Software-Defined Firewall: Enabling Malware Traffic Detection and Programmable Security Control

Shang Gao
Department of Computing
The Hong Kong Polytechnic University
cssgao@comp.polyu.edu.hk

Zecheng Li
Department of Computing
The Hong Kong Polytechnic University
cszcli@comp.polyu.edu.hk

Yuan Yao
Northwestern Polytechnical University & The Hong Kong Polytechnic University
yaoyuan@nwpu.edu.cn

Bin Xiao
Department of Computing
The Hong Kong Polytechnic University
csbxiao@comp.polyu.edu.hk

Songtao Guo
College of Electronic and Information Engineering
Southwest University
stguo@swu.edu.cn

Yuanyuan Yang
Department of Electrical and Computer Engineering
Stony Brook University
yuanyuan.yang@stonybrook.edu

ABSTRACT

Network-based malware has posed serious threats to the security of host machines. When malware adopts a private TCP/IP stack for communications, personal and network firewalls may fail to identify the malicious traffic. Current firewall policies do not have a convenient update mechanism, which makes the malicious traffic detection difficult.

In this paper, we propose Software-Defined Firewall (SDF), a new security design to protect host machines and enable programmable security policy control by abstracting the firewall architecture into control and data planes. The control plane strengthens the easy security control policy update, as in the SDN (Software-Defined Networking) architecture. The difference is that it further collects host information to provide application-level traffic control and improve the malicious traffic detection accuracy. The data plane accommodates all incoming/outgoing network traffic in a network hardware to avoid malware bypassing it. The design of SDF is easy to be implemented and deployed in today's network. We implement a prototype of SDF and evaluate its performance in real-world experiments. Experimental results show that SDF can successfully monitor all network traffic (i.e., no traffic bypassing) and improves the accuracy of malicious traffic identification. Two examples of use cases indicate that SDF provides easier and more flexible solutions to today's host security problems than current firewalls.

CCS CONCEPTS

• **Security and privacy** → **Firewalls**; *Malware and its mitigation*;

ASIACCS'18, Incheon, Republic of Korea

DOI: 10.1145/XXXXXX.XXXXXX

KEYWORDS

Malicious traffic detection, software-defined networks, software-defined firewall, network programmability.

ACM Reference format:
Shang Gao, Zecheng Li, Yuan Yao, Bin Xiao, Songtao Guo, and Yuanyuan Yang. 2018. Software-Defined Firewall: Enabling Malware Traffic Detection and Programmable Security Control. In *Proceedings of ACM Asia Conference on Computer and Communications Security, Incheon, Republic of Korea, June 4–8, 2018 (ASIACCS'18),* 12 pages.
DOI: 10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Malicious software (malware) has become one of the most serious threats to host machine security. Today's malware needs network connections to conduct malicious activities (e.g. flooding packets, leaking private data, and downloading malware updates). To detect these malicious activities, security companies have proposed security solutions on both host side (personal firewalls such as Microsoft Windows firewall and anti-viruses) and network side (network firewalls such as intrusion detection systems and ingress filtering). However, when malware lies in a lower layer than the personal firewalls, this malicious traffic becomes invisible to personal firewalls. Though network firewalls can capture all traffic, a lack of host information can make them fail to differentiate malicious traffic from other benign traffic. A typical example is the *Rovnix* bootkit [20] that can bypass the monitoring of a personal firewall via a private TCP/IP stack. Mixed with benign traffic, the network firewall may also fail to identify its traffic when *Rovnix* does not have significant features in the attack signature database, as depicted in Fig. 1.

Many solutions have been proposed for malware pattern analysis and dynamic security policy update [7, 9, 10, 15, 17]. Perdisci *et.al.* present a network-level behavioral malware clustering system by analyzing the structural similarities among malicious HTTP traffic traces generated by HTTP-based malware [17]. Amann *et.al.* propose a novel network

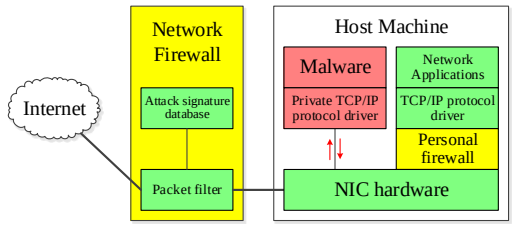


Figure 1. Personal and network firewalls may fail to identify malicious traffic when malware uses a private TCP/IP stack.

control framework that provides passive network monitoring systems with a flexible and unified interface for active response [1]. The high programmability in software-defined networking (SDN) also introduces security innovations. FlowGuard [10] enables both accurate detection and effective resolution of firewall policy violations in OpenFlow networks. Another approach, PBS [7], evaluates the idea in SDN to enable fine-grained, application-level network security programmability for mobile apps and devices. PBS introduces a more flexible way to enforce security policies by applying the concept of SDN. However, these approaches may incur high false-positive rate in attack traffic identification with no reference to host information or can be bypassed when malware adopts mechanisms to avoid personal firewall check (e.g., via a private TCP/IP stack).

To address the problem of reliable malicious traffic detection, we propose software-defined firewall (SDF), a new architecture that can prevent malicious traffic bypassing to enhance the security of host machines. The new architecture of SDF can be witnessed from its design of "control plane" and "data plane" as in SDN. The "control plane" in SDF collects host information (e.g., task names, CPU and memory utilizations of tasks) to improve the accuracy of malicious traffic detection and provides fine-grained flow management. The data plane monitors both incoming and outgoing traffic in a network hardware. The two-layer design in SDF can successfully avoid malware bypassing by integrating the host information. Another salient feature of SDF is its high programmability and application-level traffic control. Based on [7], we design a programmable language for SDF to allow users to develop control apps, through which the control plane of SDF can install rules on the data plane to manage network traffic. Thus, users can dynamically update host machine security policies, and achieve timely and precise malicious traffic filtering.

SDF is also robust to different attacks against its control plane. We leverage an audit server to avoid compromised control plane or malware installing illegal rules and removing legal rules on the data plane. When attacks are detected, the audit server will alert the network administrators about the abnormal events. With these alerts, network administrators can further check the host machine to remove the malware. SDF is easy to implement and can be deployed in either traditional or OpenFlow networks without many changes of the existing network framework. With the assist of SDF, many today's security solutions can be simplified by applying different control apps.

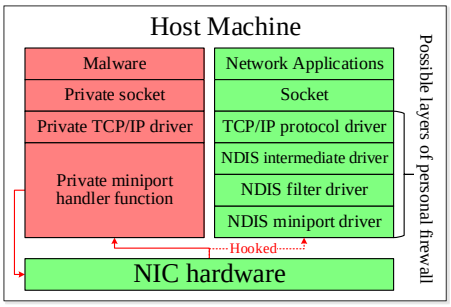


Figure 2. Malware bypasses a personal firewall.

Our main technical contributions on protecting host machine security are as follows:

•*Novel Architecture.* We propose a novel firewall architecture by abstracting the control and data planes in SDN. The "data plane" monitors network traffic on a network hardware and filters out illegal traffic based on security rules. The "control plane" collects host information and dynamically updates security rules in the "data plane". Besides, an audit server is applied to detect attacks against the control plane.

•*New Mechanism.* We introduce new mechanisms to protect host machine security and provide high programmable application-level security control. Different from existing firewall solutions, which adopt fixed classification algorithms and features, our designs allow network administrators to set up security rules based on user-defined algorithms or features. Furthermore, our design could detect malware traffic even when the malware utilizes a private TCP/IP to bypass traditional firewalls.

•*Implementation and Evaluation.* Based on the mentioned architecture and mechanisms, we design and implement SDF, and evaluate its performance in real-world experiments. Experimental results show that SDF can monitor all network traffic and precisely identify malicious traffic. The audit server can alert users when the control plane is poisoned or shut down. Furthermore, two use cases of SDF are presented to show that the network programmability simplifies today's security solutions.

2 BACKGROUND AND PROBLEM STATEMENT

2.1 Adversary Model

Regular network applications (e.g. Chrome and MSN) use the TCP/IP stack and interfaces provided by the operating system (OS) for network communication. Specifically, the traffic of these applications will pass through the TCP/IP protocol driver, network driver interface specification (NDIS) intermediate driver, NDIS filter driver, and NDIS miniport driver before reaching network interface card (NIC) hardware, as depicted in Fig. 2. The personal firewall lies in one of the four layers to analyze both incoming and outgoing traffic. When malicious traffic is detected, the firewall reports it to the user for decisions or drops it based on the security policies.

Some malware can use a private TCP/IP stack to bypass personal firewalls [20]. Specifically, the malware hooks NdisMRegisterMiniportDriver() and NdisMRegisterMiniport() functions, and registers malware's own miniport handler function before the network adapter driver registers to NDIS. With malware's own miniport handler function, the malware is able to send/receive packets through its private TCP/IP stack and bypass the monitoring of personal firewalls, as depicted in Fig. 2.

Malware also has the ability to poison personal firewalls, such as intercepting the communication between firewall and OS or even shutting down the firewall. When malware tries to damage a defense system, we should ensure that these malicious operations are noticeable to users. Users can take a further step to scan the host to remove the malware.

2.2 SDN Background

Software-defined networking (SDN) is a new network paradigm that separates the control and data planes in a network [16]. The control plane of SDN dictates the whole network behavior. This logical centralization introduces a simpler but more flexible way to manage and control network traffic by a "southbound" protocol (i.e. OpenFlow [16]).

The OpenFlow networks adopt flow rules to handle network traffic. When a packet comes, the OpenFlow switch searches its flow table to see whether this packet matches any flow rules. If a match is found, the OpenFlow switch will follow the action field of this flow rule to process the packet. The actions could be (not limited to): (i) *forward* the packet; (ii) *drop* the packet; (iii) *report* the packet to the control plane. If the packet does not match any flow entries (table-miss), the OpenFlow switch normally sends a `packet_in` message to the control plane for instruction. The control plane then decides how to process the new packet based on the logic of the apps and responds with action and flow rule(s). This reactive flow installation approach enables an easier and more flexible way to manage and control network traffic, and has been widely used in most OpenFlow applications.

2.3 Problem and Challenge

The problem studied in this paper is how to detect malicious traffic of malware on a host machine. To solve this problem, we face the following challenges.

How to avoid malicious traffic bypassing a personal firewall? As we mentioned above, malware can use a private TCP/IP stack to bypass the detection of personal firewalls. Therefore, a good solution should conduct the detection in network hardware layer (lower than the layer that malware works on to avoid being bypassed). Unfortunately, no existing personal firewall monitors traffic in NIC layer. Meanwhile, when malware attacks the firewalls, how to ensure the system remains functional or alerts users when attacks occur is also a challenging problem. Therefore, we need a new security framework for malicious traffic detection.

How to precisely identify malicious traffic? Even though malware cannot bypass network firewalls, a lack of host information on network firewalls may lead to incorrect traffic classification. Personal firewalls can adopt TCP port to associate each packet with some host information (e.g. task name), and report to the user to update blacklist/whitelist dynamically for more precise identification. However, the host information, which is not contained in a packet, cannot be used in network firewalls to identify attack traffic. For instance, the security database of a network firewall has "server name = 'evil.com'" in its blacklist to drop all traffic to "evil.com". When the malware server updates its hostname (e.g. from "evil.com" to "newevil.com"), the network firewall may fail to identify these malicious packets without the information provided by the host machine.

How to provide programmability of security services? Though firewalls may automatically update security policies based on some specific features and algorithms, malware can still bypass them when these features and algorithms (i.e. classifiers) are revealed. The management of security policies still remains in the realm of network administrators. Furthermore, we cannot directly apply SDN into host machine security for programmability since no application-level controls are enabled in OpenFlow specifications. Besides, the controller cannot control network devices that do not support SDN functions. The programmability of the network will be lost with existing commodity switches. Not many companies can afford the expensive replacement of traditional network equipments. Therefore, a good solution should follow the mechanisms in SDN to enable a fine-grained flow management on some controllable network devices (e.g. NIC). Companies can then replace their network devices on some crucial servers to protect them.

3 SYSTEM DESIGN

3.1 System Model and Architecture

The design of SDF is based on the concept of SDN by utilizing the "southbound" APIs (OpenFlow) to provide programmable and flexible security policy control. SDF has a network hardware as its data plane for traffic monitoring. It processes each packet based on the flow rules in its flow table to avoid malicious traffic bypassing the detection and support programmable security control with OpenFlow interfaces (similar to an OpenFlow switch). The implementation could be either on host side (using NetFPGA or programmable NIC [27] to replace traditional NIC), or on switch side (using an OpenFlow switch to replace the traditional switch[1]). The control plane of SDF is built in a host machine to provide programmable and flexible security policy control. This control plane is not centralized, which is different from that of SDN. Besides, it also collects host information to enable fine-grained, application-level traffic control. Based on traffic statistics from the data plane and host information from the control plane, control apps (similar to the controller applications in SDN) could precisely identify malicious traffic.

[1]The control applications in switch replacement scenarios should be carefully designed, since multiple controllers are involved, and each controller should only control the traffic of its host.

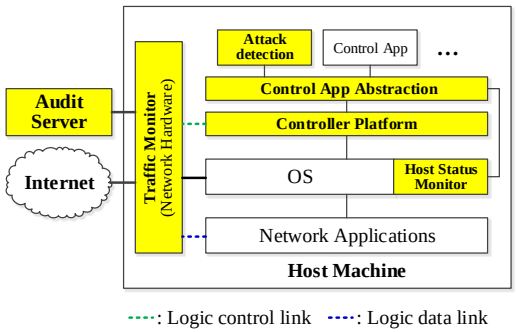


Figure 3. The architecture of SDF. Network applications mean network softwares such as chrome and twitter. Here we use an SDN-like expression to regard network applications as hosts in SDN and do not mean the network applications are "lower" than OS layer.

The architecture of SDF consists of six functional modules: traffic monitor, host status monitor, controller platform, control app abstraction, attack detection and audit server, as depicted in Fig. 3:

Traffic monitor module works as the data plane and runs on a network hardware. It processes and monitors both incoming and outgoing traffic based on the flow rules in its flow table.

Host status monitor module is a monitor application on the host machine that monitors host information. It provides host information to the control app abstraction module to enable application-level management and a precise attack detection.

Controller platform operates much like existing SDN software controllers (e.g. NOX, POX, and RYU). Since it could be implemented commonly by installing software controllers, we skip its design description in this paper.

Control app abstraction module is a middle layer between the controller platform and the control applications. It collects host and traffic information and associates each packet with host information. Besides, the control app abstraction module abstracts the controller implementation language to a high-level language and provides user-friendly interfaces to dynamically update the network security policies.

Attack detection module is a pre-installed control app which identifies malicious traffic based on the host and traffic information. We also allow users develop their own attack detection module based on their own demand.

Audit server is an additional device in the Intranet to detect whether the control plane is poisoned by malware (e.g. the controller is shut down or the flow rules are intercepted and replaced by malware). Audit server works in the Intranet to verify the flow rules on the traffic monitor. It periodically collects the host and traffic information and uses the same database and attack detection algorithm (same classifier) to verify the legality of flow rules.

The workflow of SDF is as follows. Normally, the traffic monitor module checks and forwards incoming/outgoing traffic between the Internet/Intranet and host machine based on the flow rule entries (security rules) in its flow table. When abnormal traffic is detected by the traffic monitor, SDF will follow three steps to handle it. First, the traffic monitor reports the abnormal traffic flows to the controller platform. Second, the reported abnormal flows will be sent to the control app abstraction module along with host information from the host status monitor. Flows will be tagged with host information and sent to the control app (attack detection) to precisely identify malicious traffic. Finally, the attack detection or other control apps decide actions to the reported flows and update security rules on the network monitoring module.

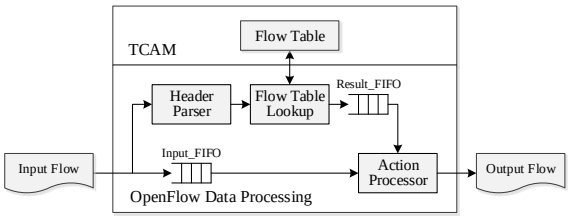


Figure 4. Packet processing pipeline in the traffic monitor.

While other modules are activated, the audit server periodically collects flow entries, host and traffic information to verify the legality of flow entries. The audit server will also alert the network administrator once unexpected/missing flow rules are detected.

3.2 Traffic Monitor

Traffic monitor is a forwarding fabric that processes each packet based on its flow rules. It is specific network hardware which monitors network traffic at network NIC layer to avoid malware to bypass SDF (e.g. via a private TCP/IP stack). The functionality of the traffic monitor stems from the maintenance of flow rules in the flow table (similar to the flow rules in an OpenFlow switch), which are used to enforce security policies. Traffic monitor also provides southbound APIs (i.e. OpenFlow interfaces) to support programmable security control. As we mentioned before, the implementation could be either on the host or switch side. Here we describe a more common scenario that the traffic monitor is a specific hardware.

Similar to an OpenFlow switch, traffic monitor decides the actions (e.g. forwarding, dropping, or reporting) of each incoming/outgoing flow based on flow rules in its flow table stored in Ternary Content Addressable Memory (TCAM). It adopts two ports (two virtual ports when implemented on the host side) to connect the host machine and Internet/Intranet. This scheme allows us to distinguish between incoming traffic and outgoing traffic by network port with great ease. The traffic monitor contains four major components: flow table, header parser, flow table lookup, and action processor, as depicted in Fig. 4. Though traffic monitor module could be implemented by exactly following OpenFlow v1.3 [21] or higher versions, we describe minimum requirements in designing since the resources can be limited in some scenarios.

Flow Table. Flow table component stores flow entries in TCAM. Besides match and action fields, a flow entry also has priority, counter, and timeout fields, as depicted in Fig. 5-a.

```
struct flow_entry {
  match;    /* match against packets based on header */
  priority; /* matching precedence */
  counter;  /* received packets and bytes */
  action;   /* actions applied to matched flows */
  timeout;  /* maximum amount of time before expired */
}
```

(a) The structure of a flow entry.

```
struct header {
  IN_PORT;    /* input port (from host or internet) */
  ETH_SRC;    /* ethernet source address */
  ETH_DST;    /* ethernet destination address */
  ETH_TYPE;   /* ethernet frame type */
  VLAN_ID;    /* VLAN ID */
  VLAN_PCP;   /* VLAN priority */
  IP_SRC;     /* IP source address */
  IP_DST;     /* IP destination address */
  IP_PROTO;   /* IP protocol */
  IP_TOS;     /* IP type of service */
  PORT_SRC;   /* transport source port */
  PORT_DST;   /* transport destination port */
}
```

(b) The structure of a header.

Figure 5. Structures in traffic monitor.

In SDF, we only need to support three actions in the action field: *forward* to host/Internet, *drop* the packet, and *report* to controller. In the counter field, we only need to count matched packets and bytes of this flow entry.

To ensure malicious activities are noticeable when malware attacks the control plane, the flow table component also provides read-only APIs for the audit server to get the current flow entries. Therefore, the audit server can find out whether the control plane is compromised (the verification will be discussed later).

Header Parser. Header parser component extracts the header information of each packet to identify each flow. Fig. 5-b shows different fields in a header.

Most fields in SDF have the same meaning with those in OpenFlow protocol. The IN_PORT field is slightly different. Since the traffic monitor only has two data ports in SDF (interfaces to Internet and host), IN_PORT in SDF only denotes whether a packet is an ingress packet (from the Internet to host) or an egress packet (from the host to Internet).

Flow Table Lookup. After extracting header information, the flow table lookup component conducts both exact and wildcard lookups to match flow entries in the flow table. To ensure efficiency and reduce collisions, we apply two Hash functions on the flow header in the exact lookup. Paralleled with the exact lookup, the wildcard lookup uses a mask to check for any matches in the flow table. If any flow entries are matched with the packet, the flow table lookup will deliver the results (all matched flow entries) to the action processor. Otherwise, the lookup result will be null.

Action Processor. Action processor decides which action should be applied to the packet. Specifically, the action(s) of a flow entry with the highest priority is applied to the packet. The default action (report to the controller) is applied in the null result scenario. Once an action is applied, the counter field of the applied flow entry is updated.

Though OpenFlow v1.3 [21] indicates that OpenFlow switches can preserve the original packet and only encapsulate the header information into `packet_in` messages, the traffic monitor delivers the whole packet to the controller by adopting "encapsulate the whole packet to the controller" in the action field of flow rules (in switch replacement scenarios the action can be "mirror to controller"). It is because the memory in NIC is always limited. Besides, encapsulating the whole packet also allows the control app abstraction module to match the application layer payload with attack signature database (e.g. malware server URL and private information).

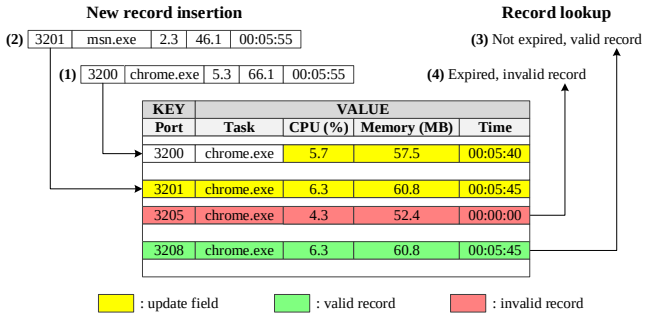


Figure 6. Port-host_info table with two operations: insertion and lookup.

3.3 Host Status Monitor

Host status monitor works on the host machine. It provides APIs to get the task name, CPU and memory utilizations of the task based on a specified port (`GetHostInfoByPort`). The task name information (*task*) serves the purpose of identifying the application that generates the packet. CPU and memory information (CPU and *memory*) indicates the current status of the task. With these features, the accuracy of attack detection can be improved. The control apps can also provide a fine-grained application-level flow management.

We use a *port-host_info* table to associate host information with each port. The host status monitor first queries all process id (*pid*) records on all enabled ports ($port - pid$ record). Then, it queries all *task* records based on the obtained *pid*s, CPU and *memory* utilizations of these tasks. Finally, it associates each *port* record with *task* (even though multiple processes can listen on the same port, these processes belong to the same task), CPU and *memory*. CPU and *memory* will be the same for different *pid* records with the same *task*. To ensure the efficiency of indexing, a *port* field is used as a key for a hash table (*port-host_info* table). *time* field is added when inserting a new record. Each record will expire after t_{expire} time (initially set to 120 seconds). The *port-host_info* table will be updated in every 5-second.

The *port-host_info* table supports two functions, new record insertion and record lookup, as depicted in Fig. 6. When the host status monitor finds that the *port-host_info* table already has an existing record during insertion, the host status monitor compares the *task* field between the existing and new records. If the *task* is the same, the host status monitor only updates the CPU, *memory*, and *time* fields, as shown in (1)-operation in Fig. 6. Otherwise, the host status monitor overwrites the whole record, as depicted in (2)-operation. When record lookup is called, the host status monitor checks the *time* field of the matched record. If the record is not expired, the host status monitor returns the matched record, as shown in (3)-operation. Otherwise, the host status monitor returns EXPIRED, as depicted in (4)-operation. Since new records can overwrite existing records, and *time* field is applied to identify expired records, the *port-host_info* table does not need to support deletion function in regular hash tables. The size of the *port-host_info* table

```
Match     := TASK | CPU | MEMORY | HEADER | PAYLOAD | HEADERS | *
Event     := (FORWARD | DROP | LOG | REPORT)
Rule      := OFMatch | Action | Trigger
OFMatch   := HEADER | *
Action    := (FORWARD | DROP | REPORT)
Trigger   := Begin | End
Begin     := (IMMEDIATE | Time)
End       := (NO_EXPIRE | Time)
Time      := HH : MM : SS
```

Figure 7. A high-level abstract language in the control app abstraction. The values in the brackets enumerated values of this field. "HEADER" in Match field means the whole header of a packet (from MAC layer to transport layer if applicable), while "HEADERS" represents the specific headers (e.g. ethernet type and source IP).

is set to 1000 entries initially. It also supports appending and compacting strategies to adjust its size dynamically.

Host status monitor also calls the OS to get *task*, *CPU*, and *memory* based on the *port* in real-time when no matched record is found or the record is expired in the *port-host_info* table. This operation is to avoid the 5-second delay in table updating. It may seem that real-time calls would satisfy the requirement of `GetHostInfoByPort`. However, the lookups in the *port-host_info* table are much more efficient than real-time calls. In some scenarios, when the connection is closed before calling `GetHostInfoByPort`, the host status monitor cannot get any information without previous records.

3.4 Control App Abstraction

Control app abstraction provides programmable interfaces to users to dynamically update the network security rules. Based on the high-level language described in [7], we design a programmable language with new match fields (i.e. payload and host information) for SDF. Designed upon the existing SDN controller platform, the control app abstraction tags additional fields to flow rules to provide fine-grained and application-level traffic control, and encapsulates the controller implementation language to provide user-friendly APIs.

To enable application-level traffic management, the control app abstraction appends *task*, *CPU*, and *memory* fields to each incoming application-level packet based on port. With PORT_SRC or PORT_DST, the control app abstraction uses `GetHostInfoByPort` to tag incoming packets, which allows control apps to process them based on *task*, *CPU*, *memory*, and other match fields (e.g. IP_SRC). After deciding the actions of these packets, the control apps can further install flow entries to the traffic monitor (these flow entries should follow the flow entry structure described in Section III-B).

The control app abstraction also utilizes a high-level abstract language (via XML) to encapsulate "northbound" APIs of the controller to provide convenient facilities for control app development. This language simplifies controller APIs in SDF scenario and enables a more convenient way to develop control apps even without knowing much about OpenFlow and controller APIs (we also allow users to embed Python script in XML). Three basic elements are included in this language: *Match*, *Event*, and *Rule*, as depicted in Fig. 7. *Match* defines a specific group of flows which the policy targets. If "*" is specified, the policy will be applied on all received packets.

```
<!--Example 1-->
<Policy PolicyID=Training_Classifier_Based_On_DB>
   <Match PAYLOAD=in_DB HEADER=in_DB>
   <Event>SVM_UPDATE</Event>
   <Rule></Rule>
</Policy>

<!--Example 2-->
<Policy PolicyID=Reporting_Suspicious_Flows_To_User>
   <Match CPU_More=20 SVM_CLASS=TRUE>
   <Event>REPORT</Event>
   <Rule></Rule>
</Policy>

<!--Example 3-->
<Policy PolicyID=Blocking_Hidden_Task_Flows>
   <Match TASK=null IN_PORT=host>
   <Event>DROP,LOG</Event>
   <Rule RuleID=Egress_Block>
      <OFMatch IN_PORT=host IP_DST=ip_dst>
      <Action>DROP</Action>
      <Trigger Begin=IMMEDIATE End=NO_EXPIRE>
   </Rule>
   <Rule RuleID=Ingress_Block>
      <OFMatch IN_PORT=internet IP_SRC=ip_dst>
      <Action>DROP</Action>
      <Trigger Begin=IMMEDIATE End=NO_EXPIRE>
   </Rule>
</Policy>
```

Figure 8. Three examples of control apps.

Event describes the action(s) to the matched flows, such as logging the packet (LOG) and reporting to the user/apps (REPORT). Lastly, *Rule* specifies the update of security rules. It will trigger the control app abstraction and controller platform to generate a new flow rule and install it in the traffic monitor. Thereby, users can utilize sophisticated techniques (e.g. machine learning) to create intricate and dynamic security policy control apps.

Fig. 8 illustrates three examples of the control apps. Example 1 implies a use case to update the parameters in attack detection. The user first updates the attack signature database manually. Then, SDF trains the SVM classifier (traffic-based classification in attack detection component) based on the results of database-based classification. Another very useful example is to report suspicious traffic to the user/apps, as depicted in Fig. 8 (Example 2). Based on this policy, SDF reports each packet to the user/apps when traffic-based classification identifies it as illegal and its task consumes more than 20% CPU. The user/apps can then decide the action of these packets[2]. A more complex scenario is the application-level table-miss management. In this scenario, the action of each table-miss packet is decided by the *task*. For instance, the user may want to block the traffic of hidden tasks (most malware conceals its *task* from the OS), and generate rules to block the traffic from/to malware servers, as shown in Fig. 8 (Example 3). Note that the policies described here work on the host machine and could provide application-level traffic management, which is different from the flow rules in the traffic monitor. Generally speaking, policies could generate new flow rules based on the packets delivered to the control plane, while flow rules ensure the efficiency of SDF and reduce the overhead.

The control app abstraction is designed to facilitate control apps to manage table-miss packets. Therefore, a packet

[2]Though SDF can atomically drop these suspicious packets, we do not encourage this action since some benign packets are dropped as well due to the false positives in traffic-based classification.

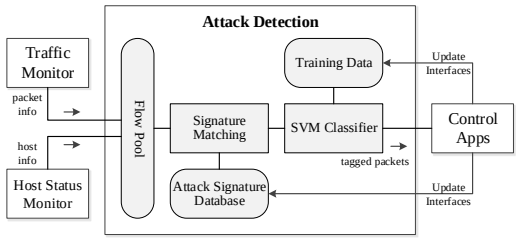


Figure 9. Two-phase matching in attack detection module.

will be first processed based on flow rules, and then handled based on control policies. The traffic monitor only delivers table-miss packets and "report"-action packets to the controller, and the control apps apply the control policies only on these reported packets. In this way, we reduce the response time of non-table-miss packets (matched packets), since adding *task*, CPU, and *memory* fields to each packet can be time-consuming. Furthermore, this mechanism also allows application-level management for table-miss packets, as we discussed in example 3.

3.5 Attack Detection

Attack detection serves the role of identifying malicious traffic and marking each reported packet to assist the easy management of control apps. We adopt a two-phase matching technique to identify malicious traffic based on attack signatures, as depicted in Fig. 9. When a packet arrives, the flow pool will associate this packet with host information, classify this packet to different flows based on header fields described in header parser (in section 3.2), and store the packet in a queue of this flow. In the first phase, attack detection module matches some fields of each packet with the attack signatures in the database and associates with in_DB (e.g. PAYLOAD=in_DB); in the second phase, attack detection module employs Support Vector Machine (SVM) to identify malicious flows and tags SVM_CLASS (e.g. SVM_CLASS=TRUE. TRUE represents the flow is classified as malicious traffic.). The control apps can further decide the action of each packet.

In the first phase, the attack detection module adopts a packet-level classification by checking whether some fields of a packet (the payload is the most significant field since most signatures in attack signature database are in the payload of application layer) are matched with signatures in the attack database. If a packet contains attack signatures, it will be classified as malicious traffic, and associates with in_DB (e.g. PAYLOAD=in_DB). Otherwise, the packet will not be associates with in_DB (e.g. PAYLOAD≠in_DB). We also provide interfaces to update the attack database with great ease. For instance, a user can download new attack signatures from the Internet and update the database manually, or write a control app to update the database based on the attack patterns identified by the traffic-base classification.

In the second phase, the attack detection module adopts a flow-level classification with SVM to precisely identify malicious traffic based on training data. SVM can maximize the distance between training samples and hyperplane. This classification algorithm is robust even with noisy training data. Besides CPU, *memory*, *task*, and header, we also use $frequency$ as a feature of each flow by counting "packets per flow" and "bytes per flow". For the training set, we use the traffic of two attacks (i.e. SYN flood and leaking private information) and normal traffic as training data to build the hyperplane $f(\boldsymbol{x})$ with Gaussian kernel. The SVM classifier can further efficiently classify each flow $\boldsymbol{x}_s$ by judging the sign of $f(\boldsymbol{x}_s)$. All packets in "illegal"-classified flows will be tagged with $SVM\_CLASS = TRUE$, while packets in "legal"-classified flows will be tagged with $SVM\_CLASS = FALSE$. To dynamically adjust the SVM classifier, we also provide interfaces to update the training data. When new training samples are added, the attack detection module can use the new training data to train the classifier.

The overhead may be a concern when much traffic are processed by the attack detection module. Since we can install flow rules in traffic monitoring to drop malicious traffic and forward benign traffic, only some "suspicious" traffic are processed by the attack detection. Furthermore, based on our experiments, the SVM classifier is time-consuming when training, but efficient when classifying. Therefore, we think the overhead of attack detection is acceptable.

3.6 Audit Server

The audit server is an additional device in the Intranet to verify the legality of the flow entries on the traffic monitor. It can be centralized, which is able to support several hosts with only one audit server. When some flow entries are identified as illegal or some crucial flow entries are missing (the control plane is poisoned or shut down by malware), the audit server alerts the network administrators for further analysis on the host machine.

Audit server collects traffic information and host information (*task*, CPU, *memory*, and attack signature database) periodically, and applies the same classification algorithms (classifiers) and security policies of the control apps to generate flow entries for verification. Similar to the procedures on the host, the audit server first tags the *task*, CPU, and *memory* to each packet, and then generate the flow entries based on the security policies. These generated flow entries will be used to identify the unexpected and missing flow entries from the traffic monitor. The inconsistencies will be reported to the network administrator to notify the network administrator when the control plane is attacked (e.g. poisoned or shut down) by malware.

To understand how serious the misclassified/missing flow entries are to reduce false alerts, we introduce "risk level" for the inconsistencies. Risk level is represented by the normalized distance from the misclassified/missing sample (a tagged packet) to the hyperplane in SVM. For instance, suppose the hyperplane is $f(\boldsymbol{x}) = \boldsymbol{\omega}^T\boldsymbol{x}+b$, where $\boldsymbol{\omega}$ is the normal vector of the hyperplane, and could be represented by m training samples $(\boldsymbol{x}_i, y_i)$ and Lagrange multipliers α_i: $\boldsymbol{\omega} = \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i$. The distance between the hyperplane and one misclassified sample $\boldsymbol{x}_s$ is $D_s = |\boldsymbol{\omega}^T\boldsymbol{x}_s + b|/||\boldsymbol{\omega}|| = |\sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i^T \boldsymbol{x}_s + b|/||\boldsymbol{\omega}||$. When the kernel function κ is employed, D_s can be calculated by $D_s = |\sum_{i=1}^{m} \alpha_i y_i \kappa(\boldsymbol{x}_i, \boldsymbol{x}_s) + b|/||\boldsymbol{\omega}||$. The risk level is represented by the normalized distance (divided

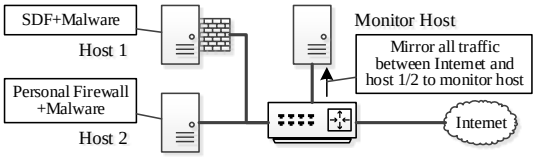


Figure 10. Topology in malware traffic detection experiment.

by the average distance of samples): $R_s = mD_s / \sum_{i=1}^{m} D_i$. The risk levels of other inconsistent flow entries (e.g. flow entries triggered by the exact matches in the attack signature database) will be set to 100 by default.

It may seem that the audit server can take over the role of the controller (similar to the centralized control plane in SDN) to avoid control plane attacks. However, this design will inevitably consume much host-controller (or traffic monitor-controller) bandwidth because of the communication between the controller and host status monitor, especially when SDF provides application-level traffic management. Therefore, we think designing the control plane on the host will reduce the communication overhead and delay. Though in this design, the control plane may be a target of malware, the audit server can alert the network administrator for the abnormality.

4 EXPERIMENT

4.1 Implementation

The prototype of traffic monitor in SDF is built on a specific OpenFlow-enabled network hardware, Broadcom BCM56960 Series [19], which supports the described OpenFlow functions in Section III. We adopt RYU controller [22] as the controller platform and install RYU controller on a PC equipped with i7 CPU and 8GB memory. The host status monitor and control app abstraction are written in Python. We use LIBSVM [3] as the SVM classifier to design the attack detection module. The audit server is built on another Linux host in Python. It adopts the same classifier and control apps on the host.

4.2 Setup

Bypassing Personal Firewalls. One of the most significant improvements in SDF is avoiding malware bypassing personal firewalls. To evaluate malware traffic monitoring performance of SDF, we install SDF and *Rovnix* bootkit on a tested host (host 1) machine and apply the policy "Match: *; Event: LOG, FORWARD; Rule: null" to log all captured traffic. Besides, two different personal firewalls (McAfee and Norton) and *Rovnix* bootkit are installed on another host (host 2) as a control subject. A switch is used to connect to the Internet, traffic monitor module, and the two hosts, and mirrors all traffic between the Internet and tested host to the monitor host to record all traffic from/to the tested host, as depicted in Fig. 10. In this way, the performance can be evaluated by comparing logged traffic with mirrored traffic.

Classification with Host Information. We further study how host information (*task*, *CPU* and *memory* features) affects the accuracy of malicious traffic identification. Two programs are built to generate malicious traffic, SYN-flooder (generating SYN packets with forged source IP) and privacy-leaker (regularly sending host information to a server, including MAC address, IP address, hostname, and running tasks). We apply two policies "Match: (PAYLOAD=in_DB or HEADER=in_DB); Event: LOG, DROP; Rule: null", and log other suspicious traffic classified as illegal by SVM classifier. Furthermore, we disable the host status monitor by returning $task = null$, $CPU = 0$, $memory = 0$ and for all requests to show the classification without host information (the *task*, *CPU* and *memory* features in the training data are also set to *null* or 0). We collect our training data under different kinds of attacks in three scenarios: website browsing, data downloading, and data uploading, and generate new traffic (not from training data) to evaluate the performance of both host-info-enabled and host-info-disabled classifications in different cases.

Table 1. Packet Capturing Rate

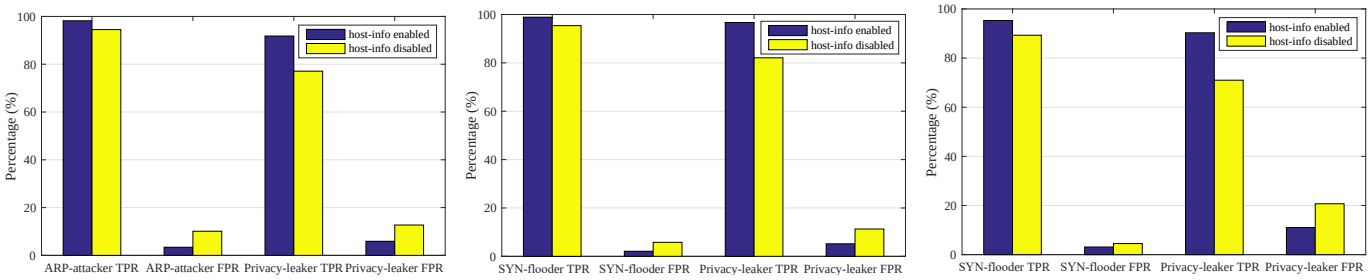
	SDF	McAfee	Norton
Total traffic	100%	84%	87%
Rovnix traffic	100%	0%	0%

Attacks Against Control Plane. We test whether the audit server can alert the network administrator when malware attacks against the control plane. Specifically, we use (i) a malware (mal1.exe) to hook the "modify state" message (for installing/removing flow entries to the traffic monitor) and add a flow entry to forward its traffic (intercepting flow rule installation attacks); (ii) another malware (mal2.exe) to hook the "FlowStatsReply" function triggered by ofp_event.EventOFPFlowStatsReply event and replace the packet count and byte count of its flows by 3 and 198 respectively to lead incorrect classification (poisoning traffic statistic attacks); and (iii) the third malware (mal3.exe) to shut down the control plane when it finds that the destination is unreachable, and install a flow rule to forward its traffic (shutting down control plane attacks). All malware will flood SYN packets to a server (20 packets per second). The security policies are set to block an illegal packet identified by the SVM classifier, and only the *task* feature of mal1.exe is in the attack signature database.

Packet Processing Overhead. The packet processing overhead is mainly incurred by the procedures of table-miss flows, including flow table lookup, `packet_in` request, appending features, classification, and flow rule installation. Thus, it is imperative to include all mentioned procedures in the evaluation of overhead (e.g. Rule=null is not acceptable). We apply the policy "Match: (SVM_CLASS=FALSE, IN_PORT=host); Event: FORWARD; Rule: (OFMatch:IN_=host & IP_DST=ip_dst, Action=FORWARD)" to allow the connections when identified as benign traffic by the SVM classifier. We evaluate the packet processing overhead by measuring the round trip time of 100 packets generated by two tested hosts (with and without SDF).

4.3 Experimental Result

Malware Traffic Capturing. In this experiment, we use (*Packets Captured by Host*)/(*Packets Captured by Monitor Host*) to calculate the packet capture rate. The result is depicted in Table 1. In the control subject, even though McAfee and Norton alert that malware is detected when *Rovnix* is copied to the host (the file of *Rovnix* matches with



(a) Browsing websites. (b) Downloading data. (c) Uploading data.

Figure 11. Comparisons between host-info enabled and host-info disabled classifications in SDF.

the attack signature database), neither of them can capture the traffic of *Rovnix* bootkit. On the other hand, SDF is able to capture all packets of *Rovnix* bootkit. Since we can hardly control the number of generated packets, we regard the performances of McAfee and Norton are the same.

Malicious Traffic Identification. First, we analyze how different features affect the classification result. In our evaluation, we find all features presented in Section 3.5 contribute to the SVM classifier. While the weights of different features vary for different kinds of attacks. Specifically, *task*, *frequency*, *CPU*, and protocol field in header are significant for ARP-attacker and SYN-flooder, and *task* and protocol field are significant for privacy-leaker (note that the traffic of privacy-leaker can also be identified by the packet-level classification, but we only consider the affect on SVM classifier here). Furthermore, we use true-positive rate *TPR*=(*Detected malicious packets*)/(*Malicious packets*) and false-positive rate *FPR*=(*Benign packets classified as malicious packets*)/(*Benign packets*)to compare the performances between host-info enabled and host-info disabled classifications in each scenario (browsing, downloading, and uploading). The results are shown in Fig. 11. Both of the classifications under SYN-flooder are more precise than those under privacy-leaker. It is because the malicious packets generated by SYN-flooder are SYN packets. Therefore, "TCP protocol" becomes a significant feature. On the other hand, the traffic of privacy-leaker is hard to be detected especially in uploading scenario. It is because some private information (e.g. running tasks) is not included in the attack signature database. Distinguish malicious traffic from regular updating traffic is difficult. The host-info enabled classification performances better than host-info disabled classification in all scenarios. The results show that host features increase the *TPR* by more than 15% and reduces the *FPR* by around 5% in identifying the traffic of privacy-leaker.

Alerts for Control Plane Attacks. The audit server can identify the inconsistencies of flow entries and alert to different kinds of control plane attacks. The notifications of intercepting flow rule installation attacks (mal1.exe) and poisoning traffic statistic attacks (mal2.exe) are presented in Fig. 12, and those of shutting down control plane attacks are presented in Fig. 13. In the intercepting flow rule installation attacks, the risk level of its fraud flow entry (rule #1 in Fig. 13) is set to 100, because the *task* of mal1.exe is preserved in the attack signature database. Since mal2.exe is not in

```
********** Auditing Host 1 (192.168.1.103) **********
Collecting Data from Host 1 (192.168.1.103)...
4 inconsistency(s) detected on Host 1 (192.168.1.103)!
 NO.    RISK    DESCRIPTION     TASK            CPU     MEMORY
 1      100     missing         mal1.exe        11.8    9.4
 2      100     unexpected      -               -       -
 3      8.2     missing         mal2.exe        12.5    8.9
 4      100     unexpected      -               -       -
Input the flow number to show details, '0' for all: 0
 1: n_packets=0, n_bytes=0, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=null
 2: n_packets=923, n_bytes=60918, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=output:INTERNET
 3: n_packets=0, n_bytes=0, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=null
 4: n_packets=1106, n_bytes=72996, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=output:INTERNET
```

Figure 12. Alerts to intercepting flow rule installation attacks and poisoning traffic statistic attacks.

```
********** Auditing Host 1 (192.168.1.103) **********
Collecting Data from Host 1 (192.168.1.103)...
2 inconsistency(s) detected on Host 1 (192.168.1.103)!
 NO.    RISK    DESCRIPTION     TASK            CPU     MEMORY
 1      6.4     missing         mal3.exe        16.7    6.1
 2      100     unexpected      -               -       -
Input the flow number to show details, '0' for all: 0
 1: n_packets=0, n_bytes=0, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=null
 2: n_packets=791, n_bytes=52206, priority=1, in_port=HOST, ip_dst=192.168.1.200 actions=output:INTERNET
```

Figure 13. Alerts to shutting down control plane attacks.

contained in the attack signature database, the risk level of poisoning traffic statistic attacks (fraud rule #2 in Fig. 13) is 8.2 calculated by the normalized distance to the hyperplane. When the controller suffers from shutting down control plane attacks (mal3.exe), the audit server can also detect the inconsistency of flow entries. The risk level of the fraud rule is 6.4. Besides, when the controller is shut down, we may find some low-risk alerts if the classifier has a high false-positive rate (the flow rules to block some regular traffic will also cause inconsistencies). Notice that the results of these attacks are almost the same (removing some flow entries and installing fraud flow entries), the audit server cannot distinguish the controller suffers from which kind of attacks. Network administrators can conduct a further analysis based on the alerts from the audit server.

Packet Delay. Table 2 describes the packet processing overhead in SDF system. The TCAM ensures very short delays for processing matched packets, incurring less than 5ms average delay. However, the processing time of table-miss packets is much longer. The average delay increases to 105ms because the traffic monitor needs to send the packets to the controller, and the controller needs to decide and send the actions back to the traffic monitor. Fortunately, the table-miss packets are only a small portion of the network traffic in most scenarios, since the controller can update the flow rules on the traffic monitor to process the packet when received again. Furthermore, we build the traffic monitor as a specific hardware in our prototype. The link delay of table-miss packets can be reduced when the traffic monitor works on NIC side.

Table 2. Time Delays

	Max	Min	Avg
Regular packets (w/o SDF)	62ms	14ms	18ms
Matched packets (w/ SDF)	66ms	16ms	21ms
Table-miss packets (w/ SDF)	214ms	78ms	105ms

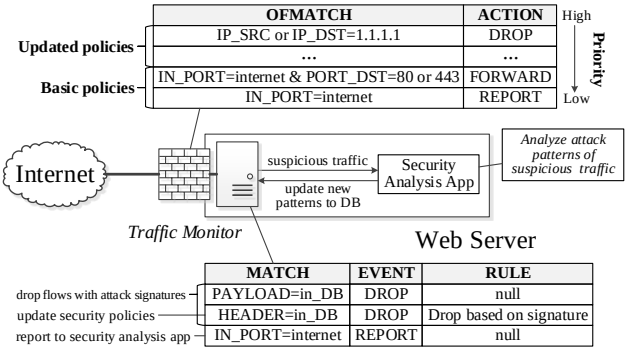


Figure 14. Use case 1: server protection.

4.4 Use Case

Use Case 1: Server Protection. Many servers (e.g. web servers) are accessible for external users and become targets of various network attacks, such as DDoS attacks and XSS attacks. To protect these servers, a more and more awared protection method is only allowing specific ports of these services (e.g. 80 for HTTP and 443 for HTTPS). Other requests to unauthorized ports will be redirected to the security agent for further analysis. Meanwhile, a traffic monitor agent is always applied to detected malicious traffic during the communication (e.g. destination IPs are in the blacklist, traffic follows WebShell models, and scripts are downloaded rather than parsed). Originally, we need two agents (i.e. traffic monitor agent and security agent) to protect the server and need to update the attack database in traffic monitor agent manually based on the feedback from the security agent.

In this scenario, we can adopt SDF to update the security policies automatically without introducing the two agents. To allow requests to port 80 and 443, and analyze other requests, we first apply proactive flow rules (basic security policies) to forward benign incoming packets (DST_PORT=80 or 443) and report other incoming packets to the controller. Second, we design a Security Analysis App which can figure out new attack signatures of these malicious incoming packets, such as malicious servers' IPs and WebShell models. Based on these new signatures, the Security Analysis App dynamically updates the attack signature database. Finally, we build a security policy control app to report suspicious traffic to the Security Analysis App and dynamically update the security policies. A model to protect server security with SDF is depicted in Fig. 14 (the audit server is not presented to simplify the description).

In the test, we try to connect to an unauthorized port (8080), and upload malware through port 80. When SDF is activated, even port 8080 is open, we still cannot establish a connection with the server through 8080. The upload is also failed, and the client's IP address is added to the attack signature database.

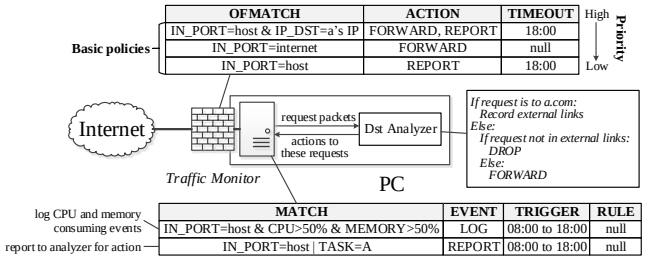


Figure 15. Use case 2: parental network controls.

Use Case 2: Parental Network Controls on PC. Parental controls can manage the network accessibility of different users. Originally, parental controls associate each account with a blacklist/whitelist. The PC simply denies/allows each request based on the blacklist/whitelist, which makes the access control very inflexible. This inflexible strategy may affect some websites with external links. For instance, if the policy only allows connecting to "a.com", other external links (e.g. the img tag "<img src='b.com/1.png'>") in "a.com" will also be blocked. We cannot view any the images which are not in the domain of "a.com". Parental controls will also fail to block online games when the games are not in the blacklist. If the policies change with applications and time, we may need to use two accounts with different policies, and set the active period of each account. Furthermore, since these policies are OS-level filters, they can be bypassed by using a private TCP/IP stack.

With the assist of SDF, we can enable a flexible management of network accessibility with only one account, as depicted in Fig. 15 (the audit server is not presented). We first adopt three proactive flow rules to forward requests to "a.com", forward all responses, and report other requests to the controller. Second, we apply a security rule to check whether the packet is triggered by applications which consume more than 50% CPU or 50% memory. In such cases, it may be generated by some online games. Third, we design a Dst Analyzer to check whether the current request is in the external links of the previous request[3], and decide the action of each request to other domains. Finally, we adopt a security policy control app to deliver each received request to the Dst Analyzer and set the trigger time to 08:00 to 18:00 (free network accessibility during other time). Notice that in the design of Dst Analyzer, we do not add new flow rules into the traffic monitor (e.g. IP_DST=b's IP, FORWARD). This ensures the requests to "b.com" will also be dropped even when "b.com/1.jpg" appears in the external links of "a.com". In this way, we enable a flexible way for parental network controls.

In our test, we allow "google.com", and then search "Facebook" in Google Image. The result shows all images (the URLs of theses images belong to external links). We also try to connect to "www.facebook.com" at 17:55, but blocked. The connection request is allowed at 18:05. We use Warcraft

[3]The Dst Analyzer should also filter internal links to avoid flushing *EL*. We do not present the detail of Dst Analyzer since we only show a simple example.

to test our traffic, and find some Warcraft traffic and software updating traffic can be logged.

5 LIMITATION AND DISCUSSION

Traffic monitor on NIC. We have pointed out that the traffic monitor component can be implemented on either switch side or NIC side, and this paper presents a more common scenario that the traffic monitor is a specific network hardware. Actually, the NIC-side implementation is more convenient for common users. Besides, the switch-side implementation is more complex with multiple hosts. The value of IN_PORT field should be more than two (to identify different hosts), and the control apps should be separated into different groups to isolate the management of each host. We regard the NIC-side implementation as a more promising solution, but the hardware resource limitation can be an obstacle. Therefore, our intention is to implement and optimize SDF on NIC side with limited hardware resources in the future.

Delay in application-level traffic control. Application-level traffic control provides a more flexible way for traffic engineering. Even though SDF is able to provide application-level table-miss control by associating the host information with each packet on the host side, the traffic monitor cannot conduct this association without the $port - host\_info$ table. It seems that regarding all packets as table-miss packets (OFMatch:*, Action=REPORT) can be a simple solution, and the control apps can then receive and identify the host information of each packet. However, this naive solution incurs much overhead into the network (e.g. long delay and significant bandwidth consumption) in switch-side traffic monitor implementation. Our intention is to maintain the $port - host\_info$ table on the traffic monitor side, and create $task$, CPU, $memory$ fields in the header. Though maintaining the table consumes some bandwidth, the overhead is significantly less than the naive solution. Since this solution modifies OpenFlow protocol by introducing additional fields and $port - host\_info$ table, it might be impractical in the switch-side implementation scenarios.

Evasion of SDF. SDF collects host information from the host status monitor to identify illegal packets. However, malware can also hook the APIs of host status monitor to provide fake host information for the attack detection and audit server. In such scenarios, we suggest the network administrator train the classifier to get the normal network behavior of each application with the application's traffic. Considering a client application, it normally connects to a DNS server to get the server's IP and establishes a connection to the server. When the SDF detects TCP packets before DNS queries, it can report these suspicious events to the network administrator for further analysis. Besides, SDF can also use some "checkpoints" to verify the host status. For instance, when several services are activated, the CPU and memory utilization rate should be in a range.

SDN attacks on control and/or data plane. Since SDF is implemented following the mechanisms in SDN, it might suffer some specific attacks. We have shown that the audit server can verify the flow entries on the traffic monitor and alert the network administrator when inconsistencies are found. However, identifying which kind of attacks still lies in the realm of the network administrator. Besides, SDN-aimed attacks such as data-to-control plane saturation attacks [24] and network topology poisoning attacks [8] (network topology poisoning attacks only work in switch-side implementation scenarios) can also be potential threats to SDF. The user can limit some network traffic or deliver the attack traffic to a specific device to mitigate data-to-control plane saturation attacks [5, 24, 28], and use fixed topology or verify the legality of link layer discovery protocol (LLDP) packets to avoid network topology poisoning attacks [8]. Furthermore, existing SDN security systems [14, 18, 29], can also facilitate users against these attacks.

6 RELATED WORK

Malware traffic detection. Since most malware needs network connections to conduct malicious activities, the detection of these malicious traffic attracts much attention of recent studies [1, 4, 11, 17, 26, 30]. Perdisci *et.al.* present a network-level behavioral malware clustering system by analyzing the structural similarities among malicious HTTP traffic traces generated by HTTP-based malware [17]. The detection is effective for HTTP-based malware, but it does no show structural similarities of other protocols. To deal with packets of other protocols, Amann *et.al.* propose a novel network control framework that provides passive network monitoring systems with a flexible and unified interface for active response [1]. Though the interface for monitoring is flexible, it does not involve host information to ensure a precise result. Jackstraws identifies command and control connections from bot traffic [11]. It leverages host-based information and associates each network connection with a behavior graph for the classification. Another approach provides an Internet worm monitoring system based "detecting the trend" with Kalman filter [30]. An accurate signature tree is proposed to detect polymorphic worms in [26].

SDN security. The approaches in SDN security can be classified into two directions: SDN-self security, which analyzes the vulnerabilities and potential threats in SDN [2, 5, 6, 12, 24]; and SDN-enabled security, which utilizes SDN to solve traditional network problems [7, 10, 13, 23, 25]. This paper mainly focuses on the SDN-enabled security. Shin *et.al.* investigate how the new features provided by SDN can enhance network security and information security process [23]. Many examples of security applications are presented, including firewall, intelligent honeypot, and network-level access control. To facilitate accurate detection as well as flexible resolution of firewall policy violations in dynamic OpenFlow networks, FlowGuard framework is proposed to support the stateful firewall for SDNs with various toolkits for supporting visualization, optimization, migration, and integration of SDN [10]. Besides network security, SDN also brings new insights into device security. PBS is a new security solution to enable fine-grained, application-level network security programmability for the purpose of network management and policy

enforcement on mobile devices [7]. By abstracting mobile device elements into SDN network elements, PBS provides network-wide, context-aware, app-specific policy enforcement at run-time.

7 CONCLUSION

Personal firewalls always fail to detect malicious traffic when malware adopts a private TCP/IP stack. Such traffic may also escape the detection from network firewalls. Motivated by the concept of SDN, we propose SDF, a programmable firewall to detect malicious traffic by abstracting traditional firewall into control and data planes. SDF monitors traffic on a network hardware to avoid being bypassed by malware, and collects host information to conduct a more precise classification to identify malicious traffic and provide application-level traffic control. SDF also enables programmable security control, which allows control apps to dynamically update the network security policies. Experimental results show that SDF can monitor all network traffic and improve the accuracy of attack detection. Besides, it also alerts the network administrator about the inconsistencies of flow entries when malware attacks the controller. We believe with the assist of SDF, many existing security solutions could be solved in an easier and more flexible way.

ACKNOWLEDGEMENT

This work was supported in part by NSFC 61772446, HK PolyU G-UACH, NSFC 61502394, and the Fundamental Research Funds for the Central Universities 3102017OQD097.

REFERENCES

[1] Johanna Amann and Robin Sommer. 2015. Providing Dynamic Control to Passive Network Security Monitoring. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*.

[2] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. 2015. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*.

[3] Chang Chih-Chung and Lin Chih-Jen. 2017. LIBSVM. https://www.csie.ntu.edu.tw/~cjlin/libsvm/. (2017).

[4] Juan Deng, Hongda Li, Hongxin Hu, Kuang-Ching Wang, Gail-Joon Ahn, Ziming Zhao, and Wonkyu Han. 2017. On the Safety and Efficiency of Virtual Firewall Elasticity Control. In *Proc. of the Network and Distributed System Security (NDSS)*.

[5] Shang Gao, Zecheng Li, Bin Xiao, and Guiyi Wei. 2018. Security Threats in the Data Plane of Software-Defined Networks. *IEEE Network* (2018).

[6] Shang Gao, Zhe Peng, Bin Xiao, Aiqun Hu, and Kui Ren. 2017. FloodDefender: Protecting Data and Control Plane Resources under SDN-aimed DoS Attacks. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*.

[7] Sungmin Hong, Robert Baykov, Lei Xu, Srinath Nadimpalli, and Guofei Gu. 2016. Towards SDN-Defined Programmable BYOD (Bring Your Own Device) Security. In *Proc. of the Network and Distributed System Security (NDSS)*.

[8] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proc. of the Network and Distributed System Security (NDSS)*.

[9] Hongxin Hu, Gail-Joon Ahn, and Ketan Kulkarni. 2012. Detecting and Resolving Firewall Policy Anomalies. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 9 (2012).

[10] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. 2014. FLOWGUARD: Building Robust Firewalls for Software-Defined Networks. In *Proc. of the ACM Workshop on Hot Topics in Software Defined Networking*.

[11] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. 2011. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *USENIX Security Symposium (USENIX Security)*.

[12] Samuel Jero, William Koch, Richard Skowyra, Hamed Okhravi, Cristina Nita-Rotaru, and David Bigelow. 2017. Identifier Binding Attacks and Defenses in Software-Defined Networks. In *Proc. of the USENIX Security Symposium (Security)*.

[13] Soyoung Kim, Sora Lee, Geumhwan Cho, Muhammad Ejaz Ahmed, Jaehoon Jeong, and Hyoungshick Kim. 2017. Preventing DNS Amplification Attacks Using the History of DNS Queries with SDN. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*.

[14] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip Porras. 2017. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proc. of the Network and Distributed System Security (NDSS)*.

[15] Zhen Ling, Junzhou Luo, Kui Wu, Wei Yu, and Xinwen Fu. 2014. TorWard: Discovery of Malicious Traffic Over Tor. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*.

[16] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. In *ACM SIGCOMM Computer Communication Review*.

[17] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces. In *Proc. of the Symposium on Network System Design and Implementation (NSDI)*.

[18] Phillip A Porras, Steven Cheung, Martin W Fong, Keith Skinner, and Vinod Yegneswaran. 2015. Securing the Software Defined Network Control Layer. In *Proc. of the Network and Distributed System Security (NDSS)*.

[19] Broadcom. 2017. Broadcom BCM56960 Series. https://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series. (2017).

[20] Microsoft. 2013. The evolution of Rovnix: Private TCP/IP stacks. https://blogs.technet.microsoft.com/mmpc/2013/07/25/the-evolution-of-rovnix-private-tcpip-stacks/. (2013).

[21] Open Networking Foundation. 2012. OpenFlow Switch Specification v1.3.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf. (2012).

[22] RYU SDN Framework Community. 2013. RYU Controller. https://osrg.github.io/ryu/. (2013).

[23] Seungwon Shin, Lei Xu, Sungmin Hong, and Guofei Gu. 2016. Enhancing Network Security through Software Defined Networking (SDN). In *Proc. of the International Conference on Computer Communication and Networks (ICCCN)*.

[24] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. 2013. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *Proc. of the ACM Conference on Computer & Communications Security (CCS)*.

[25] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2016. Enabling Practical Software-defined Networking Security Applications with OFX. In *Proc. of the Network and Distributed System Security (NDSS)*.

[26] Yong Tang, Bin Xiao, and Xicheng Lu. 2011. Signature Tree Generation for Polymorphic Worms. *IEEE Transactions on Computers (TC)* 60 (2011).

[27] Ilenia Tinnirello, Giuseppe Bianchi, Pierluigi Gallo, Domenico Garlisi, Francesco Giuliano, and Francesco Gringoli. 2012. Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*.

[28] Haopei Wang, Lei Xu, and Guofei Gu. 2015. FloodGuard: A DoS Attack Prevention Extension in Software-Defined Networks. In *Proc. of the IEEE/IFIP Dependable Systems and Networks (DSN)*.

[29] Xitao Wen, Bo Yang, Yan Chen, Chengchen Hu, Yi Wang, Bin Liu, and Xiaolin Chen. 2016. SDNShield: Reconciliating Configurable Application Permissions for SDN App Markets. In *Proc. of the IEEE/IFIP Dependable Systems and Networks (DSN)*.

[30] Cliff C Zou, Weibo Gong, Don Towsley, and Lixin Gao. 2005. The Monitoring and Early Detection of Internet Worms. In *IEEE/ACM Transactions on Networking (TON)*, Vol. 13.