SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2022, Vol. 65 172102:1–172102:18
https://doi.org/10.1007/s11432-020-3193-2

SAND: semi-automated adaptive network defense via programmable rule generation and deployment

Haoyu CHEN[1], Deqing ZOU[2], Hai JIN[1*], Shouhuai XU[3] & Bin YUAN[2]

[1]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074*, China;*
[2]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan* 430074*, China;*
[3]*Department of Computer Science, University of Colorado at Colorado Springs, Colorado Springs* 80918*, USA*

Received 29 September 2020/Revised 23 December 2020/Accepted 25 February 2021/Published online 8 February 2022

Abstract Cyber security is dynamic as defenders often need to adapt their defense postures. The state-of-the-art is that the adaptation of network defense is done manually (i.e., tedious and error-prone). The ideal solution is to automate adaptive network defense, which is however a difficult problem. As a first step towards automation, we propose investigating how to attain semi-automated adaptive network defense (SAND). We propose an approach extending the architecture of software-defined networking, which is centered on providing defenders with the capability to program the generation and deployment of dynamic defense rules enforced by network defense tools. We present the design and implementation of SAND, as well as the evaluation of the prototype implementation. Experimental results show that SAND can achieve agile and effective dynamic adaptations of defense rules (less than 15 ms on average for each operation), while only incurring a small performance overhead.

Keywords network defense, adaptive defense, automated defense, programmable defense, security services, software-defined networking, security management

Citation Chen H Y, Zou D Q, Jin H, et al. SAND: semi-automated adaptive network defense via programmable rule generation and deployment. Sci China Inf Sci, 2022, 65(7): 172102, https://doi.org/10.1007/s11432-020-3193-2

1 Introduction

The importance of adaptive network defense, or adaptive defense for short, is well recognized because defenders often need to adapt to dynamic situations, such as updating and enforcing the network security policy to counter new attacks, assuring the consistence between the dynamic network security policies enforced by multiple network defense tools (or defense tools or tools for short), and accommodating applications' dynamic requirements that may affect the network security policy. In current practice, adaptive network defense is achieved manually, which is tedious and error-prone and thus calls for automation. To the best of our knowledge, the automation of adaptive network defense is largely open despite its importance. This unpleasant situation is in contrast to that host-based defense has been automated to some extent, such as the automation in installing software patches and updating anti-malware signatures. In this paper, we make a first step towards achieving automated adaptive network defense.

1.1 Our contributions

We make three contributions. First, we initiate the study on automating adaptive network defense, and explore the requirements of automated adaptive network defense solutions. To the best of our knowledge, this is the first systematization on the requirements of automated adaptive network defense. These

* Corresponding author (email: hjin@hust.edu.cn)

requirements would guide future research directions towards fulfilling the ultimate goal of automated adaptive network defense.

Second, given the difficulty in automating adaptive network defense, we propose investigating semi-automated adaptive network defense (SAND). We design a SAND architecture, and implement a prototype system that accommodates three network defense tools (i.e., iptables[1)], Snort[2)], and Squid[3)]). For implementing the prototype system, we need to overcome three challenges. (i) How can we accommodate network defense tools that use heterogeneous, native network defense rules (or defense rules or rules for short)? We resolve this problem by using a unified rule representation and automating the mapping (or "translation") between the rules in the unified representation and the native rules used by network defense tools. (ii) How should a defender dynamically generate network defense rules? We resolve this problem by equipping a defender with the capability to write SAND apps to generate network defense rules. (iii) How should a defender dynamically deploy network defense rules? We resolve this problem by leveraging the software-defined networking (SDN) technology to automate the deployment of dynamic defense rules, while noting that leveraging SDN for SAND is economically viable in practice. We will open source SAND to the community.

Third, we evaluate the SAND prototype system via concrete adaptive defense scenarios. Our experiments show that SAND can effectively block (for example) the WannaCry attack. It is worth mentioning that SAND imposes essentially no side-effect on the throughput of network defense tools because the function of SAND (i.e., dynamically generating and deploying new defense rules) is orthogonal to the enforcement of network defense rules.

We will discuss the limitations of the present study, which point out specific problems for future research.

1.2 Related work

We classify related prior studies into two categories: those which are related to the functionality of SAND and those which are related to the implementation of SAND.

From a functionality point of view, there are some recent studies that also leverage programmability for network security, but in different contexts. For instance, Poseidon [1] defends against DDoS attacks with programmable switches; Poise [2] focuses on in-network policies for achieving BYOD security; and software-defined security platform [3] and security gateway [4] are proposed for IoT networks. The most closely related prior work is PSI [5], which investigates how to use context-based network flow forwarding to address some network-layer security problems. SAND is different from PSI in two important factors: (i) SAND addresses both application-layer and network-layer security problems, whereas PSI can only cope with network-layer security problems; (ii) SAND introduces the novel idea of dynamic generation and deployment of defense rules, whereas PSI does not have such an important capability. In addition, there are practical and theoretical studies on adaptive network defense. Practical studies typically focus on defending against distributed denial-of-service (DDoS) attacks, including the detection of DDoS attacks and the mitigation of their damage [6], coping with traffic control [7], and the investigation of countermeasures [8]. There are studies on achieving adaptive defense via moving target defense [9]. Theoretic studies focus on theoretic models of adaptive defense against multiple kinds of cyber attacks (e.g., [10,11]) or specific kinds of cyber attacks (e.g., advance persistent threats [12]); results presented in these studies often are not tested in real-world environments. It is an important future research task to investigate how to incorporate these studies into fully automated adaptive defense or the SAND architecture explored in the present paper.

From an implementation point of view, SAND leverages SDN[4)] and network function virtualization (NFV) [13], which respectively deal with the virtualization of networks and network functions. There are many studies on SDN/NFV-based security, but these studies assume static defense policies or rules. For example, FRESCO [14] provides security services for monitoring and processing network flows; FLOWGUARD [15] and VNGuard [16] implement firewall services as SDN applications; VFW Controller [17] uses SDN mechanisms to achieve elastic control on virtual firewall policies; other studies aim to achieve flexible network security enforcement [18] or assure packets are forwarded in desired paths [19]. SAND

1) iptables. http://www.netfilter.org/projects/iptables/.
2) Snort: Network Intrusion Detection & Prevention System. http://www.snort.org.
3) Squid:Optimising Web Delivery. http://squid-cache.org.
4) SDN: Software-Defined Network. https://www.opennetworking.org/sdn-resources/sdn-definition.

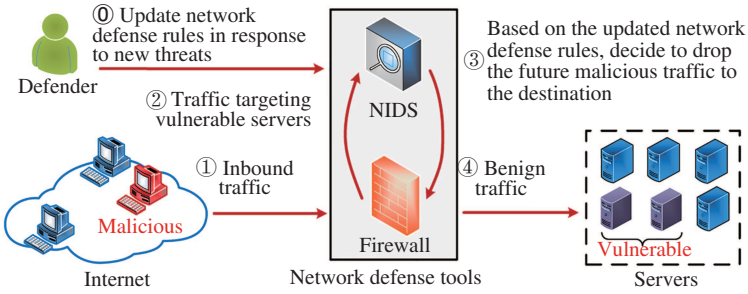

Figure 1 (Color online) Adaptive defense for new policy.

can be leveraged to make these systems enforce dynamic defense rules/policies. We stress that this novel capability of SAND in enforcing dynamic defense rules/policies is beyond the reach of existing network defense tools that may seem to be able to accomplish what SAND can do at a first glance. In particular, one may think that an intrusion detection system can be configured to forward traffic according to its detection result [20], which is comparable to what can be achieved by SAND; this is not true because the intrusion detection system's configurations are static and manually set, and cannot be dynamically and (semi-)automatically adapted in real-time, which is achieved by SAND.

1.3 Paper outline

The rest of the paper is organized as follows. Section 2 explores the SAND architecture. Section 3 describes the SAND prototype system, which is evaluated in Section 4. Section 5 describes the limitations of the present study. Section 6 concludes the paper.

2 The SAND architecture

In this section we motivate the SAND architecture by starting with three application scenarios, exploring the requirements of adaptive defense, the desirable automated adaptive defense which is beyond the scope of the current technology, and the practical SAND architecture.

2.1 Three motivating scenarios

We start with three motivating scenarios of adaptive network defense, which is achieved manually in current practice.

2.1.1 *Motivating scenario 1: adaptive network defense for accommodating new security policy*

When a defender becomes aware of a new attack (e.g., Heartbleed [21] or WannaCry[5)]), a patch may not be available or installed (e.g., in fear of disrupting a critical service). Before a patch is available or installed, the defender can adapt the network defense to counter the new attack without disrupting the service. In the example of WannaCry, which exploits vulnerability MS17-010[6)] at port #445, Figure 1 highlights the adaptive defense: ⓪ Upon becoming aware of WannaCry, the defender updates the defense rules enforced at the firewall and network-based intrusion detection system (NIDS) such that ① any inbound traffic to a vulnerable server at port #445 must be vetted at the firewall and ② when the firewall detects such a network flow, it mirrors the flow to the NIDS, which examines this flow to determine if it contains the WannaCry attack. ③ If the flow contains the WannaCry attack, the NIDS instructs the firewall to drop future packets originating from the attacker and targeting a vulnerable server. ④ The firewall drops WannaCry packets as instructed by the NIDS. Note that this adaptive defense is even more valuable when there are multiple vulnerable servers because a single adaptation in the network defense could protect all of the vulnerable servers.

5) Protecting customers and evaluating risk. https://blogs.technet.microsoft.com/msrc/2017/04/14/protecting-customers-and-evaluating-risk/.

6) Patch to vulnerability MS17-010. https://technet.microsoft.com/en-us/library/security/ms17-010.aspx.

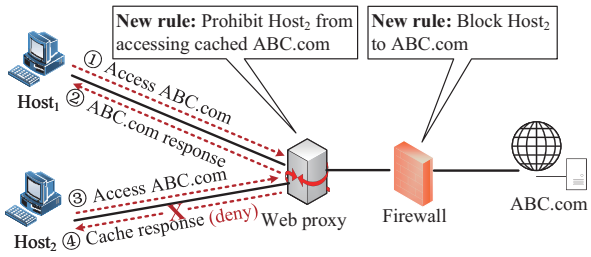

Figure 2 (Color online) Adaptive defense assuring policy consistence.

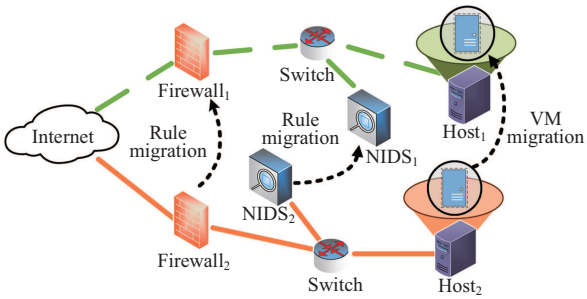

Figure 3 (Color online) Adaptive defense for achieving security policy migration.

2.1.2 *Motivating scenario 2: adaptive network defense for assuring dynamic security policy consistence*

Figure 2 illustrates a network, where a web proxy is used in conjunction with a firewall. Suppose the new security policy says that the firewall must block Host_2's access to `ABC.com`. This policy can be violated when Host_2 accesses the content of `ABC.com` cached at the proxy, which can occur after Host_1's access to `ABC.com`. Thus, the defender must adapt the network defense to assure that Host_2 is prohibited from accessing cached `ABC.com` at the proxy. This can be achieved by adding a new defense rule at the proxy.

2.1.3 *Motivating scenario 3: adaptive network defense for achieving security policy migration*

Figure 3 illustrates a scenario of policy migration, which can be triggered by an application need (e.g., load balancing). In this scenario, the communication between any virtual machine (VM) running on Host_1 and the Internet goes through the dashed path as well as Firewall_1 and NIDS_1, whereas the communication between any VM running on Host_2 and the Internet goes through the solid path as well as Firewall_2 and NIDS_2. This kind of communication path restriction is not uncommon [22]. Adaptive network defense means that when migrating a VM from Host_1 to Host_2, the relevant defense rules enforced at NIDS_1 and Firewall_1 should be respectively migrated to NIDS_2 and Firewall_2.

2.2 Requirements of adaptive defense

We propose 5 basic requirements, namely compatibility, agility, efficiency, security, and effectiveness on adaptive defense solutions.

• **Compatibility.** A solution should accommodate heterogeneous defense tools (e.g., firewall and NIDS). A solution should incur minimal modifications to defense tools when making them support the automated adaptive defense.

• **Agility.** A solution should incur the minimal delay in generating and deploying dynamic defense rules because any delay permits the attacker to cause damages.

• **Efficiency.** A solution should incur a minimal performance overhead, including the time for loading and running the modules affected by adaptive network defense.

• **Security.** A solution should assure the security in the generation and deployment of dynamic defense rules. A solution should be able to resist denial-of-service attacks.

• **Effectiveness.** A solution should achieve the adaptive defense objective in question (e.g., detecting and/or blocking the targeted attacks).

The preceding 5 basic requirements can be extended as needed.

2.3 Automated adaptive defense

The agility requirement motivates automated adaptive defense, which leads to the vision highlighted in Figure 4. This vision is centered at an Orchestrator that automatically responds to external and internal triggers, where external triggers come from the outside environment of a network (e.g., new attacks leading to new security policies as shown in motivating scenario 1) and internal triggers come from the network itself (e.g., assuring dynamic policy consistence as shown in motivating scenario 2 and achieving security policy migration as shown in motivating scenario 3). Automated adaptive defense means that the human defender is not involved.

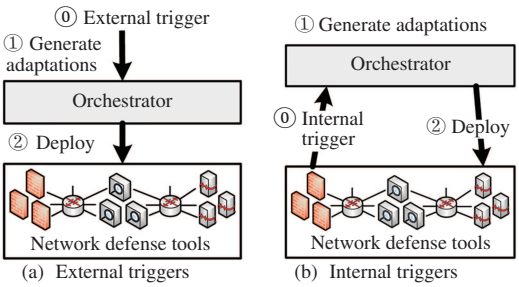

Figure 4 (Color online) A vision for automated adaptive defense. (a) External triggers; (b) internal triggers.

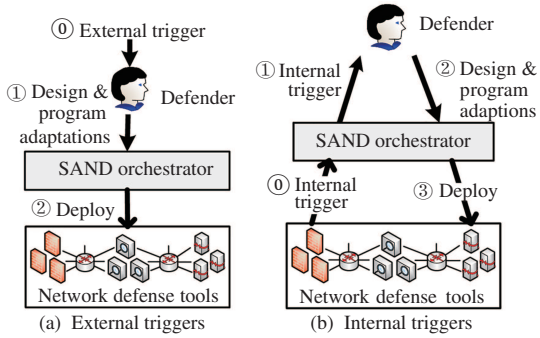

Figure 5 (Color online) The SAND idea derived from the vision in Figure 4. (a) External triggers; (b) internal triggers.

Unfortunately, it is difficult to fulfill the vision, at least for two reasons. First, external triggers are often written in natural language (worse yet, verbal communication) and thus may not be understood by the Orchestrator, which is a software module. Second, handling new attacks or application needs may have to involve human defenders. Addressing these two problems is beyond the scope of the present study.

2.4 Semi-automated adaptive defense

As a first step towards fulfilling the vision mentioned above, we propose fulfilling SAND. As highlighted in Figure 5, the idea of SAND is derived from the vision highlighted in Figure 4, but SAND may require the defender to participate (e.g., writing programs to generate new defense rules).

In order to turn the SAND idea in Figure 5 into a concrete system architecture, we bear in mind that network security management can be seen as a part of network function management because security may be seen as a particular network function. Given that SDN has been motivated by, among other things, the need to automate network function management, we propose leveraging SDN to design SAND's architecture. This choice can be justified as follows. (i) We can leverage SDN's capabilities in obtaining a centralized view of the network and automatically deploying the forwarding rules in SDN switches to flexibly route network traffic between network defense tools. (ii) In order to automate the deployment of dynamic defense rules, SAND needs to communicate with network defense tools. Since an SDN controller can already communicate with SDN switches, we can leverage this capability to make SAND communicate with network defense tools. (iii) The wide deployment of SDN in the real world may ease the adoption of SAND. It is worth mentioning that the SDN capabilities mentioned in (ii) and (iii) are not sufficient for SAND's purposes.

2.4.1 *SAND architecture*

Figure 6 describes the SAND architecture, which extends the SDN architecture and inherits its layers of application, control, and data. At the application layer, the defender can write both SAND apps and regular SDN apps using the northbound interfaces that are extended from SDN. Nevertheless, SAND apps are meant for generating and deploying dynamic defense rules, while SDN apps are used for managing traffic. At the control layer, the SAND orchestrator provides programmable interfaces to SAND apps, disseminates defense rules received from SAND apps to the relevant network defense tools, and reports to SAND apps the messages received from network defense tools. At the data layer, there are SAND enforcement points, each of which contains network defense tools (e.g., firewall or NIDS) and maintains a SAND communication middleware to facilitate the communication between network defense tools and the SAND orchestrator.

SAND apps. The defender writes a SAND app in response to an external or internal trigger. The app collects the relevant information corresponding to a trigger, generates new defense rules, and deploys these new defense rules to the relevant network defense tools. A SAND app can use the interfaces at the northbound of the SAND orchestrator (i.e., the northbound APIs) to deploy dynamic defense rules to the relevant network defense tools, and can receive reports (i.e., alerts or errors) from network defense tools and possibly network status (e.g., flow paths) from SDN switches.

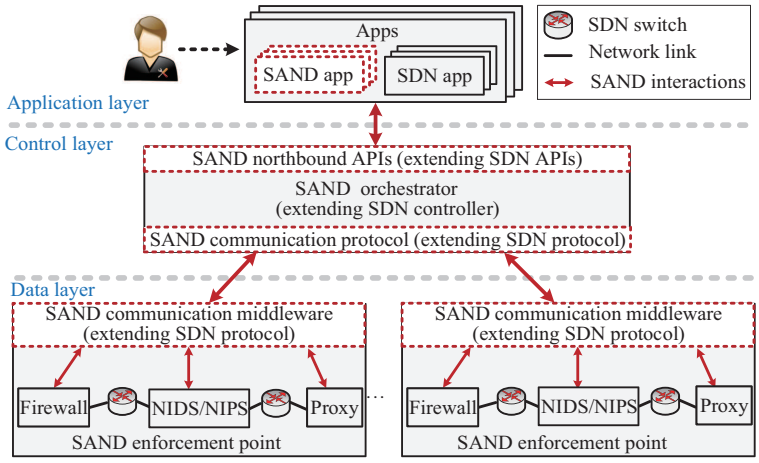

Figure 6 (Color online) The SAND architecture extends the SDN architecture, while keeping SDN's functions intact.

SAND orchestrator. The SAND orchestrator provides APIs at its northbound and communication protocols at its southbound. The SAND northbound interface extends SDN's northbound interface with new APIs while keeping SDN's northbound APIs intact (i.e., a network administrator can write programs to manage SDN networks). The defender uses the SAND northbound APIs to write SAND apps to dynamically generate and deploy new defense rules that are to be enforced at the relevant network defense tools. The SAND southbound interface facilitates the communication between the orchestrator and the network defense tools.

SAND enforcement point. A SAND enforcement point has: (i) one or multiple network defense tools, which process the network traffic forwarded by connected SDN switches; and (ii) a SAND communication middleware, which extends the SDN protocols to incorporate new functions to facilitate the secure communication between the network defense tools and the SAND orchestrator via appropriate cryptosystems (e.g., symmetric key encryption with message authentication).

2.4.2 *Why dynamic forwarding rules are not sufficient for adaptive network defense?*

SAND leverages SDN, including its dynamic forwarding rules, which however are not sufficient for adaptive defense. Although dynamic forwarding rules can instruct network traffic to go through dynamic chains of network defense tools [5,8,19], they are only able to enforce the security policies that have been set in network security tools, which we may call old or existing policies. In contrast, adaptive network defense is about enforcing new security policies, which emerge (for example) when new attacks become known by the defender; dynamic forwarding rules cannot defend against such new attacks no matter how we forward the network traffic (simply because the network defense tools cannot detect or block them yet).

3 The SAND prototype system

The SAND prototype system leverages RYU[7] (an SDN controller), Open vSwitch[8] (a virtual SDN switch), iptables (a firewall), Snort (an NIDS), and Squid (a proxy) as building-blocks. Our source code is available at the web[9].

3.1 SAND apps

We implemented three apps corresponding to the aforementioned three motivating scenarios.

7) RYU openflow controller. https://github.com/osrg/ryu/.
8) OpenVSwitch. http://openvswitch.org/.
9) Source Code of SAND Project. https://github.com/handsomeBao/SDSNF/tree/SDSNF.

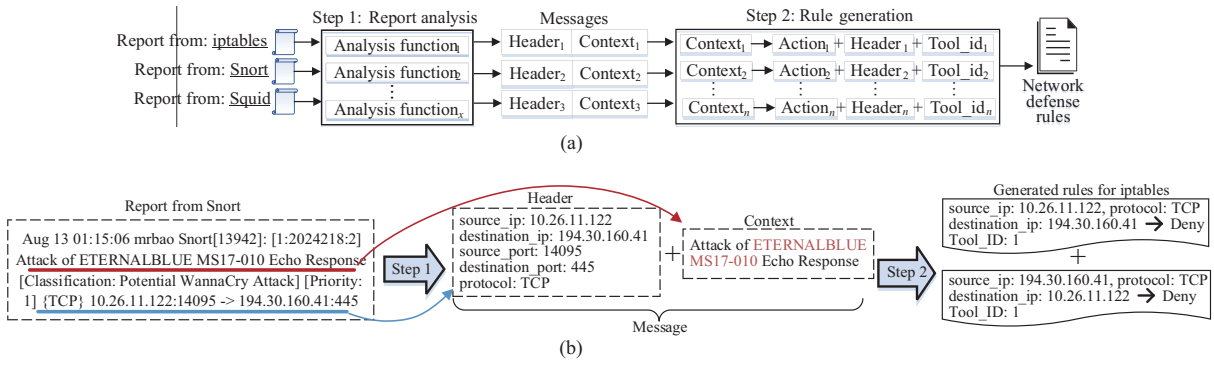

Figure 7 (Color online) An example illustrating how a SAND app generates new defense rules in response to the new threat of WannaCry. (a) A high-level description of the app's report analysis and rule generation functionalities; (b) a concrete example illustrating how the rule app analyzes a Snort report and automatically generates new rules for iptables to enforce.

3.1.1 *Programming SAND app to accommodate new security policy (motivating scenario 1)*

When the defender becomes aware of a new attack, say WannaCry, the defender can write a SAND app to adapt the network defense as follows: (i) deploy defender-defined new defense rules for the iptables to identify and mirror flows targeting port #445 to the Snort; (ii) deploy defender-defined new defense rules for the Snort to examine whether a flow (mirrored from the iptables) contains a WannaCry attack or not; and (iii) generate and deploy new defense rules to instruct the iptables to wait for the Snort's examination result and then act correspondingly (i.e., releasing/dropping the flow). In order to achieve (iii), there are two options: (1) extend both iptables and Snort to let the latter directly report its examination result to the former; (2) let the latter report to the SAND app, which then instructs the iptables to release/drop the flow. We choose option (2) for two reasons. First, Snort cannot tell which iptables forwarded the flow in question and therefore cannot tell to which iptables it should send the examination result. Second, Snort's examination result may need to be sent to a different iptables than the one that mirrored the flow or even a different defense tool because a malicious flow ideally should be dropped by the firewall closest to the source of the malicious flow.

Algorithm 1 is the pseudocode corresponding to the basic ideas discussed above. The app uses a report analysis module to extract the flow header (for identifying the flow in question) and the context (describing Snort's verdict on the flow) from the received reports (Line 4) and then analyze the context. If the analysis shows that the flow contains a WannaCry attack (Line 6), the SAND app will use the rule generation module to generate a new defense rule with respect to the flow header and iptables Tool_ID to drop the flow (Line 7), and deploy the new defense rule to the iptables to drop future packets from the attacking IP address (Line 8). The report analysis and rule generation modules are elaborated below.

Algorithm 1 SAND app for defending WannaCry attack via cooperation between iptables and Snort

Input: SAND and SAND-enabled iptables and Snort;
Output: Updated defense rules at the iptables (Tool_ID = $\mathrm{ID}_{\mathrm{iptables}}$) and Snort (Tool_ID = $\mathrm{ID}_{\mathrm{Snort}}$);
1: RuleInstall($\mathrm{ID}_{\mathrm{iptables}}$, $\mathbb{R}_{\mathrm{iptables}}$); #$\mathbb{R}_{\mathrm{iptables}}$: rules to mirror flows targeting port #445 to the Snort
2: RuleInstall($\mathrm{ID}_{\mathrm{Snort}}$, $\mathbb{R}_{\mathrm{Snort}}$); #$\mathbb{R}_{\mathrm{Snort}}$: rules for detecting the WannaCry attack.
3: **while** True **do**
4: Report ← ListenToTool($\mathrm{ID}_{\mathrm{Snort}}$);
5: flow_header, context ← ReportAnalysis(Report);
6: **if** alert of WannaCry in context **then**
7: R_{iptables} ← RuleGeneration(flow_header, $\mathrm{ID}_{\mathrm{iptables}}$, action = drop);
8: RuleInstall($\mathrm{ID}_{\mathrm{iptables}}$, R_{iptables}); #Deploy the generated rule to iptables to drop the flow.
9: **end if**
10: **end while**

Figure 7(a) presents a general description of the SAND app's report analysis module and the rule generation module. Specifically, when the report analysis module receives a report from a defense tool, the module analyzes the report to produce a message with a header (i.e., the flow header identifying the flow in question) and a flow context (describing a defense tool's annotation on the flow, such as Snort's alert indicating that the flow is malicious). Based on the flow context provided by the report analysis module, the rule generation module selects an appropriate defense tool and generates rules represented by "Action_1 + Header_1 + $\mathrm{Tool\_ID}_1$", meaning that the defense tool with $\mathrm{Tool\_ID}_1$ must impose Action_1

on every packet of a flow identified by Header_1.

Figure 7(b) presents a concrete example showing how the SAND app analyzes a Snort report and generates new rules for the iptables. Specifically, when the SAND app receives a report from the Snort, the app extracts the flow header and the context information from the report. Then, according to the context information (i.e., "Attack of ETERNALBLUE MS17-010 Echo Response" indicating WannaCry), the app interprets the report as an alert of WannaCry from the host IP 10.26.11.122 to destination IP 194.30.160.41. Finally, the app generates two rules to instruct the iptables with Tool_ID "1" to block communications between these two IP addresses.

3.1.2 *Programming SAND app to assure dynamic security policy consistence (motivating scenario 2)*

Algorithm 2 highlights the SAND app for assuring the consistence between the dynamic policies enforced at the iptables and Squid. At a high level, when new defense rules need to be deployed into the iptables (Line 1), the app extracts the destination port and the action fields from these new rules (Line 2). If the action field in a rule indicates to block the access to a website (i.e., HTTP and HTTPS, or ports #80 and #443, with the action being deny) (Line 3), the app extracts the source and destination IP addresses from the rule (Line 4) and generates a corresponding Squid rule to block the source IP address from accessing any cached content of the website corresponding to the destination IP address (Line 5), and the app deploys this new rule to the Squid (Line 6). Finally, the app deploys the given new rule to the iptables (Line 8).

Algorithm 2 SAND app for assuring dynamic policy consistence

Input: New defense rules to be deployed to the iptables;
Output: Updated defense rules at the iptables (Tool_ID = $\text{ID}_{\text{iptables}}$) and Squid (Tool_ID = ID_{Squid});
1: **for** each new rule R_{iptables} to be deployed to the iptables **do**
2: dst_port, action ← RuleParse(R_{iptables}); #Extract destination port and action for analysis.
3: **if** dst_port == (#80 or #443) AND action== deny **then**
4: src_ip, dst_ip ← RuleParse(R_{iptables}); #Extract source and destination ip for rule generation.
5: R_{Squid} ← RuleGeneration(src_ip, dst_ip, action = deny);
6: RuleInstall($\text{ID}_{\text{Squid}}, R_{\text{Squid}}$); #Deploy the generated rule for policy consistence into Squid.
7: **end if**
8: RuleInstall($\text{ID}_{\text{iptables}}, R_{\text{iptables}}$); #Deploy the iptables defense rule.
9: **end for**

3.1.3 *Programming SAND app to achieve security policy migration (motivating scenario 3)*

For the scenario, Algorithm 3 highlights the SAND app. Suppose we need to migrate a VM from Host_1 to Host_2, which are respectively protected by firewall iptables_1 and iptables_2. Suppose the IP address of the migrating VM is to be changed from IP_1 to IP_2 because of the migration. After the VM is migrated, the SAND app migrates all of the defense rules, which are related to the VM and enforced at iptables_1, from iptables_1 to iptables_2 as follows: (i) get all of the rules from iptables_1 (Line 1); (ii) identify the rules related to the VM (i.e., having IP_1 in the source or destination IP address field) (Lines 2 and 3); (iii) change IP_1 in these rules to IP_2 (Line 4) and deploy them to the iptables_2 (Line 5); and (iv) delete those rules involving IP_1 from iptables_1 (Line 6).

Algorithm 3 SAND app for achieving security policy migration

Input: IP address IP_1 of the VM before migration and its IP address IP_2 after migration;
Output: Updated defense rules in iptables_1 (Tool_ID = $\text{ID}_{\text{iptables}_1}$) and iptables_2 (Tool_ID = $\text{ID}_{\text{iptables}_2}$);
1: $\mathbb{R}^{\text{all}}_{\text{iptables}_1}$ ← GetAllRules($\text{ID}_{\text{iptables}_1}$) #Get all of the rules from iptables_1
2: **for** each rule R_{iptables_1} in $\mathbb{R}^{\text{all}}_{\text{iptables}_1}$ **do**
3: **if** R_{iptables_1}.src_ip == IP_1 OR R_{iptables_1}.dst_ip == IP_1 **then**
4: $R^{\text{updated}}_{\text{iptables}_1}$ ← Change IP_1 to IP_2 in R_{iptables_1}.src_ip and R_{iptables_1}.dst_ip
5: RuleInstall($\text{ID}_{\text{iptables}_2}, R^{\text{updated}}_{\text{iptables}_1}$); #Deploy the changed rule to iptables_2
6: RuleDelete($\text{ID}_{\text{iptables}_1}, R_{\text{iptables}_1}$); #Delete the rules involving IP_1 from iptables_1
7: **end if**
8: **end for**

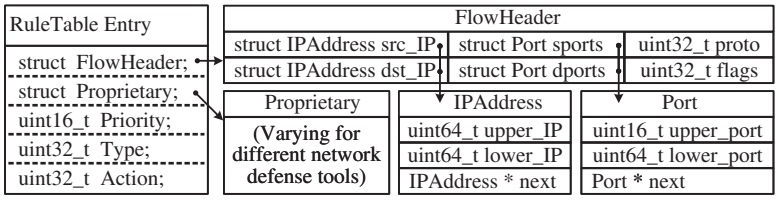

Figure 8 Data structure of a RuleTable entry.

3.2 SAND enforcement points

A SAND enforcement point contains some network defense tools and a SAND communication middleware. In practice, a SAND enforcement point can be deployed at a physical host, a virtual machine, or a container; whereas, the SAND communication middleware and defense tools run as independent software. In order to cope with the heterogeneous native rules used by defense tools, we introduce a unified rule representation and extend each network defense tool with a RuleTable to make it SAND-compatible. When writing a SAND app, the defender uses the unified rule representation to program new defense rules. When the SAND orchestrator deploys these new defense rules written in the unified rule representation, a program accompanying the RuleTable "translates" these new rules to the native representation of the network defense tool in question.

3.2.1 *Unified rule representation*

We observe that a network defense rule can have up to three elements: a match field, an action field, and an other field, despite that these fields may be located at different positions of a rule. Based on such observation, we define the unified rule representation which summarizes the native rules in five fields.

(i) `Flow Header.` This field uniquely identifies a network flow, including its source and destination IP addresses, source and destination ports, flag, and protocol. This field does not include the flow payload because some network defense tools do not process the payload. For the network defense tools that do process the payload, we put the payload in the `Proprietary Info` field.

(ii) `Proprietary Info.` This field contains the flow payload and possibly other kinds of information that is needed by some network defense tools.

(iii) `Priority.` This field indicates the execution priority of a rule. This is an important matter because different network defense tools may enforce their defense rules in different orders. For example, iptables matches flows to rules from the top to the bottom of its native rule set and stops seeking any further match once a flow is matched to a rule. We set the execution priority of a rule in the unified representation to be the same as the execution priority of the corresponding native rule of a network defense tool.

(iv) `Type.` This field specifies the type of a rule in the unified representation, which is the same as the type of the corresponding native rule.

(v) `Action.` This field indicates how a network defense tool should cope with the packets of a flow in question (e.g., dropping them or not). In our prototype, we implement the unified rule representation of the native rules used by the aforementioned three network defense tools (i.e., iptables, Snort, and Squid).

3.2.2 *RuleTable and "translator"*

Figure 8 illustrates the data structure of a RuleTable entry. Since a native rule often corresponds to a tuple of "(source IP address, destination IP address, source port, destination port)", where an IP address (or port) field can be one, multiple, or one or multiple ranges of IP addresses (port numbers), the RuleTable needs to accommodate these rich semantics. For this purpose, the IPAddress (Port) struct in the FlowHeader struct, which corresponds to the `Flow Header` field in the unified rule representation, is implemented as a linked list (via a pointer called "next"). Each element on the link corresponds to one or a range of IP addresses bounded between "upper_IP" and "lower_IP" (when these two values are equal, indicating one IP address), and multiple (ranges) of IP addresses are described by multiple elements on the linked list. The rich semantics in port numbers is accommodated in a similar fashion. The Proprietary struct, which corresponds to the `Proprietary Info` field in the unified rule representation, is implemented as a struct that varies with network defense tools.

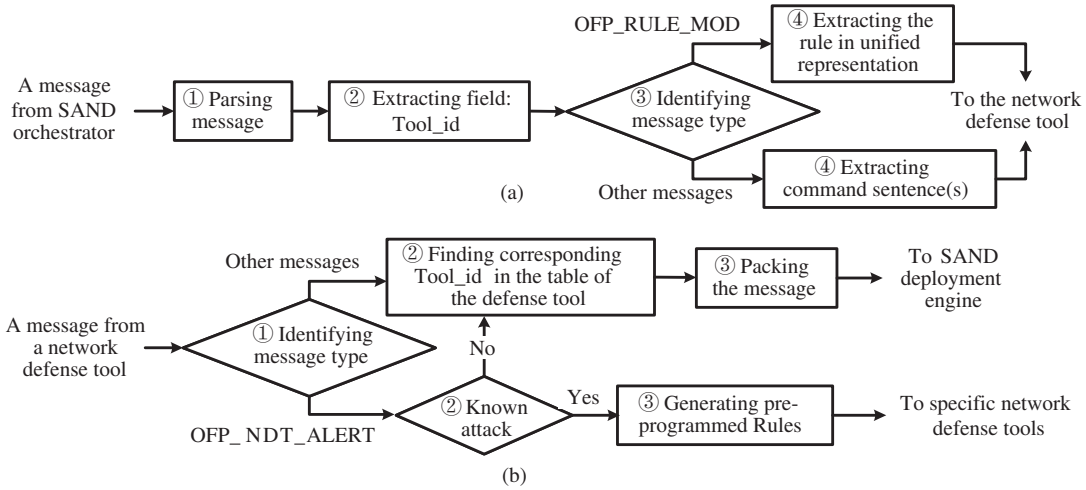

Figure 9 Function of SAND communication middleware.

A RuleTable is accompanied by a "translator" program, which maps between the rules in the unified representation and the native rules of network defense tools. In addition, this "translator" program has two more functions: (i) notifying the SAND orchestrator that an operation on RuleTable (e.g., adding, modifying, or deleting a rule) is accomplished, and (ii) executing two commands issued by the orchestrator, namely sending the rules in the RuleTable and sending the log of a network defense tool to the SAND orchestrator. We denote that the RuleTable works as an extra map of the native rules for the ease of deploying defense rules in heterogeneous network defense tools, while the defense tools still run using their native rules, which means it exerts no influence on the performance of the defense tools during the processing of network traffic.

3.2.3 *The SAND communication middleware*

We implement the middleware by extending the Open vSwitch 2.5.0, which is an open-source virtual SDN switch written in C, because we want to leverage its communication mechanism as a building-block.

The extension is in three folds. The first extension is to implement a maintenance function for bookkeeping the network defense tools at a SAND enforcement point. In the prototype, this function is implemented as a MySQL table with two attributes: Tool_ID, which uniquely identifies a network defense tool; and Port_#, which is the source port used by the network defense tool to communicate with the SAND orchestrator.

The second extension is to implement a security function, which protects the integrity of the communications (i) between the SAND orchestrator and the SAND communication middleware and (ii) between the SAND communication middleware and the network defense tools. Recall that SDN does not offer any mechanism to protect the secrecy or integrity of the communications between an SDN controller and an SDN switch (for performance reasons). Given that the SAND orchestrator and the SAND communication middleware may be often deployed at different computers, meaning that their communications can be subject to attacks, it is necessary to protect the integrity of these communications. For this purpose, we use the HMAC function in Openssl-1.0.1g. When the SAND middleware and its co-residing network defense tools are deployed on the same computer or VM, we may also use this HMAC mechanism to protect the integrity between them. However, we choose not to protect the secrecy of these communications because (i) the defense rules and the network defense tools' reports may be known to the attacker already (e.g., the defense rules for detecting WannaCry and the corresponding alerts generated by Snort) and (ii) incorporating encryption may incur a more significant performance degradation than HMAC. When the need to assure secrecy becomes apparent, symmetric key encryption can be easily incorporated to protect the secrecy of the communications.

The third extension is to implement a communication function in the middleware to enable "orchestrator to defense tools" and "defense tools to orchestrator" communications. Figure 9(a) illustrates "orchestrator to defense tools" communications. Specifically, when the orchestrator needs to communicate with a network defense tool (e.g., to update some defense rules enforced at the network defense tool), the

orchestrator sends a message to the middleware, which parses the message to extract the Tool_ID of the intended network defense tool and the message type. If the message type is `OFP_RULE_MOD`, an indicator for updating defense rules, the middleware extracts the defense new rules (in the unified representation) from the message content; otherwise, the middleware extracts the command(s) from the message and then forwards the command(s) to the intended network defense tool.

Figure 9(b) illustrates "defense tools to orchestrator" communications, while noting that the middleware distinguishes defense tools through the ports assigned to the defense tools and the orchestrator uses Tool_ID to distinguish defense tools. Specifically, when a defense tool needs to communicate with the orchestrator (e.g., notifying the orchestrator of the accomplishment of a RuleTable operation), the defense tool sends a message to the middleware through the port assigned to the defense tool at the time the tool is installed. The middleware uses this port number to identify the unique Tool_ID associated with this port number as recorded in the MySQL table mentioned above. The middleware sends the defense tool's message together with the defense tool's Tool_ID to the orchestrator. In order to reduce the delay incurred by the middleware, we propose moving some SAND app functions to the middleware to avoid unnecessary communications. As shown in Figure 9(b), when a defense tool sends an `OFP_NDT_ALERT` message to the middleware, the middleware checks whether or not the alert contains a known attack. If so, the middleware can directly generate some defender-programmed defense rules and sends them to the relevant defense tools for enforcement (e.g., blocking the relevant packets).

3.3 SAND orchestrator

The SAND orchestrator has a northbound interface and a southbound interface. The northbound interface is used by the defender to write SAND apps. This interface extends SDN's northbound APIs with the following SAND-specific APIs: (i) NDT_RULE_MOD, which is used to add, delete, or modify defense rules enforced at defense tools; (ii) NDT_ALL_RULES_REQUEST, which is used to fetch defense rules enforced at defense tools; and (iii) EVENT_LISTEN_NDT, which is used by a SAND app to instruct the SAND orchestrator to provide reports (e.g., alerts or errors) from defense tools.

The southbound interface supports secure communications between the SAND orchestrator and defense tools. It extends SDN's OpenFlow protocol[10)] by introducing two types of SAND-specific messages. First, the SAND-to-NDT type includes 8 messages, 5 of which are used by the SAND orchestrator to communicate with defense tools and the other 3 are used by defense tools to report to the SAND orchestrator after accomplishing some activities. Second, we introduce two SAND asynchronous messages, which are used by defense tools for reporting to the SAND orchestrator. These two messages, dubbed `OFP_NDT_ALERT` and `OFP_NDT_ERROR`, may be reminiscent of SDN's asynchronous messages (for SDN switches to report to an SDN controller), but are actually different. This is because SDN's buffering mechanism processes messages sequentially (i.e., a new message is not read until the previous message has been processed), which is not adequate for SAND because messages from defense tools can be in large volumes and can be lost when the buffer is full. This explains why we use a special buffer at the SAND orchestrator and why we let a dedicated thread manage the special buffer (i.e., the thread only listens for these two messages). These messages are further discussed in Appendix A.

4 Evaluating the SAND prototype

Figure 10 describes the network environment that is common to our experiments of the three motivating scenarios. We use two servers, Sever_1 and Sever_2. Each server has a 16-core Xeon 2.4 GHz, 128 GB RAM and two 1 Gbps network interfaces and runs Ubuntu 16.04. Each server runs MaxiNet 1.1 [23] so that the two servers formulate an SDN network. Each server runs 4 virtual SDN switches. Server_1 has 7 VMs (i.e., VM_1 to VM_7) that run Linux Ubuntu 14.04, VM_7 runs a SAND orchestrator and SAND apps, and each of the other 6 VMs runs as a SAND enforcement point, which contains the aforementioned SAND-enabled iptables-v1.4.21 (Ubuntu 14.04), Snort-2.9.8.0, and Squid-2.7.STABLE9 as well as the SAND communication middleware. Server_2 runs 6 VMs (i.e., VM_8 to VM_{13}), each of which runs as a SAND enforcement point that is the same as those in Sever_1. Scenario-specific configurations are described later.

10) OpenFlow specification v1.3. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onfspecifications/openflow/openflowspec-v1.3.0.pdf.

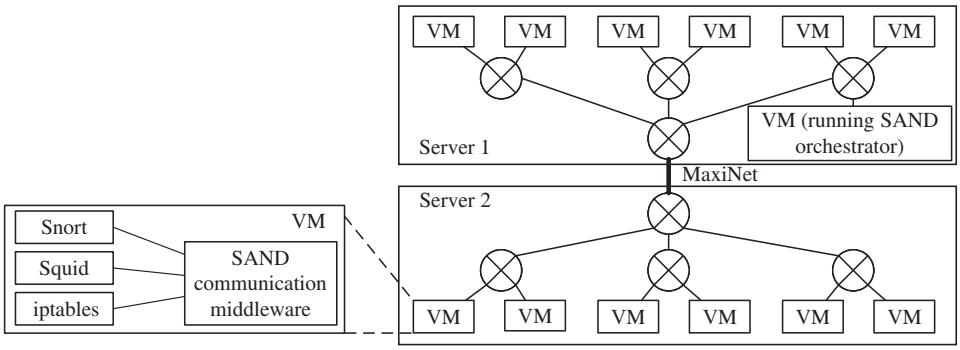

Figure 10 The network topology for our experiments.

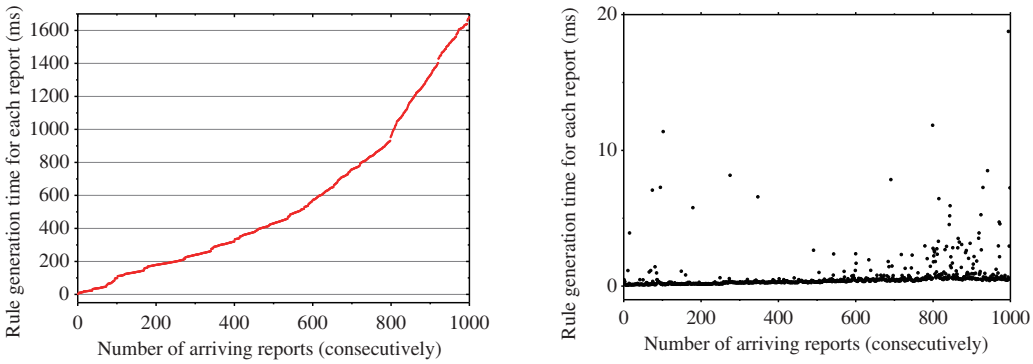

Figure 11 (Color online) Time for automated generation of iptables rules upon the SAND app receiving the Snort's reports. (a) Consecutive Snort alert arrives; (b) intermittent Snort alert arrives.

4.1 Compatibility analysis

Compatibility is evaluated in two aspects. First, SAND's unified rule representation can accommodate heterogeneous native rules of iptables, Snort, and Squid. Second, SAND incurs small modifications: the SAND orchestrator extends the RYU controller with an extra of 1.1k lines of code (LOC) in Python; the SAND communication middleware extends the Open vSwitch with an extra 1.8k LOC in C; the RuleTable only incurs an extra 2.7k LOC (or 6.43% of the original 42k codebase) in iptables, an extra 3.5k LOC (or 1.04% of the original 336k codebase) in Snort, and an extra 1.5k LOC (or 0.69% of the original 216k codebase) in Squid.

4.2 Agility analysis

4.2.1 *Measuring agility in motivating scenario 1*

In our experiments with this scenario, we further run the SAND-enabled Snort and iptables in VM_1. The SAND app runs in VM_7 to receive reports from the Snort and generate new defense rules for the iptables.

First, we measure the rule generation time. We consider two experiments: (i) the Snort generates 1000 reports consecutively (within 50 ms); (ii) the Snort generates 1000 reports intermittently at the interval of 10 ms (i.e., 100 reports per second for 10 s). Figure 11(a) plots the result in experiment (i). We observe that the rule generation time for each report increases sharply, as high as 1.5 s in the end; this is because the RYU controller, which underlies the SAND orchestrator, uses a buffer to queue Snort reports before releasing them to the SAND app sequentially. This may cause significant delays when there is a burst in reports arriving, as shown in experiment (i). In order to measure the genuine rule generation time (while excluding the impact of specific controller mechanisms), we carry out experiment (ii). Figure 11(b) plots the result in experiment (ii). We observe that the rule generation time is less than 5 ms in most cases (despite slight fluctuation), implying agility; this speedup comes from the fact that there is no waiting time at the RYU controller for releasing Snort reports to the SAND app.

Second, we measure the rule deployment time, including the time for deploying defense rules in the

Table 1 Time for deploying new rules (unit: ms)

Tool	Adding	Deleting	Modifying	Average	Integrity (%)
Squid	3.21	3.45	4.07	3.58	30.68
iptables	12.16	13.67	14.33	13.39	14.62
Snort	11.98	12.28	14.08	12.78	17.89

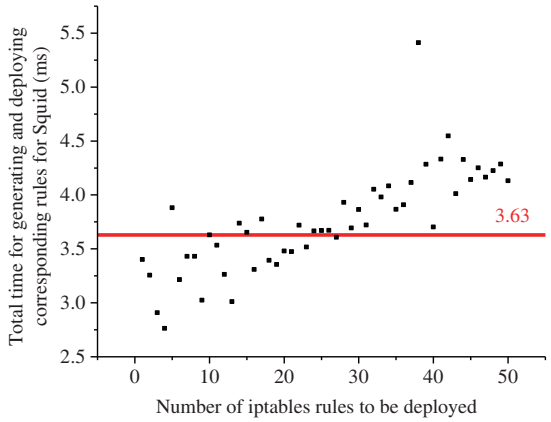

Figure 12 (Color online) Time for policy consistence.

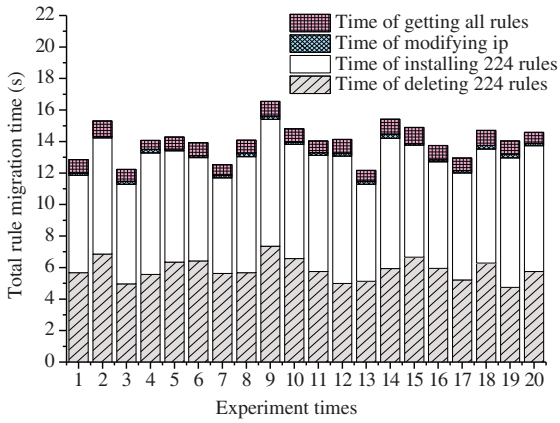

Figure 13 (Color online) Rule migration time.

unified representation into the RuleTable and the time for translating them into native rules of the defense tools by the program accompanying the RuleTable. In this experiment, we run the three network defense tools (i.e., iptables, Snort, and Squid) in VM_5 and the SAND app in VM_7. These new defense rules are deployed by the SAND app. Table 1 summarizes the results averaged over 100 independent experiment runs. The last column represents, on average, the percentage of the amount of time spent on protecting the communication integrity both between the SAND orchestrator and the SAND communication middleware and between the SAND communication middleware and the defense tools. We observe that the average rule deployment time for iptables and Snort is 2×–3× longer than that of Squid, which can be explained as follows. On one hand, the native rules of iptables reside in the user space, while iptables itself runs in the kernel space, meaning that updating a RuleTable in the user space (to avoid re-compiling the kernel) incurs an update to some native rule(s) in the kernel space, which incurs communications with the Linux kernel and causes delays. On the other hand, Snort uses complicated rules because it needs to analyze the traffic payload, which incurs much time delay.

4.2.2 *Measuring agility in motivating scenario 2*

In this experiment, we have the SAND-enabled iptables (running in VM_2), which allows anyone to access a website at a specific IP address. Suppose the new defense rules, which are to be enforced at the iptables, are to deny access to the website from 50 specific IP addresses. The SAND app (running in VM_7) not only needs to deploy these 50 rules to the iptables, but also needs to generate their corresponding defense rules for the SAND-enabled Squid (also running in VM_2) to enforce. Figure 12 plots the time of generating and deploying new defense rules for the Squid to enforce; these two are measured together because they both are small. On average, it takes 3.63 ms for the SAND app to generate and deploy a defense rule for the Squid, which demonstrates the agility of SAND.

4.2.3 *Measuring agility in motivating scenario 3*

In our experiment, the SAND app (running in VM_7) migrates the defense rules enforced by the SAND-enabled iptables (running in VM_5) to the SAND-enabled iptables (running in VM_6). Figure 13 plots the generating and deploying time it takes to migrate 224 rules, which happens to be the case in the example. The rule generation time refers to the total time for the app to get all rules from $iptables_1$ and to update the IP addresses, while the deployment time refers to the total time for the app to deploy the updated rules into $iptables_2$ and to delete the migrated rules from $iptables_1$. On average (over 20 experiment runs), it takes 0.82 s to get all of the relevant rules from $iptables_1$ and 0.11 s to update the IP addresses,

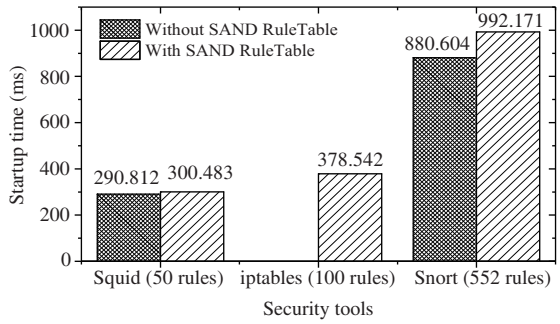

Figure 14 Defense tools' loading time.

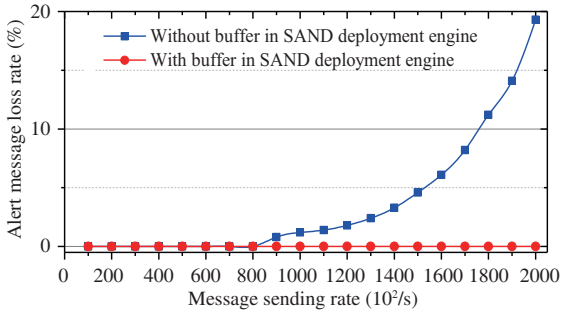

Figure 15 (Color online) Loss rate of messages from defense tools to SAND orchestrator: with vs. without using buffer at SAND orchestrator (x-axis unit: 100 messages per second).

meaning 0.93 s for generation. It takes 6.90 s to deploy the 224 rules to iptables_2 and 5.87 s to delete the 224 rules from iptables_1, meaning 12.77 s for deployment. Therefore, we can claim that the SAND app for achieving policy migration is agile.

4.3 Efficiency analysis

We extend the three defense tools (i.e., iptables, Snort, and Squid) with a RuleTable and measure the time for loading it. In our experiment, we run the three network defense tools in VM_8. We measure the loading time of the RuleTable for the three network defense tools. Figure 14 plots the loading time of the three defense tools, averaged over 30 independent experiment runs, with vs. without SAND-enabled. For iptables, the loading time is applicable only to SAND-enabled iptables because standard iptables resides in the Linux kernel (rather than the user space), while recalling that the RuleTable resides in the user space. We observe that the average loading time, including the initialization of a RuleTable with 100 rules, is smaller than 400 ms, which is small. The extra loading time incurred by SAND for the other two tools is even smaller.

4.4 Security analysis

First, security of rule generation requires the following: SAND apps are not vulnerable (otherwise, the report analysis or rule generation module can be compromised); SAND orchestrator is not vulnerable (otherwise, SAND apps running on top of it can be compromised).

Second, security of rule deployment requires the following: the SAND orchestrator is not vulnerable (otherwise, the deployment process can be compromised); the defense tools are not vulnerable (otherwise, the deployment process can be compromised). Under these assumptions, cryptographic keys used by these end points for assuring the integrity of messages are assured, so do the integrity of the new rules and messages that are protected by them.

Third, we consider robustness against denial-of-service attacks. For this purpose, we show that SAND can deal with bursts of deployment-incurred messages via the buffer at the SAND orchestrator. We use the experiment environment described in Figure 10 to measure the degree at which SAND can tolerate, with a buffer of 16 MBytes in the SAND orchestrator, bursts of messages generated by defense tools for reporting to the SAND orchestrator. In our first experiment, we let one SAND-enabled iptables instance in VM_{10} generate and send 50000 messages consecutively to the SAND orchestrator in VM_7. We observe no message loss because one iptables can only generate no more than 35000 messages per second (in our setting). In our second experiment, we let eight SAND-enabled iptables instances (i.e., VM_1 to VM_4 in Server_1 and VM_{10} to VM_{13} running in Server_2), simultaneously generate and send messages. Figure 15 plots the results and shows that (i) the SAND orchestrator without using a buffer is able to process at most 80000–90000 messages per second and (ii) the loss rate increases rapidly once the message rate goes above 90000 messages per second; in contrast, the use of the buffer leads to no message loss even when the message rate reaches 200000 messages per second. This not only justifies our use of buffers, but also highlights that the defender needs to be aware of this matter when allocating resources to the SAND orchestrator (i.e., allocation of RAM).

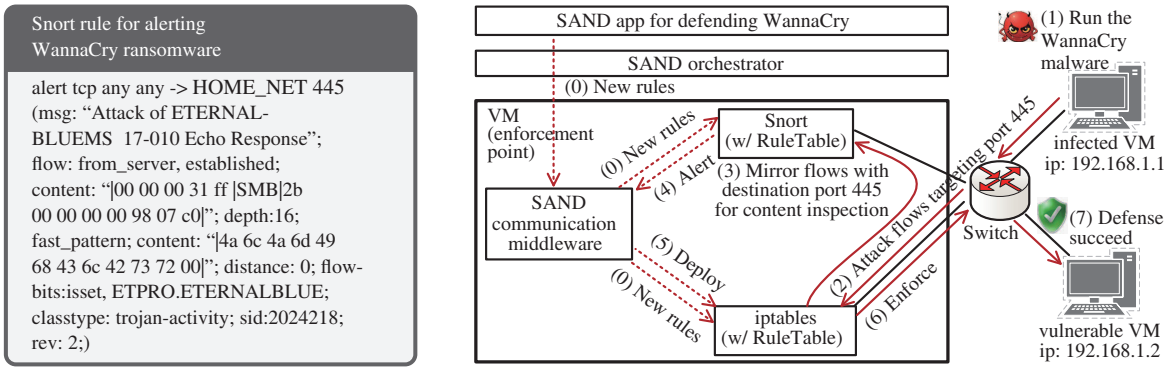

Figure 16 (Color online) Experiment showing SAND blocks WannaCry.

4.5 Effectiveness analysis

4.5.1 *Effectiveness in accommodating new security policy (motivating scenario 1)*

In this experiment, the defender, upon becoming aware of WannaCry, writes a SAND app to add the following Snort rule to detect WannaCry attacks[11)].

Figure 16 describes the experiment with VM_1 and two computers, which are connected via an SDN switch. Both computers run Windows XP SP3 without patching MS17-010 and with port #445 open (i.e., vulnerable to WannaCry). VM_1 runs SAND-enabled iptables and Snort to examine the traffic to these two computers and runs a SAND communication middleware. The experiment environment is isolated from the Internet for ethical and security reasons. The experiment proceeds as follows: (0) When becoming aware of the WannaCry attack, the defender writes the aforementioned SAND app and runs it in VM_7 to add a rule to the iptables' RuleTable to instruct the iptables to mirror flows targeting port #445 to the Snort. For performance optimization and as discussed in Subsection 3.2.3, we run the app's function for processing the reports received from the Snort at the SAND communication middleware. (1) Run the WannaCry malware in the computer with IP address 192.168.1.1, which is immediately infected (i.e., showing the ransom window). (2) When the infected computer (192.168.1.1) attempts to infect the vulnerable computer (192.168.1.2), the iptables sees the flow targeting destination port #445. (3) The iptables lets the flow pass (i.e., the iptables is used for intrusion detection rather than intrusion prevention) while mirroring the flow to the Snort. (4) The Snort examines the flow, detects that the flow contains the WannaCry attack, generates an alert, and sends the alert to the SAND communication middleware. (5) The middleware parses the alert and generates a new defense rule to be enforced by the iptables to block any further traffic originating from the infected computer (192.168.1.1). (6) The iptables enforces the new defense rule. (7) In about 2 min, we see an iptables log entry showing that it blocks a WannaCry attack from the infected computer (192.168.1.1) to the vulnerable computer (192.168.1.2) and a Snort log entry showing an alert corresponding to this attempted attack. In multiple runs of the same experiment, we consistently observe the 2-minute delay, which appears to be inherent to the WannaCry malware sample we use. Our experiments show that SAND successfully protects the vulnerable computer (192.168.1.2) from WannaCry because the generation and deployment of new rules take no more than 20 ms, meaning that the adaptive defense takes effect sooner than the 2-minute delay of the WannaCry malware's attempt at attacking the vulnerable computer (192.168.1.2).

4.5.2 *Effectiveness in assuring dynamic security policy consistence (motivating scenario 2)*

Figure 17 describes the setting for measuring the effectiveness in assuring dynamic policy consistence. First, we confirm the security policy inconsistence problem via the following experiment: let a browser running in VM_2 access a target website to cause the Squid to cache the website content; manually install a defense rule to the iptables to block VM_2 from accessing the website; observe that VM_2 still can access the website's content cached in the Squid. Second, we confirm SAND can assure the security policy consistence via the following experiment: (1) run SAND-enabled iptables and Squid in VM_2; (2) run the aforementioned SAND app for the motivating scenario 2 in the VM_7 to assure the consistence between the

11) EternalBlue Snort Rule. https://securingtomorrow.mcafee.com/executive-perspectives/analysis-wannacry-ransomware-outbreak/.

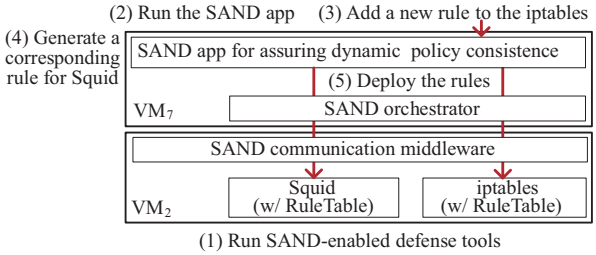

Figure 17 (Color online) Experiment for assuring policy consistence.

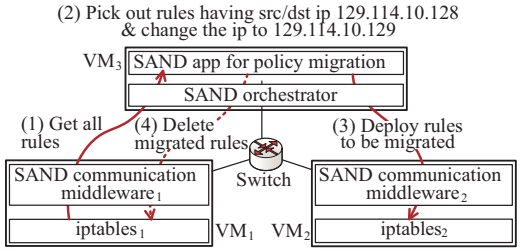

Figure 18 (Color online) Experiment for iptables rule migration.

policies enforced by the iptables and Squid; (3) add a new defense rule to the iptables to block the VM_2 from accessing the website (i.e., set the source IP address to be the IP address of VM_2, the destination IP address to be the website, the destination port to be 80 and the action to be REJECT); (4) the SAND app, upon receiving the new defense rule for the iptables, generates a corresponding defense rule for the Squid to block the VM_2 from accessing the website content cached at the Squid; (5) Deploy the new defense rules to the iptables and the Squid, respectively. Our experiment confirms that VM_2 can neither access the website nor the cached content at the Squid.

4.5.3 *Effectiveness in achieving security policy migration (motivating scenario 3)*

Figure 18 describes the setting for measuring the effectiveness in achieving security policy migration. Suppose a VM, which has IP address 129.114.10.128 and runs on one computer (not shown in Figure 10) and is protected by the $iptables_1$ running in VM_5 (in Figure 10), needs to be migrated to another computer with IP address 129.114.10.129 and then should be protected by the $iptables_2$ running in VM_6 (in Figure 10). We confirm that SAND can achieve security policy migration as follows: (1) run SAND-enabled iptables in VM_5 and VM_6; (2) run the SAND app for achieving security policy migration in VM_7 to use the aforementioned NDT_ALL_RULES_REQUEST command to get the rules enforced by $iptables_1$; (3) identify the defense rules that have the source or destination IP address 129.114.10.128 and update the address to 129.114.10.129; (4) run the SAND app to use the aforementioned NDT_RULE_MOD command to deploy the updated defense rules to $iptables_2$; (5) run the SAND app to instruct $iptables_1$ to delete the defense rules involving IP address 129.114.10.128 because they have been migrated; (6) check the rules in $iptables_1$ and $iptables_2$ to confirm that (i) SAND respectively migrates the defense rules involving IP address 129.114.10.128 from $iptables_1$ to the defense rules involving IP address 129.114.10.129 in $iptables_2$ and (ii) the migrated rules are deleted from $iptables_1$.

5 Limitations

The present study has some limitations. (i) From a functionality point of view, SAND only achieves semi-automation because defenders have to write apps to generate new defense rules, while recalling that automation cannot be achieved before addressing the difficulties described in Subsection 2.3. (ii) From a methodological point of view, the present study focuses on demonstrating the feasibility of SAND via three defense tools. As discussed in Appendix B, we also examined four other network defense tools, namely pfSense[12] and IPFire[13] (firewalls), Suricata[14], and FreeWAF[15] (web application firewall), and observed that the unified rule representation can accommodate their native rules. Still, we may need to extend the unified rule representation to accommodate other defense rules (e.g., those using scripting languages[16][17]). (iii) From an implementation point of view, SAND extends SDN and therefore inherits its limitations, such as the delay from an SDN controller to the SDN data plane. (iv) From a management point of view, complex defense rules can cause conflicts between them, which can be resolved by leveraging

12) pfSense — World's Most Trusted Open Source Firewall. https://www.pfsense.org/.
13) www.ipfire.org — Welcome to IPFire. https://www.ipfire.org/.
14) Suricata: Opensource IDS/IPS/NSM engine. https://suricata-ids.org/.
15) High-performance WAF built on the OpenResty stack. https://github.com/wangfakang/FreeWAF.
16) Nginx—High performance load balancer, Web server and Reverse Proxy. https://www.nginx.com/.
17) HAProxy — The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/.

techniques such as [24–27]. (v) From a practical point of view, SAND assumes that the triggers are true, which may or may not hold in general. This can be mitigated by only dealing with the trusted triggers.

6 Conclusion

We have motivated and presented SAND as the first step towards fully automated network defense. We introduced the SAND architecture for semi-automated adaptive network defense via programmable generation and deployment of dynamic network defense rules. We reported a SAND prototype implementation by extending both the northbound and southbound interfaces of SDN. For this purpose, we introduced a unified representation of network security rules that can be automatically mapped to native network security rules used by some network security tools; this unified representation would be of independent value because it can be extended to accommodate all of the kinds of network defense tools that are used in practice. Experiment results show that SAND can indeed automate the generation and deployment of dynamic network security rules in the defense for the motivating scenarios. We hope SAND will inspire more research towards achieving full-fledged automated cyber defense. Towards this goal, the limitations discussed in Section 5 represent some outstanding open problems for future research.

Acknowledgements This work was supported by Key Program of National Science Foundation of China (Grant No. U1936211), Shenzhen Fundamental Research Program (Grant No. JCYJ20170413114215614), and Key-Area Research and Development Program of Guangdong Province (Grant No. 2019B010139001).

References

1 Zhang M H, Li G Y, Wang S C, et al. Poseidon: mitigating volumetric DDoS attacks with programmable switches. In: Proceedings of the 27th Annual Network and Distributed System Security Symposium, San Diego, 2020
2 Kang Q, Xue L, Morrison A, et al. Programmable in-network security for context-aware BYOD policies. In: Proceedings of the 29th USENIX Security Symposium, 2020. 595–612
3 Sebastián E, Lewis G A, Grabowski C, et al. KalKi: a software-defined IoT security platform. In: Proceedings of the 6th IEEE World Forum on Internet of Things, New Orleans, 2020. 1–6
4 McCormack M, Vasudevan A, Liu G Y, et al. Towards an architecture for trusted edge IoT security gateways. In: Proceedings of the 3rd USENIX Workshop on Hot Topics in Edge Computing, 2020
5 Yu T L, Fayaz S K, Collins M, et al. PSI: precise security instrumentation for enterprise networks. In: Proceedings of Network and Distributed System Security Symposium, San Diego, 2017
6 Zou C C, Duffield N, Towsley D, et al. Adaptive defense against various network attacks. IEEE J Sel Areas Commun, 2006, 24: 1877–1888
7 Li M H, Li M. An adaptive approach for defending against DDoS attacks. Math Problems Eng, 2010, 2010: 1–15
8 Fayaz S K, Tobioka Y, Sekar V, et al. Bohatei: flexible and elastic DDoS defense. In: Proceedings of USENIX Security Symposium, Washington, 2015. 817–832
9 Cho J H, Sharma D P, Alavizadeh H, et al. Toward proactive, adaptive defense: a survey on moving target defense. IEEE Commun Surv Tut, 2020, 22: 709–745
10 Xu S H, Lu W L, Xu L, et al. Adaptive epidemic dynamics in networks. ACM Trans Auton Adapt Syst, 2014, 8: 1–19
11 Huang L N, Zhu Q Y. Strategic learning for active, adaptive, and autonomous cyber defense. In: Adaptive Autonomous Secure Cyber Systems. Cham: Springer, 2020. 205–230
12 Huang L N, Zhu Q Y. Adaptive strategic cyber defense for advanced persistent threats in critical infrastructure networks. SIGMETRICS Perform Eval Rev, 2019, 46: 52–56
13 Mijumbi R, Serrat J, Gorricho J L, et al. Network function virtualization: state-of-the-art and research challenges. IEEE Commun Surv Tut, 2016, 18: 236–262
14 Seungwon S, Phillip A P, Vinod Y, et al. FRESCO: modular composable security services for software-defined networks. In: Proceedings of Network and Distributed System Security Symposium, San Diego, 2013
15 Hu H X, Han W, Ahn G-J, et al. FLOWGUARD: building robust firewalls for software-defined networks. In: Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking, Chicago, 2014. 97–102
16 Deng J, Hu H X, Li H D, et al. VNGuard: an NFV/SDN combination framework for provisioning and managing virtual firewalls. In: Proceedings of IEEE Conference on Network Function Virtualization and Software Defined Networks, San Francisco, 2015. 107–114
17 Deng J, Li H D, Wang K C, et al. On the safety and efficiency of virtual firewall elasticity control. In: Proceedings of Network and Distributed System Security Symposium, San Diego, 2017
18 Xia M, Shirazipour M, Zhang Y, et al. Optical service chaining for network function virtualization. IEEE Commun Mag, 2015, 53: 152–158
19 Fayaz S K, Luis C, Vyas S, et al. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In: Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, Seattle, 2014. 543–546
20 Amann J, Sommer R. Providing dynamic control to passive network security monitoring. In: Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses, Kyoto, 2015. 133–152
21 Durumeric Z, Kasten J, Adrian D, et al. The matter of heartbleed. In: Proceedings of the 2014 Internet Measurement Conference, Vancouver, 2014. 475–488
22 Ramaswamy C. Secure virtual network configuration for virtual machine (VM) protection. NIST Special Publ, 2016, 800: 125B
23 Wette P, Dräxler M, Schwabe A. MaxiNet: distributed emulation of software-defined networks. In: Proceedings of IFIP Networking Conference, Trondheim, 2014. 1–9

24 Horn A, Kheradmand A, Prasad M R. Delta-net: real-time network verification using atoms. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, Boston, 2017. 735–749
25 Chen F, Liu A X, Hwang J H, et al. First step towards automatic correction of firewall policy faults. ACM Trans Auton Adapt Syst, 2012, 7: 1–24
26 Panda A, Lahav O, Argyraki K J, et al. Verifying reachability in networks with mutable datapaths. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, Boston, 2017. 699–718
27 Stoenescu R, Popovici M, Negreanu L, et al. SymNet: scalable symbolic execution for modern networks. In: Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, 2016. 314–327

Appendix A SAND-specific messages in the extended OpenFlow protocol

Table A1 Summary of the two types of SAND-specific new messages introduced at the orchestrator's southbound interface

Message type	Message name		Purpose of a message
SAND-to-NDT	OFPRT_ALL_RULES_REQUEST		Request the defense rules of a defense tool.
	OFPRT_ALL_RULES_REPLY		Reply to the SAND orchestrator with the requested defense rules.
	OFPRT_LOG_REQUEST		Request the logs of a defense tool.
	OFPRT_LOG_NDT_REPLY		Reply to the SAND orchestrator with the requested logs.
	OFPRT_RULE_MOD	OFPRM_ADD	Add a new rule to the RuleTable and map it to a native rule of a defense tool.
		OFPRM_MODIFY	Modify a rule in the RuleTable and map it to a native rule of a defense tool.
		OFPRM_DELETE	Delete a rule from the RuleTable and map it to the native rule set of a defense tool.
	OFPRM_NDT_REPLY		Reply to the SAND orchestrator that the defense rules have been updated.
SAND asynchronous	OFP_NDT_ALERT		A defense tool alerts the SAND orchestrator of a threat.
	OFP_NDT_ERROR		A defense tool reports runtime errors to the SAND orchestrator.

Table A1 summarizes the two types of SAND-specific messages that are introduced at the orchestrator's southbound interface: SAND-to-NDT messages (SAND orchestrator to defense tools) and SAND asynchronous messages (defense tools to SAND orchestrator).

Appendix B Examination on accommodation of unified rule representation in open-source network defense tools

Table B1 Summary of the 7 network defense tools whose native rules are accommodated by the unified rule representation

Category	Name	Unified rule representation							
		Basic header				Priority	Proprietary	Type	Action
		src/dest ip	src/dest ports	flag	protocol				
Firewall	iptables	✓	✓	–	✓	✓	Table name	Chain name	✓
	pfSense	✓	✓	Disable	✓	✓	Extra options	Interface	✓
	IPFire	✓	✓	–	✓	✓	Additional settings	FW/NAT	✓
IDS	Snort	✓	✓	Operator	✓	✓	Rule options	–	✓
	Suricata	✓	✓	Operator	✓	✓	Rule options	–	✓
Proxy	Squid	✓	✓	–	✓	✓	Object name	Acl/Http_access	✓
WAF	FreeWAF	✓	–	–	✓	✓	Name, Sub-policy	Default or not	–

In the main body of the paper, we reported that the unified rule representation can accommodate the heterogeneous rules used by three network defense tools, namely iptables, Snort, and Squid. In order to see the applicability of the unified rule representation, we also examined four other network defense tools. In total, we investigated the native rule representations of 3 firewalls (i.e., iptables, pfSense, and IPFire), 2 NIDSes (i.e., Snort and Suricata), 1 proxy (i.e., Squid), and 1 Web Application Firewall or WAF (i.e., FreeWAF). Table B1 summarizes how the native rules of these 7 network defense tools can be accommodated by the unified rule representation.