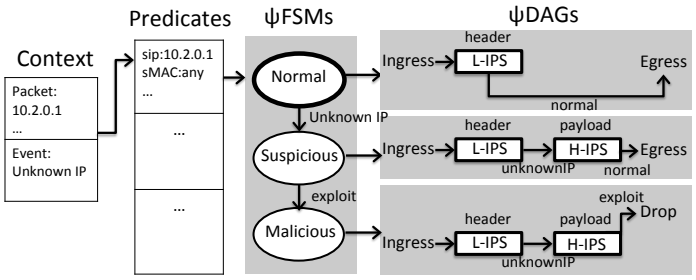


Fig. 9: A dynamic scrutiny policy expressed via PSI.

curity *intent* (ψDAG) for each device's traffic based on current context. In Section V, we show how to translate the *intents* to concrete $\psi mbox$ deployment and network configurations. We also describe a GUI for the administrators to simplify the input of the abstract policy in Section VI.

V. PSI CONTROLLER

In this section, we describe the PSI controller's orchestration mechanisms to translate the high-level intents into a concrete realization. To highlight the key scalability and responsiveness challenges, we begin with a simple *reactive* design. Then, we discuss our ideas to address these challenges: proactive tag-based forwarding, elastic controller scaling, and ψDAG prefetching.

A. Conceptual View and Challenges

The input to the PSI orchestration module is the policy intent (ψDAG) for each device. The goal of the controller is to translate these into a concrete realization; *i.e.*, launch $\psi mbox$ instances and set up forwarding rules. Recall that PSI intents capture dynamic packet processing at two levels. First, the forwarding path may depend on flow-specific context information from upstream nodes in the ψDAG. For example, in Figure 9, traffic with context *unknown IP* from L-IPS should be forwarded to H-IPS for further payload analysis. Second, the ψDAG may itself be updated based on ψFSM transitions in response to alerts or other events. For example, in Figure 9, the ψDAG will be updated when L-IPS raise an unknown IP alert and the state transits from normal to suspicious.

A seemingly natural solution to implement these intents is to adopt a *reactive* mechanism, which configures forwarding rules and deploy $\psi mboxes$ on-demand whenever the context changes. Let us use the intents for normal state and suspicious state in Figure 9 as an example to show how such a reactive controller would react to packet arrivals and other events/alerts (Figure 10).

Per-packet processing: Suppose a packet belonging to class i arrives at a network interface (either at a $\psi mbox$ or a switch). By default, this node will not have any forwarding rule. Thus it buffers the packet and sends an `PKTIN` event to the controller with the packet's header and any relevant processing context. On receiving the `PKTIN` event, the controller retrieves the current state ${s_i}^{current}$ corresponding to this packet's class and uses it to get ${\psi DAG}^{current}$. Based on the current context and the node that generated the packet, the controller decides the next hop for this packet and sets up forwarding rules to ensure that the packet will traverse the intended path. For example, in Figure 10(a), PSI will configure the L-IPS to send pkt1 (destination IP is in the normal IP list) with context "normal" to the controller, and the controller will reactively install a rule at the switch to forward pkt1 to port1.

Event processing: The previous discussion handles the case for a given state s. Other types of events (*e.g.*, IDS alerts) may trigger a state transition in the ${\psi FSM}_i$, e.g., transition from normal state to suspicious state in Figure 9. This in turn may require a new ψDAG to be instantiated. In this case, the controller retrieves the ${\psi FSM}_i$ corresponding to the event,[3] looks up the ${s_i}^{current}$ and identifies the ${s_i}^{next}$. The controller then identifies the new ${\psi DAG}^{next}$ corresponding to the new state s, deactivates the current ψDAG, and launches new $\psi mbox$ instances to implement ${\psi DAG}^{next}$, and updates internal data structures to indicate a state change. For example, when transiting from normal state to suspicious state, the orchestration module will launch a $\psi mbox$ running IPS with a set of payload check rules to implement the H-IPS node with the payload check. Figure 10(a) shows one of the rules in the IPS: check if the payload contains "meta" and "EmulateIE7" to block Magnitude EK.

Challenges: While this above workflow is conceptually correct, we identify two key challenges:

1. *Scalability with adversarial workloads:* First, handling every packet presents a fundamental scaling challenge - a single controller has to process a control message for every packet in the network. Second, even if we do not interpose the controller on every packet at every hop, the controller needs to deal with a large number of events in any reasonable-sized network that will induce ψDAG updates.[4] Thus, an adversary can easily saturate the CPU, memory, or the control channel bandwidth with this naive approach.
2. *Security downtime:* Transitioning from the ${\psi DAG}^{current}$ to the ${\psi DAG}^{next}$ will need new VMs to be launched and other forwarding rules to be setup. Even with fast VM bootup techniques, there will be a non-trivial latency. Thus, adversaries can exploit these delays in setting up ${\psi DAG}^{next}$ to achieve their goals for the types of multi-stage attacks described earlier.

B. Key Ideas in PSI

Next, we describe how address each of these challenges.

Proactive context-based forwarding: The reactive controller does not scale as it interposes on every packet at every logical forwarding between two $\psi mboxes$.

To avoid this, our goal is to keep packets in the data plane as much as possible [36]. We achieve this with a proactive forwarding approach extending prior work [36]. The core idea is illustrated in Figure 10. Each $\psi mbox$ *tags* outgoing packets specifying the relevant context needed for forwarding along the ψDAG. For example, in Figure 10, all packets are initially under "normal" tag and the tag changes to "suspicious". The forwarding logic of the network switches will incorporate these

[3] We assume that events are annotated with the class value (i).

[4] Even if the likelihood of updates for a single device is low, at any given time there are likely to be several devices that need updates.

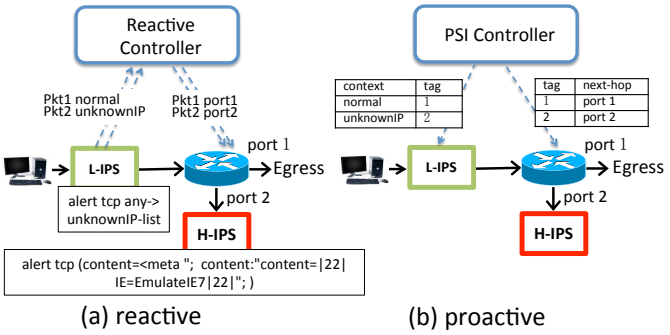


Fig. 10: Comparing reactive and proactive design.

tags as part of their packet processing actions. For example, in Figure 10, if the packet is tagged as "normal", it is forwarded to port 1, and if the packet is tagged as "suspicious" it is forwarded to port 2. Note that because the controller has a logically global view it can proactively install these per-tag forwarding rules for each class $class$ without waiting for a `PKTIN` event.

Now, there might be two potential deployment concerns. First, we need sufficient space in the packet header to add these tags. This is not an issue as new standards for virtual network forwarding and network service chaining headers explicitly include additional header space for metadata [70], [71]. With software switching (*e.g.*, OpenvSwitch) and new switch pipelines [28], it is possible to add flexible header matching rules based on these tag bits. Furthermore, these header tags are only needed inside the $\psi cluster$, where we are not constrained by legacy networking. Second, the $\psi mboxes$ have to explicitly expose these tags. Prior work shows that the modifications required to commodity middleboxes to add the tagging logic is less than 50 lines of code [36].

Scale-out controller: While proactive forwarding addresses the scaling problem in dealing with `PKTIN` arrivals, it does not address the issue of an adversary sending crafted data to generate a large event/alert volume to overload the controller. Note, however, that it may be hard to distinguish events triggered by adversary actions vs. legitimate users. Rather than introduce ad hoc anomaly detection algorithms to try and differentiate legitimate vs. adversary-induced events, we exploit the fact that the PSI design allows us to logically partition the event handling across different traffic classes. That is, we can simply *horizontally scale* the PSI controller and add more instances as needed depending on the offered load [32], [45]. This is especially easy because the different traffic classes are independent and do not introduce any synchronization bottlenecks at the controller.

We design a scale-out mechanism similar to elastic scaling solutions in cloud deployments [4], [32]. While the elastic scaling solutions only migrates switches states, PSI's scale-out mechanism migrates device attributes, security states and policy specifications within the predicates, ψFSM and ψDAG structures. A simple runtime monitor inspects the response time for each controller instance. If the response time is starting to increase more than a preset threshold, it invokes an elastic scaling routine that adds an extra control instance and splits the traffic classes currently handled by the over-loaded instance across the added instances, and migrate the corresponding structures.

ψDAG prefetching: The combination of proactive context-based forwarding, partitioning, and elastic scaling effectively addresses the scalability bottleneck. However, the problem of security downtime during ψFSM transitions still remains.

To address this, we use the following idea. Since the controller has the entire ψFSM described by the policy intent, it can look ahead and predict the next k possible states for each class $class$. Then, it proactively installs the $\psi mboxes$ corresponding to the ψDAGs for these next k possible states, in order to mask the latency involved when these might be needed in the future.

One concern might be that this needlessly increases the resources used by the $\psi cluster$ as ψDAG instances may never be exercised. While this is theoretically valid, in practice, we can address it as follows. First, the controller installs the future ψDAGs for the same class to be multiplexed on the same hardware as the current ψDAG for that class. Since there is only one active ψDAG for a given traffic class at a given time, this incurs no additional hardware resources. Second, we implement some simple optimizations to do an *incremental* launch; *e.g.*, if there are common $\psi mbox$ instances across the future ψDAGs then we can reuse these instead of cloning them. This last optimization also indirectly helps to reduce the downtime by reducing the number of new VMs we need to launch.

VI. PSI Implementation

We have implemented a fully functional PSI system using open source SDN/NFV tools, consisting of around 8K lines of code [19]. In this section, we briefly our implementation and the extensions we made to open source tools to enable the PSI vision.

Controller: We choose `OpenDaylight` a popular industry-grade SDN controller as our starting point. Since `OpenDaylight` only focuses on simple forwarding functions, we add several extensions to support PSI:

1. *Operator interface:* We write a custom GUI to make it easy for operators to enter intended policies, as shown in Figure 11. The PSI GUI can take in both textual (in a domain-specific language) and graphical (by adding and inter-connecting graph nodes) forms. Then a policy parser translates the input into ψFSM and ψDAG data structures for each traffic class. The policy parser also verifies the consistency of the policy (e.g., the same traffic flow is not be assigned conflicting policy actions, such as drop/pass) using checking mechanisms similar to those of Fireman [74].
2. *Event handler:* `OpenDaylight` natively only handles `OpenFlow` messages from SDN switches. We extend it to handle events from common security appliances; *e.g.*, Snort IDS alerts.
3. *Orchestration:* We write custom code to setup tag-based forwarding rules on switches. Our VM launching scripts interact with the KVM hypervisor running at the $\psi mbox$ node (see below). This orchestration module is also responsible for implementing the ψDAG prefetching logic from the previous section.

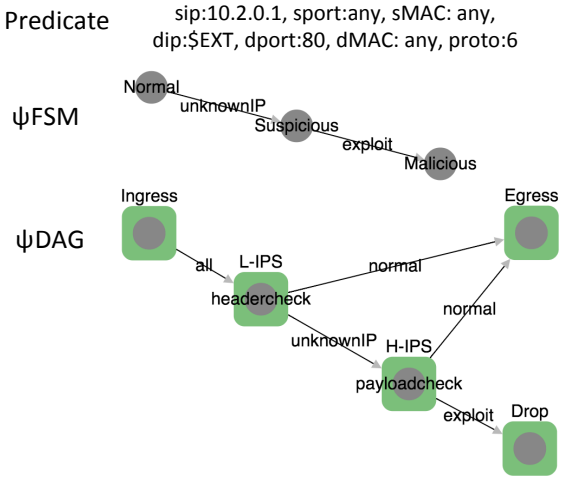
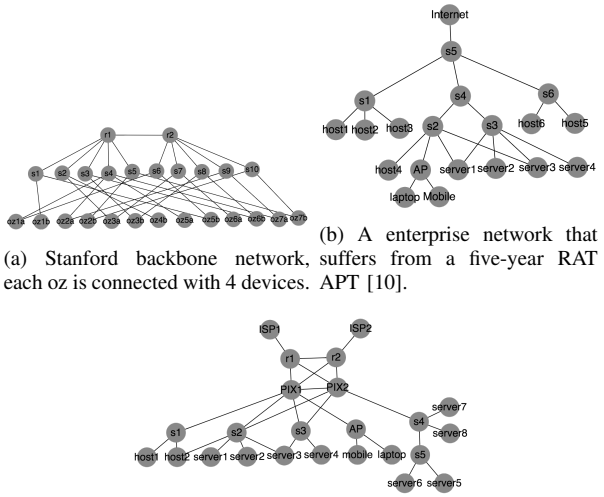


Fig. 11: The dynamic scrutiny policy expressed via PSI GUI.



(a) Stanford backbone network, each oz is connected with 4 devices.

(b) A enterprise network that suffers from a five-year RAT APT [10].

(c) Enterprise with cisco PIX firewall.

Fig. 12: Evaluation topologies.

4. *$\psi mbox$ control channel:* We also implement custom control channels between the PSI controller and the individual $\psi mboxes$. These serve two purposes. First, they implement $\psi mbox$-specific messages for reconfiguring policies; *e.g.*, installing and updating $\psi mbox$-specific configurations. Second, these control the tagging behavior of the $\psi mbox$ to enable the proactive tag-based forwarding scheme described earlier.

5. *Runtime scale-out:* We implemented simple Java runtime monitors to inspect the response time on each controller instance. Once the response time exceeds a threshold, we scale out the instance as follows. Suppose the stressed controller instance is *C1*, the corresponding runtime monitor *RM1* then: 1) launch a new instance *C2*; 2) offload half of the traffic classes, and their FSM, $NFDAG$ with the FSM current state into a policy configuration file and send to *C2* in a notify message. *C2* then inputs the policy configuration file from RM1, and setup connections with the related NFs and switches. Finally, *C2* send a ACK message to *C1* and *C1* closes the socket connections with the related NFs and switches.

6. *Composing ψFSM and scoping $\psi mbox$ instances:* We implemented an ψFSM composition module that can input all policy intents that applies to a device and compose the policy ψFSM by calculating the cross product of the states, the union of events and the new transitions. To scope $\psi mbox$ instances, we implemented a scoping scheme that inputs the natural scoping order (a sequence of scope IDs) from the administrator, assigning each $\psi mbox$ instance with a corresponding scope ID, and steering all the traffic in scope through the instance, following the scoping order.

PSI data plane: To realize each $\psi mbox$, we use virtual machines. We chose this over alternatives like containers (*e.g.*, Docker) as it offers stronger performance isolation across different traffic classes. We currently support several open source security functions; *e.g.*, NAT/firewalls using iptables, IDS/IPS using Snort [57], proxies using Squid [23], and load balancers using Balance [7]. Each $\psi mbox$ runs inside a VM; we use Centos 6.5 as the host OS running the KVM hypervisor. We set up simple tunnel-based forwarding rules at the ingress switch to steer the traffic to the $\psi cluster$. These $\psi mboxes$ need two minimal extensions to integrate with PSI. First, we extend the $\psi mbox$ implementations to support the addition of tags to outgoing packets to enable the tag-based forwarding [36]. Second, we need to forward events/alerts to the PSI controller. We implement this by having a light-weight PSI client program that parses these alerts (*e.g.*, Snort alerts) and forwards them to the controller. This alert parsing program is configurable and can be customized to only forward relevant events to reduce the control plane load.

VII. Evaluation

In this section, we evaluate PSI to show that:

- **Security benefits:** When working against stealthy attacks, PSI identifies and mitigates 35% more attacks than a distributed Firewall/IPS solution. PSI is able to more effectively implement complex distributed solutions. PSI eliminates the logical interference and reduces the performance interference damage by 85%.
- **Scalability, responsiveness and resilience:** Proactive context-based forwarding reduces the end-to-end latency by at least 10X over the baseline performance. PSI's ψDAG prefetching mechanism reduces security downtime during ψFSM transitions from seconds scale to zero. With the optimizations, a single PSI controller can support a network with 100K devices, and can support complex policies with up to 10 states with a size-10 DAG for each state. Our scale-out scheme cuts the response time down to 10ms even in the presence of an adversary.

Since PSI is an enabler for existing/emerging security tools, not a new detection algorithm for a specific attack, we evaluate the benefits for a whole enterprise network (coverage over attack paths, collateral damage) rather than show the ROC (receiver operating characteristic) curve (FP vs. FN) for a particular end-point.

Experimental Testbed: Our experiments run on a cluster of 12 Dell R720 machines, each with 20-core 2.8 GHz Xeon

CPUs and 128GB of RAM. A single PSI controller is running on a VM with CentOs 6.5, assigned with 4 cores and 8GB RAM. On the testbed, we setup three typical enterprise/campus topologies as shown in Figure 12, details about the topologies are given in Table I. We use 1 Dell R720 machine as the PSI $\psi cluster$. To stress test our setup (controller scalability/overloading from adversary), we extended the `cbench` tool [68] to emulate IDS/IPS alerts in large enterprise-like settings.[5]

Topology	Devices	Switches	Information
mini-stanford [42]	56	12	Stanford backbone network
apt-mcafee [10]	12	6	Enterprise network with APT, reported by McAfee
pix-cisco [22]	14	8	Enterprise network with Cisco PIX firewall

TABLE I: Experiment topologies.

A. Security Benefits for Network Structure

We now show how PSI's context-aware, dynamic and isolated approach can mitigate stealthy attacks and reduce the collateral damage caused by logic&performance interference. Here a stealthy attack is one where the attacker can exploit a blind spot in network design to circumvent defenses. For example, an insider threat that originates within a network, evading all outward-facing defenses. In this section, we conduct two analyses. First, we compare PSI against distributed Firewall/IPS approach by evaluating their capabilities to prevent potential insider threats and APT (Advanced Persistent Threats) in three sample enterprise networks. Second, we use real enterprise policy and manipulated enterprise traffic to show how PSI's isolation mechanism can effectively reduce the collateral damage caused by logic/performance interference.

Coverage over stealthy attacks: We now evaluate PSI's ability to deal with stealthy attacks. To do so, we conduct an attack graph analysis against two example stealthy attacks: an example insider threat attack and an example APT. We compare distributed Firewall/IPS's and PSI's ability to detect and mitigate both attacks over three different network topologies shown in Table I. Through this analysis, we show that PSI is capable of identifying and mitigating 35% more potential stealthy attacks than a distributed Firewall/IPS approach, as demonstrated in Table II. These results are due to topological blind spots that PSI is designed to address, and which are common on live enterprise networks such as the examples in Table I.

Attack graph analysis [61] evaluates defensive systems via graph coverage. Each attack is expanded to a graph showing the potential routes the attacker can use to achieve their goals, and defenses are evaluated by comparing the number of paths each defense cuts off from the attack. We chose attack graph analysis because it integrates the structural impact of the network's design on the attack's effectiveness, and enables us to identify the cause of defensive failure. Our attack graphs consist of a tree, G, where each node is a device on the network, and each edge is an attack step. In G, an attack is a path from the root device (the source of the attack) to the leaf (target), e.g., an malware exploit from a laptop to a server through a local switch. An attack is *prevented* if one step of the attack is detected and prevented by the defense system, e.g., a IPS connected to the switch detects and blocks the malware traffic. This yields a coverage metric of the form: $coverage = \frac{num.\ of\ prevented\ attacks}{num.\ of\ all\ possible\ attacks}$. This coverage metric evaluates how many potential attacks are prevented in a given network topology.

[5] `cbench` only supports simple OpenFlow messages by default.

Topology	distributed Firewall/IPS Coverage	PSI Coverage
mini-stanford [42]	52%	92%
apt-mcafee [10]	59%	91%
pix-cisco [22]	56%	89%
all	56%	91%

TABLE II: Coverage over stealthy attacks.

In our evaluation, we instantiate the three enterprise networks in Table I in our testbed and install distributed Firewall/IPS and PSI respectively. For distributed Firewall/IPS, we assume there are no resource constraints and deploy a distributed instance at every switch in the topology. To setup insider threats in each enterprise network, each time a device is set as an insider and it exfiltrate data from all other devices using ftp-based exfiltration or DNS-based exfiltration [2]. For APT, we setup a device in external network as an attacker, and assume it can always break-in with a zero-day attack. We, then, use two pcap traces (Angler EK and Magnitude EK [15]) to emulate the following port-scanning and exploits phases of the APT attack and see if the exploit traffic can reach another device (if yes, the APT attack on the device is valid).

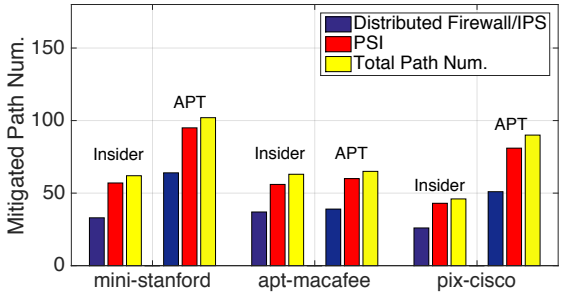


Fig. 13: Detailed results about coverage for insider threats and APT in each enterprise network.

The detailed evaluation results are presented in Figure 13 and Table II. In summary, out of 428 potential attack paths, distributed Firewall/IPS mitigates 240 paths (56% coverage), while PSI mitigates 392 potential paths (91% coverage). We analyzed the 152 paths that PSI mitigated but distributed Firewall/IPS did not—all of them are caused by one or more fundamental topology constraints that distributed Firewall/IPS cannot address. Specifically, 103 paths involve NAT/DHCP; 75 paths involve devices connected to multiple switches (*e.g.*, For high bandwidth or failure tolerance [20]); and 54 paths involve dumb, unmanaged switches. PSI cannot provide perfect coverage because first step, a zero-day attack, is undetectable. In summary, PSI improves the coverage for stealthy attacks by 35%.

Logical interference: Now, we evaluate the defense system's ability to reduce logical interference. To do so, we take real ACL policies [22] from enterprise/campus networks and see if distributed Firewall/IPS configuration or PSI policy language causes any logical interference when expressing them. The ACL policies are already expressed as distributed Firewall/IPS configurations by the administrator; 249 are expressed using Cisco PIX, 65 policies using Juniper SRX, and 34 using iptables. We use Springbok [22], an automated firewall misconfiguration checking tool using the mechanisms in Fireman [74],

to check the logical interference in the distributed Firewall/IPS configurations. Springbok checks for three types of logical interferences: shadowing, redundancy and correlation; in each case a packet is specified with two interfering actions (pass/-drop). Then, we express all 348 policies using PSI policy abstraction and check the logical interference with a Springbok implementation extended to support PSI policies.

System	policies	Shadowing	Redundancy	Correlation
Cisco PIX	249	8	14	4
Jupiter SRX	65	5	5	3
Iptable	34	4	6	4
PSI	348	0	0	0

TABLE III: logical interferences in distributed Firewall/IPS configurations/PSI policy abstractions.

Table III shows the number of interferences of each type between configuration rules. As the table suggests, logical interference occurs in the PIX, SRC and iptable configurations, while it is absent in PSI. The key reason is that the distributed Firewall/IPS configurations follow an If-Match-Then-Action list where preceding rules can interference with the rules after; while PSI provides isolated states and intents for non-overlapping traffic.

Performance Interference: Next, we evaluate the defense system's ability to reduce performance interference. To do so, we generate one elephant traffic flow and several mice flows in the network when different security functionalities are deployed, and measure the damage induced by the elephant flow (in terms of packet drops).

We install both PSI and distributed Firewall/IPS on the three topologies in Table I. The security functionalities we run include deep packet inspection (DPI), anti-virus (AV) and application control (AppCtrl). We generate 8 pcap files by accessing a server in our testbed that runs ssh, ftp, smtp and http service; each pcap file records 24 hours' traffic. The pcap files contains 9% of SSH (secure shell), 12% of FTP (file transfer), 19% of SMTP (email) and 60% of HTTP traffic (by traffic volume). Then, we replay the pcap files on 8 devices in each topology. To generate the elephant flow, we replay a 24 hours trace in 10 min. The other 7 traces are replayed at normal speed as mice flows. We enforce DPI, $DPI + AV$ or $DPI + AV + AppCtrl$ on the elephant flow with PSI or distributed Firewall/IPS. We measure the average packet loss rate over all 8 Devices as the metric to indicate the collateral damage. The results in Figure 14 show that PSI generates around 5% average packet drop rate in three cases, while packet drop rate with distributed Firewall/IPS is 34%. This is because PSI provides customized & isolated $\psi mbox$ for each device, and can precisely enforce DPI only on the elephant flow. In conclusion, PSI reduces the collateral damage of an elephant flow (in terms of average packet drop) by 85%.

B. Benefits of PSI optimizations

Next, we demonstrate how PSI's optimizations from Section V improve the system performance.

Proactive context-based forwarding: To analyze the impact of proactive controller setup, we send flows from one host to another using a ψDAG with two paths and measure the latency per flow. With a reactive controller, the first packet of the flow

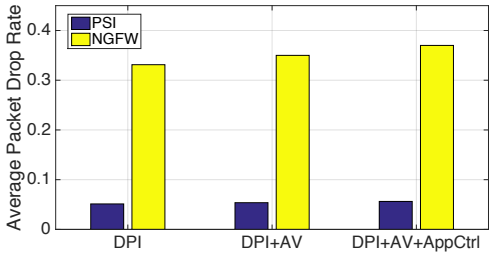


Fig. 14: Collateral damage due to an elephant flow.

generates an event (Snort alert) and waits for the controller to process the event, select the path, and install the forwarding rules. With proactive handling, the packet is tagged at the NF as discussed in §V-B. We gradually increase the load and see how it affects the end-to-end latency in Figure 15. We see that proactive handling mechanism reduces latency by at least 10X. Note that, unlike the reactive controller, the latency of proactive handling is independent of the traffic rate.

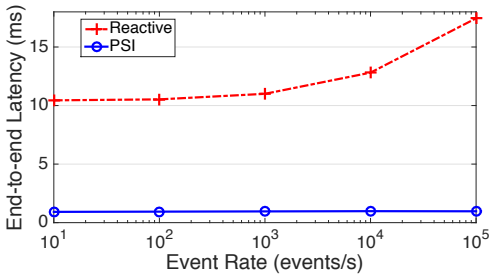


Fig. 15: Proactive context-based forwarding.

ψDAG **prefetching:** To evaluate the effect of ψDAG prefetching, we test the following scenario: a host $H1$ is protected by a policy with 11 states, each with a unique ψDAG. For simplicity, the state transitions are sequential as $s_0 \xrightarrow{E1} s_1 \xrightarrow{E2} s_2 \xrightarrow{E3} \cdots \xrightarrow{E10} s_{10}$. An attacker create 10 types attack flows $a1, \cdots, a10$ against $H1$, which triggers the transition event $E1, \cdots, E10$ respectively. The hope of the attacker is to make transitions from $a1$ to $a10$ sequentially to get through before the defense system can respond. In our evaluation, we write a script to send 10 attack flows $a1, \cdots, a10$ as a group (with different payloads) sequentially from $H2$ to $H1$ to simulate such this attacker. We vary the attack flow arrive interval from 2s to 5s to simulate the frequency of the attack, as illustrated by the x axis of Figure 16. We measure the time between the arrivals of the first and the last packets of the flow at $H1$ as the downtime of the flow and calculate the average down time across 10000 groups of flows. We compare a naive update scheme which waits for the event to trigger ψDAG launch vs. our prefetch scheme with 1-hop and 2-hop look-ahead. For the naive-scheme, there are 3 VMs in each ψDAG and they are launched in parallel. Figure 16 shows that with a 2-hop prelaunch, we can ensure zero security downtime.

Controller scale-out: In this evaluation, we stress our controller with our test generator to emulate an adversary and see how PSI's scale-out scheme maintains a small response time under attack. We connect 20 test generators to our controller, with a 100 events/s rate initially. Then, we select 10 test generators to increase the rate to 110,000 events/s simultaneously to emulate the adversary. At each PSI controller

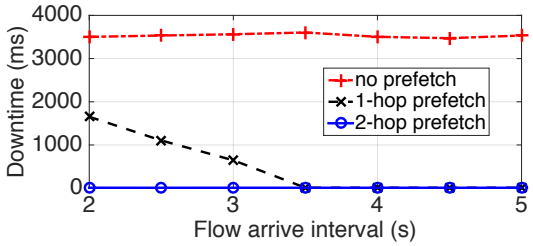


Fig. 16: Effect of ψDAG prefetching.

VM, PSI monitors the average process time of each event with an high delay threshold (set to 8ms). If the threshold is reached, then PSI scale-out scheme will launch new VMs to scale-out as described in Section V-B.

Figure 17 shows the changes of the response time as the load increases. We observed that each peak decrease is caused by scaling out the PSI controller to 2, 3, 4 and 5 VMs respectively. From the result we can conclude that the PSI scale-out scheme successfully prevents the response time from increasing beyond 10ms in the presence of an adversary.

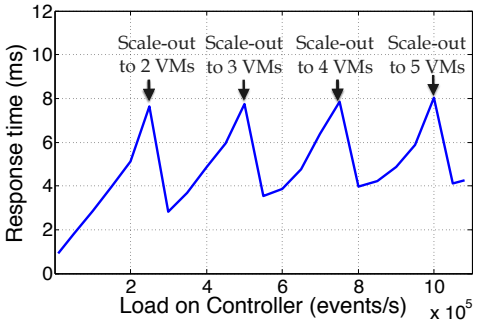


Fig. 17: PSI controller scale-out.

C. PSI Scalability

We now evaluate PSI's scalability in supporting a large-scale enterprise network.

Single controller benchmark: First, we benchmark the event processing throughput a single PSI controller instance. Our test generator sends event messages to the controller as fast as possible (until their TCP send buffer blocks). Each message triggers a state change in the controller logic. We increase the number of generators and measure the observed throughput.[6] In this evaluation, the controller is assigned 8 GB RAM, and hosts ψFSM and ψDAGs for 100,000 devices (one policy for each device). Each ψFSM has 4 states and each ψDAG has 4 $NFInstance$s. The result shows that a single PSI Controller has a maximum throughput of around 230,000 events/s (not shown). To put this in context, if each device generates one event every 5s (a high average alert rate [43]); a single instance could still handle a network with 100,000 devices.

Sensitivity to policy complexity: Next, we evaluate the impact of ψFSM and ψDAG size, which reflect the policy complexity. In this evaluation, the controller is fixed to host ψFSM and ψDAGs for 100k devices (one policy for each device). Then increase the size of ψFSM (number of states) and size of ψDAG (number of security appliances for one state) respectively from 2 to 10 to measure the controller's performance's sensitivity to policy complexity. Figure 18 shows that while the policy complexity does impact throughput and response time, it is quite negligible. The RAM increases linearly with the number of ψFSMs or ψDAGs but can easily supported by extension. In summary, we observe that richer policies are not fundamental bottlenecks for PSI deployment.

[6]We observed that a single generator cannot saturate the controller.

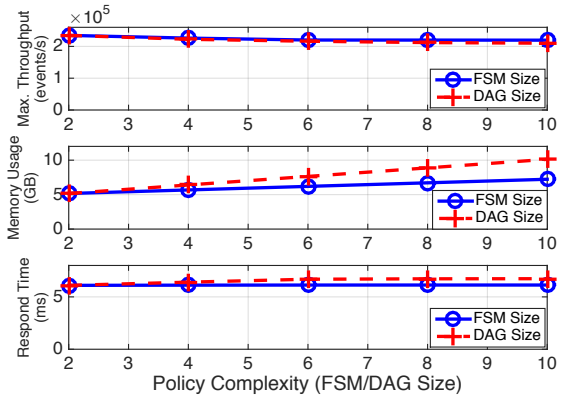


Fig. 18: Sensitivity to policy complexity.

D. PSI Use Cases

In this section, we describe a number of use cases that demonstrate how PSI can enable new security capabilities:

Protecting IoT devices with default passwords: Commonly known default passwords are a problem for embedded devices. To secure these devices, we implemented a simple proxy system as shown in Figure 19. This IoT-$\psi mbox$ (A Ubuntu VM with a customized Squid proxy) serves as a gateway to all traffic to embedded systems on the network. The target in this example, was a D-link surveillance camera, which ships with a hardcoded admin password that the user has no interface to delete. In this case, incoming traffic is inspected for known suspicious password combinations (*e.g.*, admin/admin) are dropped and actual access is only granted by rewriting packets to the correct password combination. The $\psi mbox$ can enforce the use of a new administrator-chosen password to access the camera's management interface or images.

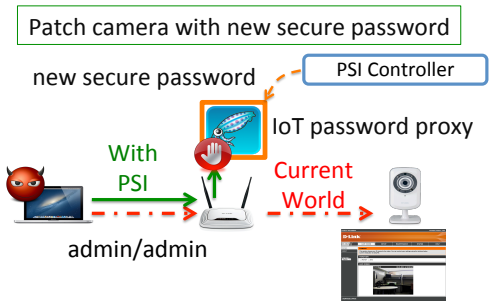


Fig. 19: $\psi mbox$ to Patch Embedded Vulnerability.

Disrupting botnets via on-demand captchas: Today's communication between bots and command-and-control (C&C) servers use common protocols such as `HTTP` to hide among legitimate traffic flows. Now a naive whitelist-based approach (*i.e.*, allowing known popular sites) that raises an alert (or worse block) every off-whitelist HTTP URL access can result in very high false positives (or disrupt legitimate traffic.)

Instead, we can enable a *agile* botnet disruptor to verify if the connection was intended by the user as shown in Figure 20. We implement this a $\psi mbox$ running an `Apache` server, with the Google `ReCaptcha` service as follows. when a user is seen sending a log of suspicious HTTP requests (*e.g.*, short-lived domains), then the botnet disruptor *dynamically* forces the user to enter a "captive portal" with a captcha forcing the user to verify the connections. If the request was legitimate and validated as human, the connection is allowed, otherwise the device is flagged (or disconnected pending further investigation).

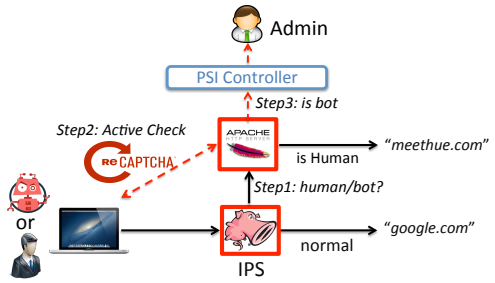


Fig. 20: PSI Bot Disruptor.

Context-Aware On-Demand Decoys: Scans and background radiation are an operational reality today viewed as "ankle biters" that are fundamentally not worth analyst attention [26]. However, in some cases, scans might foreshadow an upcoming attack; *e.g.*, after HeartBleed is disclosed, and understanding the attack workflow may provide valuable information for future defenses. Building on the earlier example, we implement a context-aware dynamic decoy system [25]. Once we identify a specific scanner IP from a host H_1 (*e.g.*, Ubuntu-12.04, SSL1.01-4), we subject it to tighter monitoring via an IPS. If the IPS detects a follow-up exploit action, then PSI instantiate a "decoy" honeypot $\psi mbox$ tailored to Ubuntu-12.04 with SSL1.01-4 and redirects the exploit traffic to the honeypot in order to investigate the attack's intent.

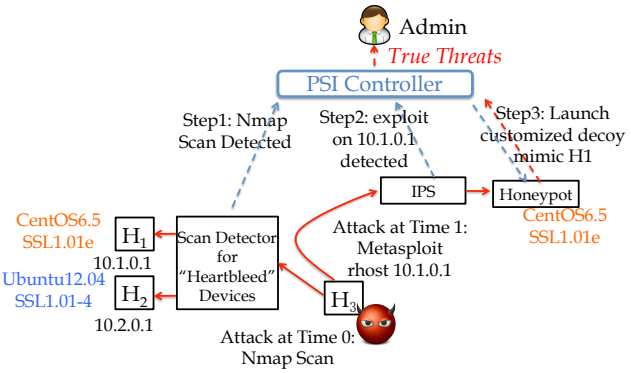


Fig. 21: Context-aware Decoy.

VIII. Related Work

Firewalls and IDSes: Middleboxes such as firewalls and IDS/IPS [51], [57] are the "workhorses" of network security mechanisms today. Unfortunately, these have well-documented concerns with respect to (a) performance (*e.g.*, 30% of administrator disable useful security features such as DPI and anti spoofing [16]); (b) misconfigurations (*e.g.*, [35], [49]); (c) lack of expressiveness to tackle novel threats; and (d) undesirable tradeoffs between stronger security postures vs. user backlash [16]. By design, PSI addresses these pain points. PSI's vision shares conceptual similarity with classical work on distributed firewalls [27], [41]. Other work addresses orthogonal problems related to scalability (*e.g.*, [69]) and alert correlation (*e.g.*, [58]).

SDN, NFV, and Security: Prior work has aimed to take advantage of SDN and NFV to make network security enforcement more flexible. Ethante [29] uses a centralized controller controls switches at critical points to authorized traffic. FlowGuard [40] resolves interfering ACL policies from firewalls in SDN network. However, Ethane or FlowGuard does not support security policies beyond ACLs and do not allow more advanced security policies captured by ψDAGs and ψFSMs. Flowtags [36] and Simple [53] are two other related works close to PSI. Flowtags provides interfaces between middleboxes and the SDN controller to enable the enforcement of "fixed" ψDAG. Simple provides simplified traffic steering over a set of statically deployed middleboxes. FlowTags or Simple supports neither dynamically changing ψDAGs (which are accommodated in PSI using ψFSMs) nor guarantees policy isolation.

FRESCO [62] implements detection and mitigation modules in the SDN controller. However, the controller becomes a critical bottleneck for scalability and requires reimplementing functionality that is commonly available in security middleboxes. OFX [66] and Kinetic [43] focuses on networks composed of switches. PBS [39] shares some our motivation in addressing security challenges induced by BYOD. In contrast to PSI, PBS: (a) only focuses on BYOD mobile apps; (b) involves a reactive controller; and (c) needs end-point instrumentation on Android. Other work focuses on exploiting SDN/NFV to provide elastic scaling of security functions [34], [50]; such elastic scaling is orthogonal to the focus of PSI. That said, the PSI $\psi cluster$ can leverage these elastic scaling capabilities if needed.

Controller scaling, either via horizontal scaling (*e.g.*, [32], [44]) or proactive orchestration (e.g., [34], [36]) are active areas of work in SDN/NFV. PSI synthesizes and extends these efforts through a combination of proactive tag-based forwarding and elastic scaling.

Policy languages: There has been renewed interest in programming abstractions for networks [43], [48]. Kinetic provides a domain specific language and an SDN controller to dynamically change OpenFlow switch actions [43]. However, Kinetic is constrained and cannot express richer policies that involve stateful middleboxes [38]. PGA [52] provides support for composing forwarding policies across aggregates and detecting conflicts. PSI provides a richer abstraction that subsumes these prior efforts.

Industry efforts: Google's BeyondCorp initiative [72] focuses on authentication and user trust, and can conceivably be defeated by evading the authentication system (*e.g.*, a malicious insider). PSI's focus on traffic behavior is intended to be robust to insider threat and other attacks by avoiding a single point of trust. VMWare's NSX and microsegmentation tackles datacenter security by pushing firewalling functionality into

hypervisors to tackle "east-west" traffic, which is not protected by perimeter firewalls [71]. However, the security mechanisms and abstractions are restricted to simple firewalling rules. In contrast, PSI targets a much richer set of policies that can involve multiple security middleboxes and does not rely on every device to run atop a hypervisor. Finally, the vision of SDN/NFV is gaining a lot of traction in industry; *e.g.*, Cisco's Evolved services platform [8] describes a high-level architecture similar to PSI. Based on public documentation, however, it does not specifically tackle the kinds of security, policy, and scalability challenges we describe here. These trends further corroborate our arguments that the PSI architecture is viable, and likely the inevitable culmination in response to today's security woes.

IX. Conclusions

Existing network security mechanisms leave defenders at a disadvantage as they have fundamental limitations in terms of: (1) isolation, leading to policies interfering with each other; (2) context, preventing the defenders from creating a truly customized response; and (3) agility, constraining the defenders' abilities to specify dynamic security postures. To address these pain points, PSI leverages recent advances in software-defined networking and network functions virtualization to enable isolated, context-aware, and agile security postures. We addressed key challenges in developing expressive policy abstractions and scalable orchestration mechanisms. We showed that PSI is scalable and can be an enabler for new security capabilities that would be exceedingly difficult to implement with legacy solutions.

We identify two natural directions for future work: (1) better user interfaces for operators to express PSI-based policies and (2) support for cross-device policies (*e.g.*, when the IPS flags a host, the administrator may want to increase monitoring fidelity for other hosts in the same subnet.) Finally, we acknowledge that PSI is not a panacea and can be vulnerable to covert channels used by attackers; *e.g.*, malware via encrypted cloud storage. That said, PSI is a significant step that can help restore some balance in favor of defenders in the constant tussle with more advanced adversaries.

Acknowledgments

This work was supported in part by NSF award number CNS-1440056 and by Intel Labs University Research Office.

References

[1] 10 strategies of a world-class cybersecurity operations center. https://www.mitre.org/sites/default/files/publications/pr-13-1028-mitre-10-strategies-cyber-ops-center.pdf.

[2] 13 signs that bad guys are using dns exfiltration to steal your data. https://theworldsoldestintern.wordpress.com/2012/11/30/dns-exfiltration-udp-53-indicators-of-exfiltration-udp53ioe/.

[3] Acl and nat conflict each other. router stop working. www.networking-forum.com/viewtopic.php?f=33&t=7635.

[4] Amazon web service: Elastic load balancing. http://aws.amazon.com/elasticloadbalancing/.

[5] Angler ek exploits recently patched flash bug to deliver bedep. http://www.securityweek.com/angler-ek-exploits-recently-patched-flash-bug-deliver-bedep.

[6] Arcsight. http://www.ndm.net/arcsight.

[7] Balance. https://www.inlab.de/balance.html.

[8] Cisco Evolved Services Platform. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/service-provider-strategy/brochure-c02-731348.html.

[9] Cisco pix 500 series security appliances. http://www.cisco.com/c/en/us/products/security/pix-500-series-security-appliances/index.html.

[10] Diary of a rat. http://www.provision.ro/threat-management/data-security/content-aware-and-dlp/diary-of-a-rat-remote-access-tool.

[11] Exposed: An inside look at the magnitude exploit kit. http://www.csoonline.com/article/2459925/malware-cybercrime/exposed-an-inside-look-at-the-magnitude-exploit-kit.html.

[12] Fireeye. https://www.fireeye.com/.

[13] Ieee std. 802.1q-2005, virtual bridged local area networks. http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf.

[14] Integrating sdn into the data center. http://www.juniper.net/us/en/local/pdf/whitepapers/2000542-en.pdf.

[15] malware-traffic-analysis.net. http://malware-traffic-analysis.net/.

[16] McAfee Report Reveals Organizations Choose Network Performance Over Advanced Security Features. http://www.mcafee.com/us/about/news/2014/q4/20141028-01.aspx.

[17] Opendaylight. http://www.opendaylight.org/.

[18] Opendaylight,group based policy. https://wiki.opendaylight.org/view/Group_Based_Policy_(GBP).

[19] Psi. https://github.com/PreciseSecurity/PSI.git.

[20] Server nic teaming to multiple switches. http://itknowledgeexchange.techtarget.com/network-engineering-journey/server-nic-teaming-to-multiple-switches/.

[21] Splunk. http://www.splunk.com/.

[22] springbok. https://github.com/conix-security/springbok.

[23] Squid: Optimising web delivery. http://www.squid-cache.org/.

[24] Throughput and Scalability Report, McAfee NGFW 5206, v5.8. http://www.mcafee.com/us/resources/reports/rp-miercom-throughput-scalability-ngfw.pdf.

[25] M. H. Almeshekah and E. H. Spafford. The case of using negative (deceiving) information in data protection. In *Academic Conferences and Publishing International*, 2014.

[26] R. Bejtlich. *The Tao of network security monitoring: beyond intrusion detection*. Pearson Education, 2004.

[27] S. Bellovin. Distributed firewalls. *login* , pages 39–47, November 1999.

[28] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[29] M. Casado et al. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.

[30] M. Chiosi, D. Clarke, J. Feger, C. Cui, J. Benitez, U. Michel, K. Ogaki, M. Fukui, D. Dilisle, I. Guardini, et al. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, 2012.

[31] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[32] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 7–12. ACM, 2013.

[33] B. Elgin, D. Lawrence, and M. Riley. Neiman Marcus Hackers Set Off 60,000 Alerts While Bagging Credit Card Data. Bloomberg BusinessWeek, http://www.businessweek.com/articles/2014-02-21/neiman-marcus-hackers-set-off-60-000-alerts-while-bagging-credit-card-data.

[34] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Flexible and elastic ddos defense using bohatei. In *USENIX Security Symposium*, 2015.

[35] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proc. NSDI*, 2016.

[36] 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 *Proc. NSDI*, 2014.

[37] FirstPost. Wikileaks trial: Tech experts tie Bradley Manning to database breach. http://tech.firstpost.com/news-analysis/wikileaks-trial-tech-experts-tie-bradley-manning-to-database-breach-214128.html.

[38] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, pages 115–131, 2001.

[39] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. Towards sdn-defined programmable byod (bring your own device) security. In *NDSS*, 2016.

[40] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. Flowguard: building robust firewalls for software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 97–102. ACM, 2014.

[41] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199. ACM, 2000.

[42] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.

[43] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, May 2015. USENIX Association.

[44] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Network. In *Proc. OSDI*, 2010.

[45] A. Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson. Pratyaastha: an efficient elastic distributed sdn control plane. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 133–138. ACM, 2014.

[46] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *Proc. NSDI*, 2014.

[47] S. McCarthy. *Business Strategy: U.S. Federal Government IT Security Spending Forecast and Market Outlook*. IDC Govenrment Insights, 2013.

[48] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[49] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[50] S. Palkar, C. Lan, S. Han, K. J. amd Aurojit Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for NFV applications. In *Proc. SOSP*, 2015.

[51] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[52] 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. In *Proc. SIGCOMM*, 2015.

[53] Z. Qazi, C. Tu, L. Chiang, R. Miao, and M. Yu. SIMPLE-fying middlebox policy enforcement using sdn. In *Proc. SIGCOMM*, 2013.

[54] M. N. Rabe, P. Lammich, and A. Popescu. A shallow embedding of hyperctl*. *Archive of Formal Proofs*, Apr. 2014. http://afp.sf.net/entries/HyperCTL.shtml, Formal proof development.

[55] F. Rashid. For discerning hackers, malware is so last year. http://www.infoworld.com/article/2980341/security/for-discerning-hackers-malware-is-so-last-year.html.

[56] M. Riley, B. Elgin, D. Lawrence, and C. Matlack. Missed Alarms and 40 Million Stolen Credit Card Numbers: How Target Blew It. Bloomberg BusinessWeek, http://www.businessweek.com/articles/2014-03-13/target-missed-alarms-in-epic-hack-of-credit-card-data.

[57] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[58] S. Roschke, F. Cheng, and C. Meinel. A flexible and efficient alert correlation platform for distributed ids. In *Network and System Security (NSS), 2010 4th international conference on*, pages 24–31. IEEE, 2010.

[59] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. NSDI*, 2012.

[60] A. Sharma, Z. Kalbarczyk, R. K. Iyer, and J. Barlow. Analysis of credential stealing attacks in an open networked environment. In *NSS*, pages 144–151, 2010.

[61] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Security and privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 273–284. IEEE, 2002.

[62] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. FRESCO: Modular composable security services for software-defined networks. In *Proc. NDSS*, 2013.

[63] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 78–89. ACM, 2014.

[64] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 413–424. ACM, 2013.

[65] P. Simmonds. Perimeter Security: In Memoriam. Infosecurity Magazine, http://www.infosecurity-magazine.com/view/30984/perimeter-security-in-memoriam/.

[66] J. Sonchack, J. M. Smith, A. J. Aviv, and E. Keller. Enabling practical software-defined networking security applications with ofx. In *NDSS*, 2016.

[67] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster. Managing the network with merlin. In *Proc. HotNets*, 2013.

[68] A. Tootoonchian et al. On controller performance in software-defined networks. In *USENIXHot-ICE*, 2012.

[69] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: scalable, stateful network intrusion detection on commodity hardware, 2007.

[70] VMWare. The vmware nsx network virtualization platform. https://www.vmware.com/files/pdf/products/nsx/VMware-NSX-Network-Virtualization-Platform-WP.pdf.

[71] Vmware. Next generation security with vmware nsx and palo alto networks vm-series. In *White Paper*, pages 1–25, 2013.

[72] R. Ward and B. Beyer. Beyondcorp: A new approach to enterprise security. *;login:*, Vol. 39, No. 6:6–11, 2014.

[73] M. Yu, J. Rexford, X. Sun, S. Rao, and N. Feamster. A survey of virtual lan usage in campus networks. *Communications Magazine, IEEE*, 49(7):98–103, 2011.

[74] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[75] H. Zeng. *Automatic Data Plane Testing*. PhD thesis, Stanford University, 2014.