# ProSec: Enabling programmable security in Software-Defined infrastructure

---

A Dissertation
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Hongda Li
August 2020

---

Accepted by:
Dr. Hongxin Hu, Committee Chair
Dr. Kuang-Ching Wang
Dr. James Martin
Dr. Feng Luo

# Abstract

There is an emerging trend in modern infrastructure, which is also known as Software-Defined Infrastructure (SDI), pushing a shift of computing from static and hardware-based control systems to dynamic and easily-reconfigurable software-defined systems. However, the development of corresponding security mechanisms that are suitable to protect modern infrastructure is lagging behind.

In the network level, traditional network security functions, such as firewalls and Network Intrusion Detection Systems (NIDS), are implemented on vendor proprietary appliances. Those hardware-based appliances are placed at fixed locations and with fixed capacity. As a result, they are limited in elasticity. However, altering traditional network security functions to make them provide well-suited protection to modern infrastructure requires significant efforts. In the host level, traditional security functions are primarily built upon simple binary security directives, usually recognized as deny and allow. The poor expressiveness of security directives used by traditional security functions greatly hinders the innovations on host security functions. In addition, a unified interface that offers an easy way to reconfigure both network and host security functions is missing.

In this dissertation, our research goal is to fill the gap between the dynamic nature of SDI and the ossified traditional security functions through enabling programmable security, which features elastic, programmatically-extensible, and easily-reconfigurable security functions. To achieve the research goal, we propose to develop a framework called ProSec for supporting programmable security in the network level and host level. In the network level, two critical network security functions, the programmable firewall and programmable NIDS, are designed and implemented, leveraging two emerging networking paradigms, Software-Defined Networking (SDN) and Network Function Virtualization (NFV). In the host level, ProSec abstracts the activities in operating systems with a flow-based model, called System Flow, which is anticipated to offer richer expressiveness to security

directives. To demonstrate the usefulness of System Flow, we develop an Internet of Things (IoT) malware infection detection and an IoT malware infection prevention security function based on System Flow. Finally, as a use case study, we develop a cross-level IoT malware defense security function, taking the advantages of both network-level and host-level security functions to defense IoT malware.

# Acknowledgments

I thank my advisor Dr. Hongxin Hu for his continuous support in overcoming obstacles I have been facing through my research (and my life). I thank Dr. Juan Deng for her guidance on me for the firewall research and efforts on paper writing. I thank Dr. Nuyun Zhang for her guidance on me for the NIDS research and her suggestions on system development. I thank Dr. Guofei Gu from Texas AM University for his useful comments and suggestions on my research about NIDS, system flow, and IoT malware defense. I thank Lei Xu, Sungmin Hong, Jianwei Huang, and Hao Jin Texas AM University for their help with the system flow model implementation and paper writing. I thank Dr. Long Cheng for his insightful suggestions that improved the design and implementation of IoT malware defense approach, thank Qiqing Huang for this efforts on deploying and evaluating the system, and thank Fei Ding for his help in data collection and analysis. I thank the graduate students that I have worked with on the firewall, NIDS, and malware research, especially, Zhizhong Pan, Jingyao Ma, Jinrui Wang, Feng Liu, Fuqiang Zhang, Guoze Tang, and Lavaniyaa Seshathiri who devoted lots of efforts in system implementation, DevOps, and system testing for the research in this dissertation. I thank Dr. James Martin, Dr. Kuang-ching Wang, and Dr. Feng Luo for all the recommendations they gave during my proposal presentation. I am also grateful to all my labmates from NSSLab for their help on my research (and my life), especially, Feng Wei who provides me with useful suggestions on the modeling approach for malware detection.

# Table of Contents

**Title Page** . . . . . **i**

**Abstract** . . . . . **ii**

**Acknowledgments** . . . . . **iv**

**List of Tables** . . . . . **viii**

**List of Figures** . . . . . **ix**

**1 Introduction** . . . . . **1**
- 1.1 Problem Motivation . . . . . 1
- 1.2 Research Motivation . . . . . 2
- 1.3 Research Objective . . . . . 3
- 1.4 Overview of Research Tasks . . . . . 3
  - 1.4.1 Task1: Enable safe and efficient virtual firewall elasticity control (RQ1) . . . 3
  - 1.4.2 Task2: Achieve safe and efficient NIDS virtualization (RQ2) . . . . . 4
  - 1.4.3 Task3: Enable expressive security directives for host security functions (RQ3) 5
  - 1.4.4 Task4: Case study: defending malware in IoT with ProSec (RQ4 & RQ5) . 6

**2 Background and Related Work** . . . . . **7**
- 2.1 Researches on Network Security Functions . . . . . 7
  - 2.1.1 Firewall Policy Analysis . . . . . 7
  - 2.1.2 Scalable NIDSes . . . . . 8
  - 2.1.3 Network Security Functions Employing SDN & NFV . . . . . 8
- 2.2 Researches on Host Security Functions . . . . . 9
  - 2.2.1 Provenance Analysis Systems . . . . . 9
  - 2.2.2 Information Flow Control Systems . . . . . 9
  - 2.2.3 System Resource Access Control . . . . . 10
  - 2.2.4 Security Monitoring Systems . . . . . 10
- 2.3 Researches on Malware Detection . . . . . 11
  - 2.3.1 Network-based Solution . . . . . 11
  - 2.3.2 Host-based Solution . . . . . 11

**3 Safe and Efficient Virtual Firewall Elasticity Control** . . . . . **13**
- 3.1 Introduction . . . . . 13
- 3.2 Research Challenges . . . . . 14
- 3.3 Methodology . . . . . 15
  - 3.3.1 High-level View of Virtual Firewalls . . . . . 15
  - 3.3.2 Dependency Analysis And Semantic Consistency . . . . . 16
  - 3.3.3 Flow Update Analysis . . . . . 18

3.3.4 Buffer Cost Analysis . . . 22
3.4 Optimal Scaling . . . 24
3.4.1 Virtual Firewall Modeling . . . 24
3.4.2 Optiaml Scaling Out . . . 24
3.4.3 Optimal Scaling-In . . . 26
3.5 Implementations . . . 27
3.6 Evaluation . . . 28
3.6.1 Experiment Setup . . . 28
3.6.2 Performance of Virtual Firewalls . . . 28
3.6.3 Rule Dependencies in Firewall Policies . . . 29
3.6.4 Elasticity of Virtual Firewalls . . . 29
3.6.5 Impact of Migration . . . 33
3.6.6 Performance of Optimal Scaling Calculation . . . 35
3.7 Conclusion . . . 36

**4 Safe and Efficient NIDS Virtualization . . . 37**
4.1 Introduction . . . 37
4.2 Research Challenges . . . 38
4.3 Methodology . . . 39
4.3.1 High-level View of vNIDS . . . 39
4.3.2 Effective Intrusion Detection . . . 40
4.3.3 Non-monolithic NIDS Provisioning . . . 42
4.3.4 Implementations . . . 49
4.4 Evaluation . . . 49
4.4.1 Data Collection & Attack Trace Generation . . . 51
4.4.2 Detection Effectiveness of vNIDS . . . 52
4.4.3 Detection State Sharing Overhead . . . 53
4.4.4 Microservice Efficiency . . . 54
4.4.5 Flexibility of Placement Location . . . 55
4.4.6 Flexibility of Processing Capacity . . . 56
4.5 Conclusion . . . 56

**5 Expressive Security Directives for Host Security Functions . . . 61**
5.1 Introduction . . . 61
5.2 Methodology . . . 62
5.2.1 System Flow Overview . . . 62
5.2.2 Event Generator . . . 63
5.2.3 Flow Table Manager . . . 65
5.2.4 Action Scheduler . . . 68
5.2.5 Host Security Functions . . . 69
5.3 Evaluation . . . 69
5.3.1 Micro-Benchmark Results . . . 70
5.3.2 Scalability with Flow Rules . . . 70
5.4 Conclusion . . . 72

**6 Case Study: Defending malware in IoT with ProSec . . . 74**
6.1 Introduction . . . 74
6.2 Host-based IoT Malware Infection Detection . . . 75
6.2.1 Understanding IoT Malware Infection . . . 75
6.2.2 Modeling IoT Malware Infection Process . . . 86
6.2.3 Infection Detector Development . . . 92

6.2.4 Evaluation . . . 95
6.2.5 Effectiveness Evaluation of SOTER . . . 96
6.2.6 Generalization Evaluation of Modeling Approach . . . 99
6.2.7 Performance Overhead Evaluation of SOTER . . . 100
6.3 IoT Malware Infection Prevention . . . 102
6.3.1 Design Choices . . . 102
6.3.2 Implementation . . . 103
6.4 Cross-level IoT Malware Defense . . . 105
6.4.1 Problem Statement . . . 106
6.4.2 Infrastructure-wide Architecture . . . 106
6.4.3 Implementation . . . 107
6.4.4 Evaluation . . . 108
6.5 Conclusion . . . 110

**7 Discussion . . . 112**
7.1 Network-level Security Functions . . . 112
7.2 Host-level Security Functions . . . 113
7.3 IoT Malware Defense . . . 113

**8 Conclusion . . . 115**

**Bibliography . . . 117**

# List of Tables

3.1 Rule dependencies in real-world firewall policies. . . . . . . . . . . . . . . . . . . . . . 31

4.1 Detection programs that we have written for our virtualized NIDS. . . . . . . . . . . . 50

5.1 Supported system events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
5.2 Example system object attributes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.3 Security actions that we define. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.4 LMbench results for System Flow. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.5 UnixBench resulted sources for System Flow. . . . . . . . . . . . . . . . . . . . . . . 71

6.1 Summary of the phases of infection process involved in 3 open-source IoT malware and 18 reverse-engineering reports. . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.2 Top 20 commands invoked by malicious ELF files, infection scripts, and overall files in our dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
6.3 A list of infection capabilities we abstracted and corresponding explanations. . . . . 81
6.4 Infection capabilities exploited in each phase, exploitation percentage, and top commands (up to 5). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.5 A breakdown of space consumed by each asset of the infection detector. . . . . . . . 93
6.6 Software IoT device deployment statistics. . . . . . . . . . . . . . . . . . . . . . . . . 97
6.7 Real-time detection results of Soter. FN: False Negative; FNR: False Negative Rate; TPR: True Positive Rate. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.8 Details of each sample set used in our evaluation. . . . . . . . . . . . . . . . . . . . . 99
6.9 Software IoT device deployment statistics. . . . . . . . . . . . . . . . . . . . . . . . . 109
6.10 Real-time detection results of cross-level IoT malware defender. FN: False Negative; FNR: False Negative Rate; TPR: True Positive Rate; HP: Host-level Prevention; HPR: Host-level Prevention Rate; PS: Port Scan; PSR: Port Scan Rate; CP: Cross-level Prevention Rate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

# List of Figures


1.1 ProSec consists of three major piece of work, enabling network-level programmable security, host-level programmable security, and cross-level security interaction, respectively. The black blocks indicate the components that will be contributed by this dissertation. . . . . 4

3.1 `VFW Controller` components and workflow. . . . . 15
3.2 Example of firewall rule associations and classes. . . . . 18
3.3 *SW* in three different cases with respect to a classic 3-tire cloud architecture. . . . 20
3.4 Packet paths during the migration. . . . . 22
3.5 Relationship between virtual firewall performance and rule size. . . . . 30
3.6 `VFW Controller` for VFW elasticity. . . . . 32
3.7 Performance overhead of migration. . . . . 34
3.8 Performance of provision plan calculation. . . . . 35

4.1 High-level view of vNIDS architecture. . . . . 39
4.2 Different DLs can be implemented by different microservice chains. . . . . 44
4.3 Major steps of the DLP partitioning. (1) Generating PDG of the DLP; (2) Computing payload-based DLP slice; (3) Constructing header-based DLP; (4) Constructing payload-based DLP; (5) Emit the code. . . . . 46
4.4 Detection rate of known malicious activities for Bro, vNIDS, Sharing all states, and No sharing solutions. The experiments are based on three different real-world traffic with generated attack traces: (a) CAIDA+Attack.trace; (b) LBNL+Attack.trace; and (c) Campus+Attack.trace. . . . . 57
4.5 (a) Packet processing time of each detection program in three scenarios. (b) Packet processing time reduced by vNIDS compared with sharing all detection states. . . . 58
4.6 Launch time (a) and processing capacity (b) of *header-based detection*, *protocol parse*, *payload-based detection*, and Monolithic NIDS instances. . . . . 58
4.7 (a) Traditional NIDS requires rerouting traffic of Applications B back to *Site-1* for processing. (b) vNIDS can provision its instances flexibly at *Site-2*, if Applications B have migrated to *Site-2*. . . . . 59
4.8 Required bandwidth between *Site-1* and *Site-2* for vNIDS and traditional NIDS respectively. . . . . 59
4.9 (a) Runtime throughput of vNIDS and Bro Cluster. (b) Number of instances of vNIDS and Bro Cluster. . . . . 60

5.1 Overview of system flow. . . . . 63
5.2 Syntax of system flow rule. . . . . 66
5.3 CDF of the latency of file read operation with various numbers of system flow rules installed. . . . . 71
5.4 CDF of the latency of file write operation with various numbers of system flow rules installed. . . . . 72

5.5 CDF of the socket throughput with various numbers of system flow rules installed. . 72

6.1 Shell commands executed during Mirai infection (left), and the five phases of IoT malware infection process that we identified (right). . . . . . . . . . . . . . . . . . . . 76
6.2 Examples of shell commands invoked by an infection shell script (upper) and a malicious ELF file (bottom) in our dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . 78
6.3 Modeling approach overview. Firstly, we generate the CFGs of infection and benign scripts in our dataset (1). Then we build an ISM that represent the behavior patterns of malware infection (2). After that, we assign weights to the ISM through a correlation analysis (3). The generated WISM works as a general model of malware infection. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
6.4 Building the ISM from shell commands in infection scripts. Refer to Table 6.3 for abbreviations of the capabilities in the ISM. . . . . . . . . . . . . . . . . . . . . . . . 89
6.5 Overview of SOTER implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
6.6 CDFs of risk scores of all benign scripts and all infection scripts. . . . . . . . . . . . 95
6.7 Geo-distribution of our deployed software IoT Devices. . . . . . . . . . . . . . . . . . 98
6.8 The testing results **(a)** and the ROC curve **(b)** of our infection detection model over infection scripts unseen during the training process. . . . . . . . . . . . . . . . . . . 100
6.9 Average CPU loads of three types of IoT platforms without human interaction **(a)** and with human interaction **(b)**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.10 Overview of infection preventor implementation. . . . . . . . . . . . . . . . . . . . . 104
6.11 The system flow rule that is defined to implement the scheduling sequence that PROCESS GAURD is invoked after SOTER. . . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.12 The system flow rule that will be generated by PROCESS GAURD. $pid is the process id passed from SOTER. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.13 Average CPU loads of three types of IoT platforms without human interaction **(a)** and with human interaction **(b)**. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
6.14 Infrastructure-wide architecture of cross-level IoT malware defense approach. . . . . 107
6.15 Key components of cross-level IoT malware defense security function. . . . . . . . . 108
6.16 Network flow rule generated by cross-level IoT malware defense security function to block scanning traffic. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.17 System flow rule generated by cross-level IoT malware defense security function to quarantine all processes that access sockets. . . . . . . . . . . . . . . . . . . . . . . 108
6.18 Geo-distribution of our deployed software IoT Devices. . . . . . . . . . . . . . . . . . 110
6.19 Cross-level defender response latency to each port scan incident. . . . . . . . . . . . 111

# Chapter 1

# Introduction

## 1.1 Problem Motivation

Software-Defined Infrastructure (SDI) [83] has become a popular design paradigm of modern infrastructure in today's data centers, enterprise compute clusters, and public or private clouds. In the industry, many large service providers have announced their interests or solutions to deploying SDI [159, 169, 157, 160]. Among the examples of SDI are Software-Defined Networking (SDN) [106], Network Function Virtualization (NFV) [69], and Software-Defined Radios (SDR) [148]. The key idea of SDI is the introduction of a software-defined control layer, which is highly programmable.

Thanks to SDI, the computing resources throughout the entire infrastructure can now be elastically controlled, easily reconfigured, and programmatically extended. For example, NFV implements network security functions as software instances that can be created or destroyed rapidly to handle attack traffic volume variations on demand; SDN can dynamically redistribute the traffic to support flexible placement of network security function via a well-defined interface, such as OpenFlow [106], which makes the reconfiguration of the entire network much easier; besides, SDN and NFV provide open standards that allow the operators to extend applications through a programmatic way.

Although SDI brings significant benefits to modern infrastructure and attracts great attention from industry service providers, the corresponding security mechanisms that are suitable to protect the modern infrastructure are still underdeveloped. In the network level, traditional hardware-based network functions are often placed at fixed network entry points and have a constant capacity

with respect to the maximum amount of traffic they can process. Such rigid nature renders them awkward in protecting SDI. In the host level, many existing host security approaches [155, 147, 154] mainly base on simple (binary) access control actions, i.e., explicitly allowing or denying specific system events or activities. The limitation of expressiveness in security directives greatly hinders the innovation on host security functions.

## 1.2 Research Motivation

In the network level, some pioneered researches [135, 140, 165, 59] have made use of the advantages of SDN and NFV. However, the great challenges inherent in the management, design, deployment, and execution of network security functions prevents the full use of the benefits of SDN and NFV. In particular, the design of two critical network security functions, the firewalls and NIDSes, must be improved. First, when the firewalls and NIDSes are overloaded from large traffic loads, they require scale out [130, 64]. Similarly, when firewalls and NIDSes are underloaded, they require scale in. This scaling of the two network security functions must be safe and efficient. A safe scaling of firewalls and NIDSes prevents any loss of legal traffic, the allowance of illegal traffic, and ensures the effective detection of any potential attacks. An efficient scaling of firewalls and NIDSes ensures a bounded latency overhead caused by scaling and consumes minimum compute and network resources. Second, traditional NIDSes [30, 29] are typically designed as a monolithic piece of software, which can lead to inflexibility and a significant waste of resources [172].

In the host level, prevention of suspicious or malicious system activities mainly by enforcing simple access control (i.e., allow or deny), such as the process-level firewall [155] and information flow control systems [10, 1], may hinder further innovating in security applications under some circumstances. For instance, if we monitor a pending access originated from a suspicious process to a file that contains sensitive information, the simple access control actions can only block such access to prevent potential compromise, which may hinder further investigations of the harmfulness of the process and its attack patterns. To facilitate innovative security practice, more expressive security directives must be offered to enable more fine-grained response to threats in addition to simple access control. Ideally, such security directives should be reusable as a countermeasure library that could be coupled with different host security functions.

## 1.3 Research Objective

In this dissertation, our research goal is to fill the gap between the dynamic nature of SDI and the ossified traditional security functions by enabling network-level and host-level programmable security, which features *elastic*, *programmatically-extensible*, and *easily-reconfigurable* security functions. Through the research, we will answer the following research questions.

- Enable network-level programmable security
  - **RQ1:** How can we elastically scale the firewall in a safe and efficient way?
  - **RQ2:** How can we virtualize the NIDS in a safe and efficient way?
- Enable host-level programmable security
  - **RQ3:** How can we enable expressive security directives for host security functions?
  - **RQ4:** Can we build host security functions with expressive security directives?
- Enable cross-level security interaction
  - **RQ5:** Can we build security functions that take the advantages of the network-level and host-level programmable security at the same time?

## 1.4 Overview of Research Tasks

To achieve the research goal, we propose to develop a framework called ProSec to support network-level programmable security, host-level programmable security, and cross-level security interactions. ProSec consists of three piece of work shown in Figure 6.3.

### 1.4.1 Task1: Enable safe and efficient virtual firewall elasticity control (RQ1)

In the network level, we start from the most widely used network security function, the firewalls. To make use of the benefits of SDN and NFV introducing more flexibility to firewalls with respect to processing capacity and placement location, we will implement firewalls as software

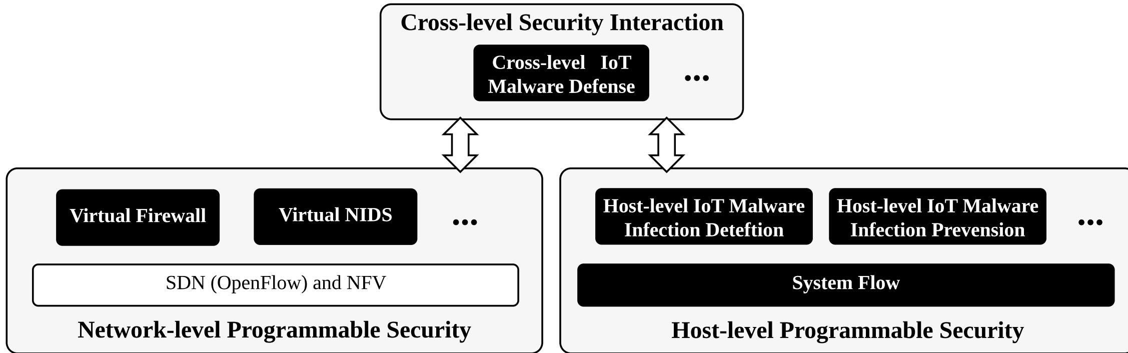



Figure 1.1: ProSec consists of three major piece of work, enabling network-level programmable security, host-level programmable security, and cross-level security interaction, respectively. The black blocks indicate the components that will be contributed by this dissertation.

instances (a.k.a virtual firewalls), which can be created or destroyed rapidly on demand. We identified four key challenges – semantic consistency, correct flow update, buffer overflow avoidance, and optimal scaling – that must be solved in order to safely and efficiently scale virtual firewalls.

To address the challenge of *semantic consistency*, we will apply packet space analysis to identify intra-dependencies of firewall rules. A group-based migration strategy will be applied to guarantee the semantic consistency of security policies. To find *correct flow update*, intra-dependencies of flow rules and inter-dependencies between firewall rules and flow rules will be identified, which help locate the subset of flow rules to update and the correct update operations (e.g., change, insertion). To avoid *buffer overflow*, we will model migration process and predict the amount of in-flight traffic generated during the migration. Firewall rules that may cause buffer overflow during migrating are not selected to move. To achieve *optimal scaling-out* of virtual firewalls, we will adopt a three-step heuristic approach to minimize resource usage. To achieve *optimal scaling-in*, integer linear programming (ILP) will be used to ensure maximum resources can be released.

### 1.4.2 Task2: Achieve safe and efficient NIDS virtualization (RQ2)

NIDS is another critical network security function that monitors the traffic in a network to detect malicious activities or security policy violations. As NIDSes are much more complicated than firewalls, it is difficult to virtualize NIDSes leveraging SDN and NFV. Especially, in edge computing [132, 114] where computing resources are limited and computing environments are diverse, traditional NIDSes can hardly be tailored to reduce the footprint while provide proper protection

since they are designed as a monolith. We identified tow key challenges, effective intrusion detection and non-monolithic NIDS provisioning, that must be addressed to achieve safe and efficient virtualization of NIDSes.

To address the *effective intrusion detection* challenge, we will classify detection states of virtualized NIDSes into local and global detection states to minimize the amount of detection states shared between instances. The local detection states are only accessed within a single instance, while the global detection states can be shared among multiple instances. We would like to provide a guideline to classify detection states, and employ static program analysis and NIDS domain-specific knowledge to automatically achieve detection state classification. To address the *non-monolithic NIDS provisioning* challenge, virtual NIDSes will be provisioned as microservices [51]. We will decompose NIDSes into three microservices, *header-based detection*, *protocol parse*, and *payload-based detection*. Each microservice can be instantiated independently, provisioned at different locations with a different number of instances, and configured to execute different detection programs. Multiple microservices can be chained up in different ways as "microservice chains" that provide flexible detection capabilities. To automate the decomposition, we will design an algorithm that leverages program slicing technique to automatically partition a detection logic so that the detection logic can be transparently executed by microservices.

### 1.4.3 Task3: Enable expressive security directives for host security functions (RQ3)

In the host level, we expect the security directives can support more fine-grained response actions to the events in a host. In addition, those expressive security directives should be reused to implement different host security applications regardless the low-level operating system details.

To make the security directives support more fine-grained response actions, we will define the security directives in a "Match-Action" paradigm. The match field of a security directive indicates what events this security directive responds to. The action field depicts what actions will be taken against the matched events. A list of actions will be defined in this research based on existing commonly used host security applications. To make those expressive security directives reusable by different host security applications, we will abstract the operating system events using a flow-based model, called system flow. Based on system flow, the events in a host will be formalized with a

three-tuple, including the source that generates the events, the destination that the event goes to, and the operation enforced by the source.

### 1.4.4 Task4: Case study: defending malware in IoT with ProSec (RQ4 & RQ5)

To demonstrate the benefits provided by ProSec, we first develop an IoT malware infection detection approach and an IoT malware infection prevention approaches that utilize the new fine-grained response actions to the events in a host. The IoT malware infection detection approach relies on a model that is trained based on benign and infection samples. To satisfy the limited resource requirement of IoT devices, we implement a malware infection detector monitoring the events abstracted by system flow that can be run on small computing devices, such as Raspberry Pi. The IoT malware infection prevention approach is implemented based on the system flow model. Then, we implement a port scan detection and prevention approach at the network level. In particular, we add port scan detection logic into virtual NIDSes, and use virtual firewall to block the detected port scan traffic. Finally, we develop a cross-level IoT malware defense approach that enable the collaboration between network and host security functions. Through a northbound interface, virtual NIDSes in the network can send their detection results to the cross-level IoT malware defense security function. The cross-level IoT malware defense security function then enforce access control rule in both network and host levels. Once the cross-level IoT malware defense security function receives port scan alerts from virtual NIDSes, it will update the firewall rules in virtual firewalls. The system flow rules of the infected host will be updated as well to prevent the IoT malware from accessing network resources.

# Chapter 2

# Background and Related Work

## 2.1 Researches on Network Security Functions

### 2.1.1 Firewall Policy Analysis

Detecting and solving anomalies in firewall policies concerns with ensuring the semantic correctness of firewall policies. The major purposes of firewall policy analysis are detecting anomalies reducing firewall rule table size. Lihua Yuan et al. present a toolkit in [168] to detect misconfigurations in firewall policies using static analysis. Ehab S. Al-Shaer et al. propose a tool in [32] to detect the anomalies between two firewall rules. Hongxin Hu et al. identify four typical firewall policies anomalies and invent a novel presentation method to assist the firewall operators to detect and solve the firewall anomalies in [75]. Except for anomalies in firewall policies, when the access control logic gets complex and consists of a large number of overlapping firewall rules, there may be redundant firewall rules. Alex X. Liu et al. proposes an algorithm to minimize the size of firewall rule table in [98]. The same authors later apply their algorithm on packet classifiers on ternary content-addressable memory [97]. To evaluate firewall optimization techniques, a benchmarking framework has been proposed in [111]. Zhang et al. [171] demonstrated that careless policy updates may result in security violations. They presented safe and efficient policy update algorithms for firewall policy updates. But the proposed algorithms are only able to deal with policy updates on a single firewall.

### 2.1.2 Scalable NIDSes

Existing work of NIDSes is devoted to developing scalable architectures for NIDSes [139, 46, 150]. The authors of [46] propose a concurrency model for stateful deep packet inspection to enable NIDSes to run with multiple threads in parallel for better scalability. Their approach attempts to avoid inter-thread communication to reduce the performance overhead induced by detection state sharing. However, such an approach proposes to maintain detection states in each instance separately, without any sharing. Unfortunately, for complex detection logics, it is impractical to achieve effective intrusion detection only relying on traffic distribution. NIDS Cluster [150] and multi-core architecture for NIDS [139] enable information exchange between NIDS nodes or threads to achieve effective intrusion detection. However, both of them fail to give any analysis or classification to the detection states. In particular, NIDS Cluster relies on users to manually determine which detection states need to be shared.

Another line to scale the NIDSes is to use GPU to process the traffic in parallel. But this approach can work only on signature-based NIDSes, which process the traffic by byte-level string matching. Cheng-Huang Lin et al. proposes an algorithm to accelerate the string matching on GPU [94]. Giorgos Vasiliadis et al., proposes a novel architecture for Snort, one of the most widely-used open source signagure-based NIDSes, to be deployed with GPUs in [152].

### 2.1.3 Network Security Functions Employing SDN & NFV

In Bohatei [59], Seyed Kaveh Fayaz et al. address the limitations of traditional DDoS defense systems with respect to flexibility and elasticity. Bohatei utilizes NFV to dynamically launch virtual machines at different locations to handle DDoS attack traffic and leverages SDN to redistribute the traffic that optimize the bandwidth consumption. However, Bohatei is specific for DDoS attacks and cannot tackel other types of attacks. Tianlong Yu et al. in PSI [165] propose a security architecture for enterprise networks leveraging NFV and SDN to provide better isolation between security functions, support context-aware security policies, and agilely change its security policies.But the design of each specific network security function remains unsolved in this work. Seung Won Shin et al. develop a platform on top of SDN controller to support modular composable security service development in [135]. However, this work can only accommodate security services that are implemented as SDN applications. John Sonchack et al. in OFX [140] propose to extend

the SDN switches so that they can run compiled network security functions that are defined in the SDN controller.

## 2.2 Researches on Host Security Functions

### 2.2.1 Provenance Analysis Systems

Many prior works record system events into audit logs [6, 90, 66, 56] and leverage forensic analysis to investigate and reconstruct intrusions and attacks [81, 87, 137]. Recently, system-level data provenance techniques [136] are used to investigate system-wide attacks. Many prior works are proposed to efficiently collect system run-time information for a holistic view of the operations of a system over time. The collected information will further be analyzed at run-time or offline to investigate suspicious/malicious activities. For example, PASS [117] and Hi-Fi [125] can efficiently capture and store system-level provenance. Linux Provenance Modules (LPM) [35] allows developers to write custom provenance rules to constrain system information flows. These systems are limited to coarse-grained provenance, which could generate many false dependencies. To reduce false positives and logging sizes, DataTracker [142] utilizes dynamic information flow tracking for finer granularity and Rain [79] adopts record-and-reply technology to determine attack causalities.

In contrast, network provenance systems [80, 162, 174, 173] utilize the interaction between networked hosts since they lack detailed system-level information. For example, Rtag [80] proposes a cross-host information flow tagging and tracking that enables practical attack investigations across multiple hosts in infrastructure by taking advantage of system-call-level provenance information. However, those approaches are not designed to provide a framework to program security apps to dynamically respond to security dynamics in infrastructure.

### 2.2.2 Information Flow Control Systems

Some previous works, such as SELinux [10], HiStar [170], Flume [89], Asbestos [55] and Weir [118], propose to enforce Decentralized Information Flow Control (DIFC) at the OS level by using labels to define security and integrity contexts and restrict information flows between kernel objects. Those works abstract information flow in a graph-based representation to detect malicious/suspicious activities in a system. In addition, dynamic taint tracking systems are proposed to

solve security problems. Some of them [53, 164, 52, 175] detect data misuses by tainting/coloring sensitive data and resources in the system. However, such information flow control based solutions based on the information flow abstraction only have single-system visibility and limited programmability for system security applications (e.g., the detection of data misuse). In addition, some works such as Silverline [116] and PivotWall [120] extend information flow tracking on system resources with network-level visibility/control, which provide cross-host visibility. Yet, their approaches are not designed to support a programmable framework for diverse security applications throughout the infrastructure.

### 2.2.3 System Resource Access Control

There are a couple of prior works proposed to enable access control towards critical system resources. Existing Unix/Linux systems embed security kernel modules (e.g., AppArmor [1]) to allow users to write mandatory access control (MAC) policies to prevent unwanted accesses to protected system resources. Also, several system security applications are proposed to defend against unauthorized accesses to resources via race conditions, e.g., TOCTOU attacks [36, 42, 47, 147] and Link Following attacks [154, 40]. For enabling fine-grained access control, the process-level firewall [155, 153] is proposed to prevent system resource attacks by enforcing more fine-grained access control. However, compared with , they have some major limitations: first, they cannot effectively defend against cross-host attacks due to their limited view of events from a single system; second, they only provide limited threat response strategies and can not support advanced security intents, e.g., cyber deception defense.

### 2.2.4 Security Monitoring Systems

Many security monitoring systems monitor security incidents in infrastructure and report/alert indicators of compromise. However, compared with , they mostly focus on single-system events and do not enforce real-time detection of attacks. Alert correlation is proposed by many existing works [126, 149, 57] and Security Information and Event Management (SIEM) systems, such as Splunk [11], LogRhythm [8] and IBM QRadar [4], which correlate log events collected from various sources by using different indicators of compromise. Also, some previous studies [54, 108, 124, 58] also target real-time attack story constructions from system-level logs. However, collecting logs

from various network/security devices and networked hosts, as well as analyzing and storing the logs, are complex, expensive, and non-trivial tasks that need significant efforts from security experts. Instead, can program the entire operations without log-related burden and enable rapid threat responses against detected threats in addition to real-time analysis of threat.

## 2.3 Researches on Malware Detection

### 2.3.1 Network-based Solution

Network-based security solution has been commonly used for protecting IoT systems [76, 49, 110, 119]. Gu [68] presented BotHunter, a dialog correlation method that utilizes malware-specific signatures to recognize the malware infection for botnet detection in traditional networks. However, IoT network traffic is device-specific and depends on different environmental settings. Due to the diversity of IoT devices and manufacturers, it is impractical or non-scalable to create malware signatures [166]. It is also challenging to define normal baselines of IoT networks for anomaly detection. In addition, IoT Botnets have continued to evolve and adapt to the advanced techniques. For example, a newer version of Mirai, DvrHelper is the first malware designed to bypass an anti-DDoS solution by using challenge-response policies and shared Google reCAPTCHA response token [22]. Although new network-based detection methods for IoT malware are constantly being proposed [110, 107, 119], they still cannot collect more detailed activities that occur in IoT devices. Our detection approach is able to discovery detailed information about infection activities of IoT malware, which could be a promising complement to existing network-based solutions.

### 2.3.2 Host-based Solution

To secure IoT devices, the preferred solution is to update and patch buggy firmware for these vulnerable devices. However, due to lack of suitable facilities, it is difficult to keep track of the available patches and apply them to all unpatched devices. Also, not all devices are compatible with the available updates due to their outdated hardware. Considering the constraints of limited resources, only a few studies have focused on host-based IoT security solution. Sun [144] proposed a cloud-based detection with reversible sketch for resource-constrained IoT devices to improve the security of the devices. Abbas [31] proposed a low complexity signature-based method for IoT

devices that only identifies and stores a subset of signatures to detect a group of malware. Su [143] proposed a light-weight detection method for IoT malware, based on a local and cloud-based malware detector. However, all those solutions require the installation of software in IoT devices, not all devices can afford such runtime overhead.

# Chapter 3

# Safe and Efficient Virtual Firewall Elasticity Control[1]

## 3.1 Introduction

Traditional hardware-based firewall appliances are placed at fixed locations with fixed capacity. Such nature makes them difficult to protect today's prevailing virtualized environments. Two emerging networking paradigms, SDN and NFV, offer the potential to address these limitations. NFV envisions to implement firewall function as software instance also known as virtual firewall. Virtual firewalls provide great flexibility and elasticity, which are necessary to protect virtualized environments in SDI. In this task, we implement firewalls as software instances and build an innovative virtual firewall controller, `VFW Controller`, to enable safe, efficient and cost-effective virtual firewall elasticity control. `VFW Controller` addresses four key challenges with respect to *semantic consistency*, *correct flow update*, *buffer overflow avoidance*, and *optimal scaling* in virtual firewall scaling. To demonstrate the feasibility of our approach, we implement software firewall instances using the Click element. Then we implement the core components of `VFW Controller` on top of ClickOS [105], an operating system optimized for NFV. We test our virtual firewall and `VFW Controller` in an SDN environment.

[1] This work has been published in Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017).

## 3.2 Research Challenges

The four technical challenges that must be addressed to achieve safe, efficient, and optimal virtual firewall scaling are as follows.

- **Semantic Consistency** The split and mergence of firewall rules in virtual firewall scaling must not change the semantics of security policies. Otherwise *safety* is violated. Keeping the semantic consistency after rounds of splits and mergences is non-trivial, because firewall rules are often logically entangled with each other resulting in complicated rule dependencies. Sometimes rule dependencies are intentionally introduced by system administrators to obtain fewer firewall rules [75] [168].

- **Correct Flow Update** When migrating firewall rules and states to new/remaining firewall instances, network flow rules in SDN switches must be correctly updated to re-route corresponding traffic to new/remaining instances. Incorrect flow update may cause some traffic is missent to an instance that does not have the firewall rules intended for the traffic, hence violates *safety*. Finding correct flow update is difficult, since flow rules in an SDN switch may be dependent [84] [85] and the traffic space defined by the flow rules often does not exactly match that defined by the firewall rules.

- **Buffer Overflow Avoidance** A *safe* scaling also requires buffering in-flight traffic during a migration [63] [64]. In-flight traffic refers to the traffic that arrive at the source instance after the matching firewall rules and states have been migrated, or the traffic that arrive at the destination instance before corresponding firewall rules and states become ready. However, buffer space is not unlimited. We also observe that migration of different firewall rules incurs different amount of in-flight traffic. Therefore, care must be taken while selecting firewall rules to migrate so that buffer overflow.

- **Optimal Scaling** Compute and network resources for building virtual firewalls are neither unlimited nor free. Resource optimization is an important goal that needs to be achieved in virtualized environments. Creations of virtual firewalls consume compute resource, and flow rule updates are constrained by the limited capacity of the Ternary Content Addressable Memories (TCAMs) used for holding rules in each SDN switch. It is NP-hard to achieve optimization of resource usage during virtual firewall scaling.

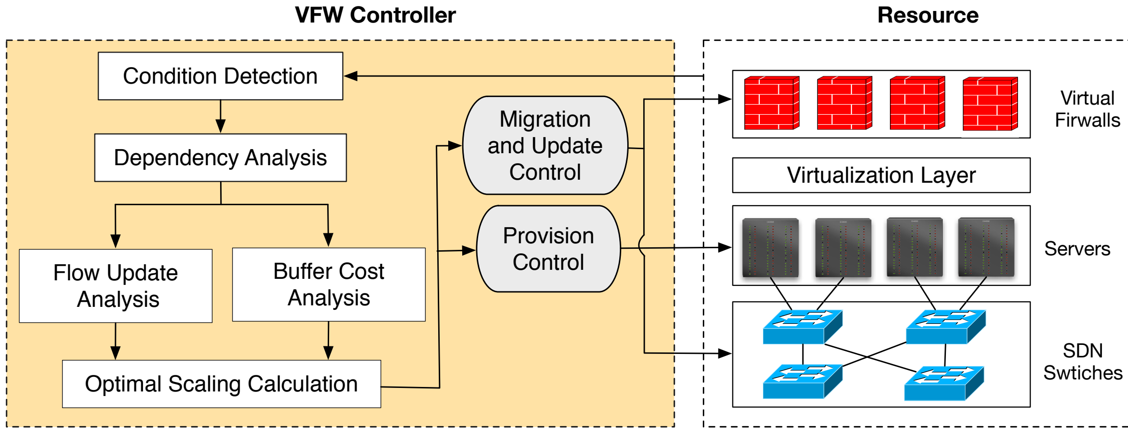



Figure 3.1: `VFW Controller` components and workflow.

## 3.3 Methodology

### 3.3.1 High-level View of Virtual Firewalls

The components and workflow of our virtual firewall are shown in Figure 3.1. The virtual firewall consists of software instances that run in the data plane and a controller called `VFW Controller` that run on a centralized server.

Software instances are provisioned on top of a virtualization layer, such as virtual machines or containers. They are responsible to processing the network traffic and enforce firewall rules. `VFW Controller` monitors each virtual firewall and detects traffic overload and underload conditions. Once a condition is detected, `VFW Controller` first performs *Dependency Analysis*, *Flow Update Analysis*, and *Buffer Cost Analysis*. Those analysis results are utilized by *Optimal Scaling Calculation*. Then, *Provision Control* and *Migration and Update Control* interact with the compute and network resources and execute virtual firewall scaling.

*Dependency Analysis* in `VFW Controller` identifies three dependency relations: (1) the dependency relations of firewall rules on the virtual firewalls; (2) the dependency relations of the flow rules in SDN switches; and (3) the inter-dependency relations between the firewall rules and the flow rules. Understanding the dependency relations of the firewall rules is critical to ensure the semantic consistency after scaling. The dependency relations of the flow rules and the inter-dependency relations serve to find the flow rules to be updated. *Dependency Analysis* divides both firewall rules and flow rules into groups based on their dependency relations.

*Flow Update Analysis* determines the correct update of flow rules and the update cost for each firewall rule group. *Buffer Cost Analysis* predicts the amount of in-flight traffic generated by the migration of each firewall rule group. This prediction is necessary to avoid buffer overflow.

*Optimal Scaling Calculation* component considers previous analyses and uses a three-step heuristic approach to determine, in the case of overload, (1) the minimum number of new instances to be created; (2) selective firewall rule groups to be migrated to each new instance; and (3) flow rules to be updated. This approach also achieves minimum update cost and no buffer overflow. In the case of underload, *Optimal Scaling Calculation* component uses an ILP approach to determine (1) which instances are to be killed among all underloaded instances; (2) how to redistribute firewall rule groups; and (3) corresponding flow rule updates. The ILP approach guarantees that the most resources are released after the mergence of instances.

*Provision Control* creates or deletes instances according the calculation results from *Optimal Scaling Calculation* component. *Migration and Update Control* coordinates the migration of firewall rules and states, and flow rule updates.

### 3.3.2 Dependency Analysis And Semantic Consistency

To analyze the dependency, we first define some concepts that are used by dependency analysis.

**Definition 1** (Packet space). *Packet space of a rule $r$, denoted as $PS(r)$, is defined as a 5-dimensional hyperspace with dimensions being protocol, source IP, source port, destination IP, destination port.*

**Definition 2** (Direct dependency). *Two rules $r_i$ and $r_j$ in a rule set $\mathbb{R}$ are directly dependent iff $PS(r_i) \cap PS(r_j) \neq \emptyset$, where $PS(r_i)$ is the packet space defined by $r_i$, and $PS(r_j)$ is the packet space defined by $r_j$.*

**Definition 3** (Indirect dependency). *Two rules $r_i$ and $r_j$ in a rule set $\mathbb{R}$ are indirectly dependent iff $PS(r_i) \cap PS(r_j) = \emptyset$ and there exists a subset $R \subseteq \mathbb{R} \backslash \{r_i, r_j\}$ such that $PS(r_i) \cap PS(R) \neq \emptyset$ and $PS(r_j) \cap PS(R) \neq \emptyset$.*

We identify the relation between a firewall rule group $V$ and a flow rule group $F$ is one of the following:

- Independency *iff* $PS(V) \cap PS(F) = \emptyset$. We denote independency as $V \xrightarrow{ind} F$.

**Algorithm 1:** Partitioning of a rule set to disjoint groups.

```
   Input: R, a set of ordered rules to be partitioned;
          𝔾 = ∅, a set, initially set to be empty, to store
          rule groups.
   Output: 𝔾, the set of rule groups;
 1 foreach r ∈ R do
 2   Γ = ∅; /* A set to store preceding rules that has dependency relation
        with r . Initially ∅.                                          */
 3   if 𝔾 = ∅ then
       /* That is to say r is the first rule in R                      */
 4     Γ.Append(r);
 5   else
 6     PS(r) ⟵ PacketSpace(r);
 7     foreach G ∈ 𝔾 do
 8       foreach r_i ∈ G do
 9         PS(r_i) ⟵ PacketSpace(r_i);
10         if PS(r) ∩ PS(r_i) ≠ ∅ then
11           Γ.Append(G);/* Sequentially append each rule in group G to
                Γ.                                                     */
12           𝔾.Remove(G);/* Remove group G from 𝔾.                     */
13           break;
       /* Now all the rules that precede r and have dependency relation
          with r are stored in Γ.                                      */
14     Γ.Append(r); /* Append r to Γ.                                  */
15   𝔾.Add(Γ); /* Add Γ as a new group to 𝔾.                           */
16 return 𝔾;
```

- Congruence *iff* $PS(V) = PS(F)$. We denote congruence as $V \xrightarrow{con} F$.
- Superspace *iff* $PS(V) \supset PS(F)$. We denote superspace as $V \xrightarrow{sup} F$.
- Subspace *iff* $PS(V) \subset PS(F)$. We denote subspace as $V \xrightarrow{sub} F$.
- Intersection *iff* $PS(V) \cap PS(F) \subset PS(V)$ and $PS(V) \cap PS(F) \subset PS(F)$. We denote intersection as $V \xrightarrow{int} F$.

**Definition 4** (Inter-dependency). *A firewall rule group $V$ and a flow rule group $F$ are inter-dependent if $PS(V) \cap PS(F) \neq \emptyset$.*

**Definition 5** (Direct association). *Two firewall rule groups $V_i$ and $V_j$ in a firewall rule set $\mathbb{V}$ are directly associated iff there exists a flow rule group $F$ in a flow rule set $\mathbb{F}$ such that $V_i$ and $F$ are inter-dependent, and $V_j$ and $F$ are inter-dependent. We use the notation $V_i \xrightarrow{dir\text{-}asso} V_j$ to denote $V_i$ and $V_j$ are directly associated.*

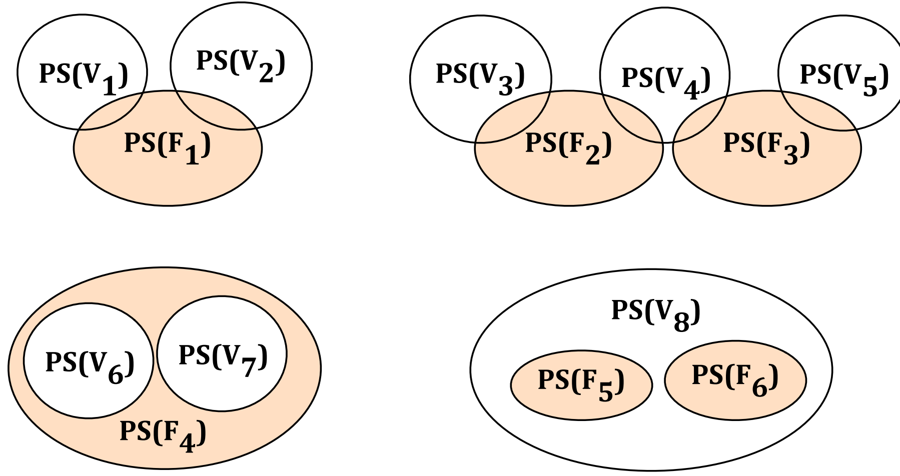



Figure 3.2: Example of firewall rule associations and classes.

**Definition 6** (Indirect association). *Two firewall rule groups $V_i$ and $V_j$ in a firewall rule set $\mathbb{V}$ are indirectly associated iff they are not directly associated and there exists $\bar{V} \subseteq \mathbb{V} \backslash \{V_i, V_j\}$, such that $V_i$ and $\bar{V}$ are directly associated, and $V_j$ and $\bar{V}$ are directly associated.*

**Definition 7** (Firewall Rule Class). *A firewall rule class is the union of firewall rule groups that are directly or indirectly associated.*

According to the definition, we now have identified the dependencies within the firewall policies. To ensure the semantic consistency, we employ a group-based migration strategy to move the firewall rules during virtual firewall live migration.

**Group-Based Migration Strategy**: *To guarantee semantic consistency, firewall rules in a group are migrated to the same destination virtual firewall instance and are in the same order as they are in the source virtual firewall instance. The destination virtual firewall instance can only start to process traffic matching rules in a group until all the rules and flow states associated with the group are ready on the destination instance.*

### 3.3.3 Flow Update Analysis

We first define two types of operations on the flow rules.

Let $V_i \in \mathbb{V}$ be the firewall rule group to be migrated from a source virtual firewall $\text{VFW}_1$ to a destination virtual firewall $\text{VFW}_2$. The key SDN switch is the last SDN switch, through which all the traffic matching $V_i$ will pass before diverging on their paths to reach $\text{VFW}_1$ and $\text{VFW}_2$. In what

**Algorithm 2:** Classification of firewall rule groups to classes.

**Input:** $\mathbb{V} = \{V_1, ..., V_m\}$, a set of firewall rule groups; $\mathbb{F} = \{F_1, ..., F_n\}$, a set of flow groups;
$\mathbb{C} = \emptyset$, a set of firewall rule classes which is initially set to be empty;

**Output:** $\mathbb{C}$;

/* Identify direct associations. */

1 **foreach** $F \in \mathbb{F}$ **do**

2    $\bar{V} = \emptyset$; /* A set to store firewall rule groups that are directed associated via $F$, initially set to be empty; */

3    **foreach** $V \in \mathbb{V}$ **do**

4      **if** *$V$ and $F$ are inter-dependent* **then**

5        $\bar{V}.Append(V)$;/* Each firewall rule in $V$ is sequentially appended to $\bar{V}$. */

6    **if** $\bar{V} \neq \emptyset$ **then**

7      $\mathbb{C}.Add(\bar{V})$;/* $\bar{V}$ is added to $\mathbb{C}$ as a class. */

/* Identify indirect associations. Any two classes that intersect are united to make a bigger class. */

8 **foreach** $\bar{V}_i \in \mathbb{C}$ *and* $\bar{V}_j \in \mathbb{C}$ **do**

9    **if** $\bar{V}_i \cap \bar{V}_j \neq \emptyset$ **then**

10      $\bar{V}_i.Append(\bar{V}_j)$; /* Each firewall rule in $\bar{V}_j$ is sequentially appended to $\bar{V}_i$. */

11      $\mathbb{C}.Remove(\bar{V}_j)$;

12 **return** $\mathbb{C}$;

follows, we denote such key SDN switch as $SW$. Figure 3.3 shows $SW$ switches in different cases: $(a)$ VFW$_1$ and VFW$_2$ are on the same server; $(b)$ VFW$_1$ and VFW$_2$ are in different servers on the same rack; and $(c)$ VFW$_1$ and VFW$_2$ are in different racks. In case $(a)$, the $SW$ is an open vSwitch on the server. In case $(b)$, the $SW$ is the Top-of-Rack (ToR) switch that connects the two servers. In case $(c)$, the $SW$ is the Aggregation Switch that connects the two racks. We consider case $(a)$ will be the most common, as it introduces the least traffic overload and update efforts. Only when two virtual firewalls cannot be placed on the same sever due to resource inadequacy, they will be placed on separate servers.

Let $\mathbb{F} = \{F_1, ..., F_n\}$ be the set of flow groups (obtained using the partition algorithm in §**??**) on $SW$. To find the updates on $\mathbb{F}$, `VFW Controller` iterates through $\mathbb{F} = \{F_1, ..., F_n\}$ sequentially, and compares the inter-dependency relation between $V_i$ and each $F_j \in \mathbb{F}$ to determine the updates. We identify two types of update operations: **CHANGE** and **INSERT**.

- If $V_i \xrightarrow{ind} F_j$, no update is required.

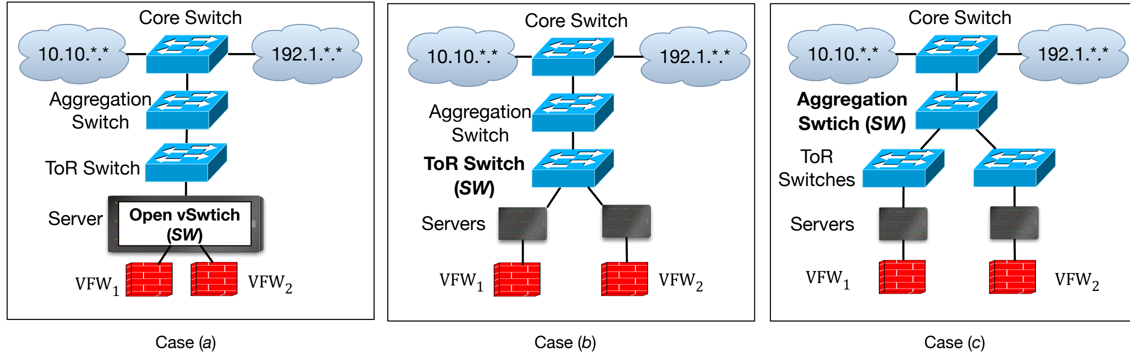



Figure 3.3: $SW$ in three different cases with respect to a classic 3-tire cloud architecture.

- If $V_i \overset{con}{-} F_j$ or $V_i \overset{sup}{-} F_j$, only CHANGE operation is required. For every flow rule $f \in F_j$, if its forwarding actions contain 'send to $\text{VFW}_1$', the forwarding action is changed to 'send to $\text{VFW}_2$'. Note that the same flow rule may contain forwarding actions for the routing intentions of other applications (see the example flow rule $f_3$ in Figure **??**). Flow update should not change any of those actions.

- If $V_i \overset{sub}{-} F_j$ or $V_i \overset{int}{-} F_j$, both CHANGE and INSERT operations are required. Comparisons between $PS(v)$, the packet space defined by each $v \in V_i$, and $PS(f)$, the packet space defined by each $f \in F_j$, will be performed in order to find the correct update.

  (1) If $PS(v) \cap PS(f) = \emptyset$, $f$ needs no update.

  (2) If $PS(v) \supseteq PS(f)$, then CHANGE operation is performed to change the 'send to $\text{VFW}_1$' action of $f$ to 'send to $\text{VFW}_2$'.

  (3) If $PS(v) \subset PS(f)$, then INSERT operation is performed. A new flow rule $f'$ is inserted right before $f$ to express that traffic matching $v$ is sent to $\text{VFW}_2$. Take the $v$ and $f$ below for example.

| | | | | | | |
|---|---|---|---|---|---|---|
| $v$ | tcp | 10.10.2.* | * | 192.1.1.9 | 80 | deny |
| $f$ | * | 10.10.2.* | * | 192.1.1.* | 80 | send to $\text{VFW}_1$ and $\text{IDS}_1$ |

  Then $f'$ below is inserted right before $f$.

| | | | | | | |
|---|---|---|---|---|---|---|
| $f'$ | tcp | 10.10.2.* | * | 192.1.1.9 | 80 | send to $\text{VFW}_2$ and $\text{IDS}_1$ |

(4) If $PS(v) \cap PS(f) \subset PS(v)$ and $PS(v) \cap PS(f) \subset PS(f)$, INSERT operation is performed. A new flow rule $f'$ is inserted right before $f$. Each field of $f'$ is the same as $f$, except that the protocol, source IP, source port, destination IP, destination port fields of $f'$ are the intersection of the respective fields of $v$ and $f$, and the forwarding action of $f'$ is 'send to VFW$_2$'. For example, for the following $v$ and $f$

| | | | | | | |
|---|---|---|---|---|---|---|
| $v$ | * | 10.10.2.* | * | 192.1.1.9 | 80 | deny |
| $f$ | * | 10.10.2.1 | * | 192.1.1.* | 80 | send to VFW$_1$ and IDS$_1$ |

the new flow rule $f'$ is:

| | | | | | | |
|---|---|---|---|---|---|---|
| $f'$ | * | 10.10.2.1 | * | 192.1.1.9 | 80 | send to VFW$_2$ and IDS$_1$ |

**Definition 8** (Update Cost). *The update cost of a firewall rule group is the total number of new flow rules inserted in SW during its update to support the migration of $V_i$.*

Let $\gamma_i$ be the update cost of $V_i$, then we have

$$\gamma_i = \sum_{j=1}^{n} \alpha_{ij}$$

There are two ways to reduce the total update cost:

- Select firewall rule groups that cause smaller update cost for migration.
- Analyze the relationships between firewall rule class and flow rule group. The comparison we have performed in previous section is pairwise between a firewall rule group $V_i$ and a flow group $F_j$.

We observed that new flow rules are inserted only when $V_i \overset{sub}{\longrightarrow} F_j$ or $V_i \overset{int}{\longrightarrow} F_j$. If there is a class $\bar{V}$ that consists of $V_i$ and other firewall rule groups and $\bar{V} \overset{sup}{\longrightarrow} F_j$, then no new flow rules will be inserted. To see why, consider a simple case as follows. Two firewall rule groups $V_1$ and $V_2$ are to be migrated. $F_j \in \mathbb{F}$ is one flow rule group in $SW$ and we have $V_1 \overset{int}{\longrightarrow} F_j$ and $V_2 \overset{int}{\longrightarrow} F_j$, and $V_1 \cup V_2 \overset{sup}{\longrightarrow} F_j$. Both INSERT and CHANGE operations are required for the migration of $V_1$ and $V_2$. Let $\alpha_{1j}$ be the number of new flow rules that will be inserted to $F_j$ in order to support the migration of $V_1$,

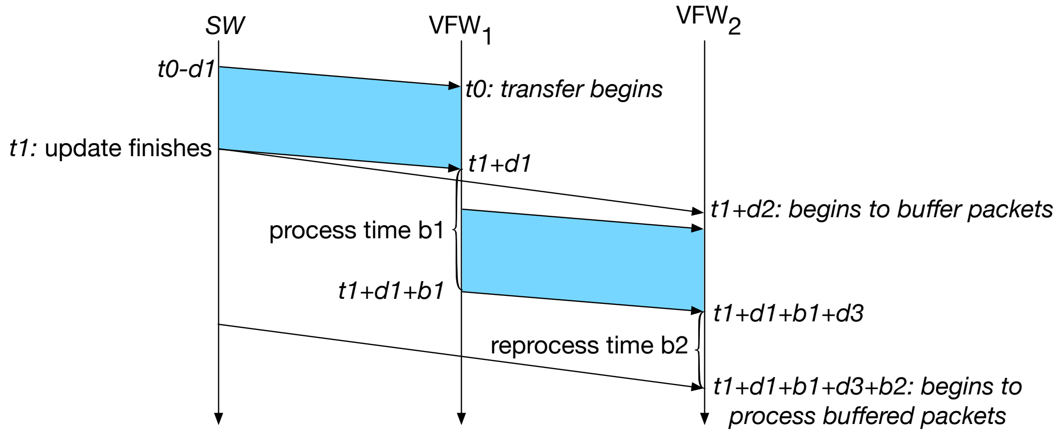



Figure 3.4: Packet paths during the migration.

and $\alpha_{2j}$ be the number of new flow rules needed in order to support the migration of $V_2$. Thus the total number of new flow rules to be inserted to $F_j$ is $\alpha_{1j} + \alpha_{2j}$. Let $\bar{V} = V_1 \cup V_2$, because $\bar{V} \xrightarrow{sup} F_j$, then no INSERT operation is required to support the migration of $\bar{V}$, hence the total number of new flow rules to be inserted is 0. Based on such observation, we propose the following class-based migration strategy in `VFW Controller` to reduce the total update cost.

**Class-Based Migration Strategy**: *If the update cost of migrating a class $\bar{V} = \cup_{i=1}^{m'} V_i$ is smaller than the sum of the update cost of each $V_i$ that constitutes $\bar{V}$, then $\bar{V}$ is migrated as a big group to the same destination virtual firewall to reduce update cost.*

### 3.3.4 Buffer Cost Analysis

**Definition 9** (Buffer Cost). *The buffer cost of a firewall rule group is the amount of in-flight traffic that are generated during the migration of the group.*

Figure 3.4 shows the packet paths during the migration of $V_i$. In Figure 3.4, we focus on the following time points.

- $t0$: is the time that $\text{VFW}_1$ starts to transfer the firewall rules and flow states specified in *fspace*;
- $t1$: is the time that $SW$ finishes the update;
- $d1$: is the transmission delay between $SW$ and $\text{VFW}_1$;
- $d2$: is the transmission delay between $SW$ and $\text{VFW}_2$;

- $d3$: the transmission delay between $\text{VFW}_1$ and $\text{VFW}_2$;
- $b1$: is the average time that $\text{VFW}_1$ spends processing a packet; and
- $b2$: is the average time that $\text{VFW}_2$ spends processing a packet.

At $\boldsymbol{SW}$, the traffic matching $V_i$ that arrives before $t1$ is sent to $\text{VFW}_1$, and after $t1$ it is sent to $\text{VFW}_2$. Let $\Gamma$ be a set comprising the matching traffic that arrives during $(t0 - d1, t1)$. $\Gamma$ is shown as the blue part in Figure 3.4. Traffic in $\Gamma$ is sent to $\text{VFW}_1$.

At $\textbf{VFW}_1$, the firewall rules and flow states defined in *fspace* are sent to $\text{VFW}_2$. Traffic in $\Gamma$ starts to arrive after $t0$, and the last packet in $\Gamma$ arrives before $t1+d1$. $\text{VFW}_1$ processes all the traffic in $\Gamma$. $\text{VFW}_1$ finishes processing $\Gamma$ before $t1 + d1 + b1$. If any traffic in $\Gamma$ whose processing causes the state of a flow to evolve, $\text{VFW}_1$ will send the traffic to $\text{VFW}_2$ for reprocessing in order to keep state consistency. For generality, we assume all the traffic in $\Gamma$ are sent to $\text{VFW}_2$ for reprocessing. This assumption will lead to an estimate of buffer cost that is larger than the actual cost in some cases. Given that we want to avoid buffer overflow, over-estimates are preferable.

At $\textbf{VFW}_2$, traffic directly sent from $SW$ starts to arrive after $t1 + d2$. However, the traffic directly from $SW$ must be buffered until $\text{VFW}_2$ has received and reprocessed all the traffic in $\Gamma$. The last packet in $\Gamma$ arrives at $\text{VFW}_2$ before $t1 + d1 + b1 + d3$ and it is processed before $t1 + d1 + b1 + d3 + b2$. Therefore, traffic that is directly sent from $SW$ and arrives at $\text{VFW}_2$ during $(t1 + d2, t1 + d1 + b1 + d3 + b2)$ is buffered.

Suppose there are $k_i$ flows matching $V_i$ and the rate (bits per second) of flow $j$ is $\lambda_j$. Then we estimate the buffer cost of $V_i$ as

$$\begin{aligned} \beta_i &= (\sum_{j=1}^{k_i} \lambda_j) \times \{(t1 + d1 + b1 + d3 + b2) - (t1 + d2)\} \\ &= (\sum_{j=1}^{k_i} \lambda_j) \times (d1 + b1 + d3 + b2 - d2) \end{aligned} \tag{3.1}$$

## 3.4 Optimal Scaling

### 3.4.1 Virtual Firewall Modeling

The runtime throughput of a virtual firewall, $\eta$, is a function of $\Lambda$, the aggregated rate of the incoming traffic flows.

$$\eta(\Lambda) = \begin{cases} \Lambda & \text{if } \Lambda \leq c \\ c & \text{if } \Lambda > c \end{cases}$$

When $\Lambda > c$, packet loss may be expected at a virtual firewall, because the rate of traffic coming to the virtual firewall exceeds the rate of traffic leaving. Service Level Agreement (SLA) on performance generally requires the processing capacity of a virtual firewall to meet a predefined value $\Phi$, that is, $c(S) \geq \Phi$. Since $c$ decreases as $S$ increases, $\Phi$ essentially places an upper bound on $S$. The upper bound is $c^{-1}(\Phi)$. Therefore, $c(S) \geq \Phi$ is equivalent to $S \leq c^{-1}(\Phi)$.

### 3.4.2 Optiaml Scaling Out

After a virtual firewall is detected overloaded, `VFW Controller` runs *Dependency Analysis*, *Flow Update Analysis*, *Buffer Cost Calculation*, and has the following parameters:

- A set of $m$ firewall rule groups $\mathbb{V} = \{V_1, V_2, ..., V_m\}$ with group $V_i$ having a rule size of $s_i$, an update cost of $\gamma_i$, a buffer cost of $\beta_i$, and $k_i$ matching traffic flows with the flow rates of $\lambda_{i1}$, $\lambda_{i2}$, ..., $\lambda_{ik_i}$.
- A processing capacity of $c(\sum_{i=1}^{m} s_i)$, and a runtime throughput of $\eta$.
- An overload condition that $c(\sum_{i=1}^{m} s_i) < \Phi$ or $\eta > 0.9c(\sum_{i=1}^{m} s_i)$.

Then, `VFW Controller` adopts a three-step heuristic approach to work out the minimum number of new instances to create and the firewall rule group distribution.

**Step 1:** `VFW Controller` estimates the minimum number of new instances to create. Let $n$ be the number.

$$n = \begin{cases} \lfloor \frac{\sum_{i=1}^{m} \sum_{j=1}^{k_i} \lambda_{ij} - 0.9c(\sum_{i=1}^{m} s_i)}{\Phi} \rfloor & \text{if } \eta > 0.9c(\sum_{i=1}^{m} s_i) \\ \lfloor \frac{\sum_{i=1}^{m} s_i - c^{-1}(\Phi)}{c^{-1}(\Phi)} \rfloor & \text{if } S > c^{-1}(\Phi) \end{cases}$$

In the case when $\eta > 0.9c(\sum_{i=1}^{m} s_i)$, the amount of traffic that should be shunted away from the overloaded virtual firewall is $\sum_{i=1}^{m} \sum_{j=1}^{k_i} \lambda_{ij} - \eta$, which will be undertaken by the new instances, each of which must guarantee a processing capacity of $\Phi$ as dictated by the SLA. In the case when $S > c^{-1}(\Phi)$, the number of firewall rules that must be migrated is $\sum_{i=1}^{m} s_i - c^{-1}(\Phi)$, which will be split among the new instances, each of which is constrained by an upper bound $c^{-1}(\Phi)$. The floor operation in the estimate above implies that the estimate may be smaller than required.

**Step 2:** `VFW Controller` applies ILP to solve for firewall rule distribution. Let $\mathbf{x} = \{x_{11}, ..., x_{mn}\}$ be a set of unknown variables, where $x_{ij} \in \{0, 1\}$ is an indicator of migrating firewall rule group $V_i$ to new instance $j$. The ILP formulation to solve $\mathbf{x}$ is below:

$$\min \quad \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} \gamma_i$$

*s.t.*

(1) $x_{ij} \in \{0, 1\}$ for $1 \leq i \leq m, 1 \leq j \leq n$

(2) $\sum_{j=1}^{n} x_{ij} \leq 1$ for all $1 \leq i \leq m$

(3) $\sum_{i=1}^{m} x_{ij} \beta_i \leq B$ for all $1 \leq j \leq n$ // Prevent buffer overflow on each new instance.

(4) $\sum_{i=1}^{m} x_{ij} s_i \leq c^{-1}(\Phi)$ for all $1 \leq j \leq n$ // Each new instance must satisfy the SLA.

(5) $\sum_{i=1}^{m} x_{ij} (\sum_{l=1}^{k_i} \lambda_{il}) \leq 0.9c(\sum_{i=1}^{m} s_i x_{ij})$ for all $1 \leq j \leq n$ // Each new instance must not be overloaded.

(6) $(\sum_{i=1}^{m} \sum_{j=1}^{k_i} \lambda_{ij} - \sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{l=1}^{k_i} x_{ij} \lambda_{i,l}) \leq$ $0.9c(\sum_{i=1}^{m} s_i - \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} s_i)$ // After scale out, the old firewall is not overloaded. This constraint is used when overload condition (i) occurs.

or

(6') $(\sum_{i=1}^{m} s_i - \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} s_i) \leq c^{-1}(\Phi)$ //After scaling out, the old virtual firewall must satisfy the SLA. This constraint is used when overload condition (ii) occurs.

Solving the above ILP formulation, we obtain $\mathbf{x}$. If $x_{ij} = 1$ then firewall rule group $V_i$ is to be migrated to new instance $j$. If $x_{ij} = 0$ for all $j$, then firewall rule group $V_i$ stays on the old virtual firewall. If the above ILP formulation has no solution, which implies that the estimate of $n$ at Step 1 is smaller than required, then go to Step 3.

**Step 3:** Increase $n$ by one and perform Step 2 again until a solution is found.

### 3.4.3 Optimal Scaling-In

When more than one virtual firewall are underloaded, `VFW Controller` merges them. `VFW Controller` applies the following ILP formulation to determine the maximum number of virtual firewalls to be killed while meeting the following constraints:

- Each remaining virtual firewall must not be overloaded after the mergence.
- Each remaining virtual firewall must not have its buffer space overflowed.
- Each remaining virtual firewall must satisfy the SLA.
- The total update cost in the mergence is bounded by a predefined value $\boldsymbol{\gamma}$.

Suppose there are $n$ underloaded virtual firewalls. underloaded virtual firewall $j$ has:

- A set of $m^j$ firewall rule groups $\mathbb{V}^{\mathbb{j}} = \{V_1^j, V_2^j, ..., V_{m^j}^j\}$ with group $V_i^j$ having a rule size of $s_i^j$, an update cost of $\gamma_i^j$, a buffer cost of $\beta_i^j$, and $k_i^j$ matching traffic flows with the flow rates of $\lambda_{i1}^j$, $\lambda_{i2}^j$, ..., $\lambda_{ik_i^j}^j$.
- A processing capacity of $c(\sum_{i=1}^{m^j} s_i^j)$, and a runtime throughput of $\eta^j$.

Let $\mathbf{x} = \{x_{11}, ..., x_{nn}\}$ be a set of unknown variables, where $x_{qj} \in \{0, 1\}$ is an indicator of merging underloaded virtual firewall $j$ onto $q$. The ILP formulation to solve $\mathbf{x}$ is presented below:

$$\max \sum_{j=1}^{n} \sum_{q=1}^{n} x_{qj}$$

*s.t.*

(1) $x_{qj} \in \{0, 1\}$ for all $1 \leq q \leq n, 1 \leq j \leq n$

(2) $x_{qq} = 0$ for all $1 \leq q \leq n$

(3) $\sum_{q=1}^{n} x_{qj} \leq 1$ for all $1 \leq j \leq n$

(4) $\sum_{i=1}^{m^q} s_i^q + \sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} s_i^j \leq c^{-1}(\Phi)$ for all $1 \leq q \leq n$

(5) $\sum_{i=1}^{m^q} \sum_{l=1}^{k_i^q} \lambda_{il}^q + \sum_{j=1}^{n} \sum_{i=1}^{m^j} \sum_{l=1}^{k_i^j} \lambda_{il}^j x_{qj} \leq$
$0.9c(\sum_{i=1}^{m^q} s_i^q + \sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} s_i^j)$ for all $1 \leq q \leq n$

(6) $\sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} \beta_i^j \leq B$ for all $1 \leq q \leq n$

(7) $\sum_{q=1}^{n} \sum_{j=1}^{n} \sum_{i=1}^{m^j} x_{qj} \gamma_i^j \leq \boldsymbol{\gamma}$

Solving the above ILP formulation, we obtain all $x_{qj}$. If $x_{qj} = 1$, then virtual firewall $j$ is merged onto $q$. If $\sum_{q=1}^{n} x_{qj} = 0$, then virtual firewall $j$ remains. The maximum number of virtual firewall to be killed is $\sum_{j=1}^{n} \sum_{q=1}^{n} x_{qj}$.

## 3.5 Implementations

We have implemented a prototype of `VFW Controller` on top of ClickOS [105]. ClickOS is a Xen-based NFV platform optimized for fast provision of virtualized network functions in large scale. ClickOS creates small VMs (each less than 12MB) and boots one instance within 30 milliseconds. We have enhanced ClickOS to provide stateful virtual firewalls using Click modular router software [88]. Click provides rich networking processing elements, which can be leveraged to construct a wide range of virtual middleboxes. We have used three elements, *IPfilter*, *IPClassifier* and *IPFragmenter*, provided by Click to implement firewall packet processing function. We have also developed new Click elements for firewall rule management, buffer management, and interfaces for migrating rules and states. In particular, we have developed a programming interface on top of XL [13] [2], which `VFW Controller` uses to exert migration controls on individual virtual firewalls. Such a control automation is essential to fully take advantage of virtual firewall benefits. Our VFW implementation provides useful insights to implement and control other virtualized network functions as well.

Key functions of `VFW Controller` have been realized as individual modules. In particular, we have implemented a *Dependency Analysis* module based on Header Space Library (Hassel) [3], which is a tool for static header space analysis, a *Flow Update Analysis* module to find the correct flow updates and calculate update costs, a *Buffer Cost Analysis* module to calculate buffer costs, and an *Optimal Scaling Calculation* module that realizes the approaches for optimal scaling by calling a Matlab ILP solver. `VFW Controller` also includes a Floodlight module that implements Floodlight SDN controller functions [9]. `VFW Controller` uses Floodlight REST APIs to communicate flow updates to Floodlight module, which programs SDN switches through an OpenFlow interface. The same channel is also used by SDN switches to send network traffic statistics back to `VFW Controller`.

To improve the performance of `VFW Controller`, our implementation uses both online and prior processing. All the analyses, including dependency analysis, flow update analysis, update cost calculation, and buffer cost calculation are carried out prior. `VFW Controller` maintains a copy of firewall rules for each virtual firewall and the flow rules in its database for prior analyses. Results from the analyses are stored and retrieved whenever scaling is to be performed. Overload/underload detection, optimal scaling calculation, virtual firewall creation/deletion, migrations of firewall rules and flow states, and flow updates are carried out online.

[2] XL is a toolstack that provides the capability to provision guest VM in Xen.

## 3.6 Evaluation

We evaluate our virtual firewall with the following goals:

- Demonstration of the relationship between virtual firewall performance and the rule size (Figure 3.5). This justifies `VFW Controller`'s choice of rule *split* over rule *copy*.
- Study of the rule dependency relations in firewall policies (Table 3.1). This justifies the necessity of dependency analysis.
- Demonstration of virtual firewall's capability to quickly scale (Figure 3.6).
- Quantifying the impact of firewall rule migration on virtual firewall throughput (Figure 3.7).
- Evaluating the performance of optimal scaling calculation (Figure 3.8).

### 3.6.1 Experiment Setup

Our experiments were conducted using CloudLab [2], an open cloud platform providing various resources (server, storage, and network) for experimenters to run cloud software stacks such as OpenStack and CloudStack. In our experiments, we deployed a client machine that generated traffic, a server machine that received traffic, and a firewall machine created by `VFW Controller` to process the traffic between the client and the server. The client generated synthetic workloads using scapy [3], a powerful interactive packet manipulation program.

### 3.6.2 Performance of Virtual Firewalls

its firewall rule size $S$. We used three traffic datasets captured from real-world networks: 1) the CAIDA UCSD anonymized Internet trace [16] is a representative of Internet traffic; 2) the LBNL/ICSI enterprise trace [5] is a typical traffic collected from an enterprise network; and 3) the Campus network trace that was collected from our campus network gateway. Against each of the dataset, we conducted experiments to study how the firewall rule size affects the performance of a virtual firewall. In each experiment, we let $S$ increase from 1 to 3000, which we considered as a maximum number of rules in a single VFW. Then we measured the processing capability for each $S$ value and repeated each measurement 100 times to calculate the average processing capacity, as

[3] http://www.secdev.org/projects/scapy/.

shown in Figure 3.5(a). The average processing capacity linearly decreases as $S$ increases over all datasets. The LBNL enterprise traffic contains packets with greater length of payload, thus bares a more significant impact on performance. We applied polynomial curve fitting linear regress on the CAIDA trace, which was captured by April 2016 and represents the most up-to-date characteristic of today's Internet traffic, and obtained the function $c(S)$ as

$$c(S) = -0.0043S + 6.2785 \tag{3.2}$$

This function fits the corresponding CAIDA curve in Figure 3.5(a) with $R^2 = 0.9864$. $R^2$ is a measure of goodness of fit with a value of 1 denoting a perfect fit. $c(S)$ will be used by the evaluation of optimal scaling calculation.

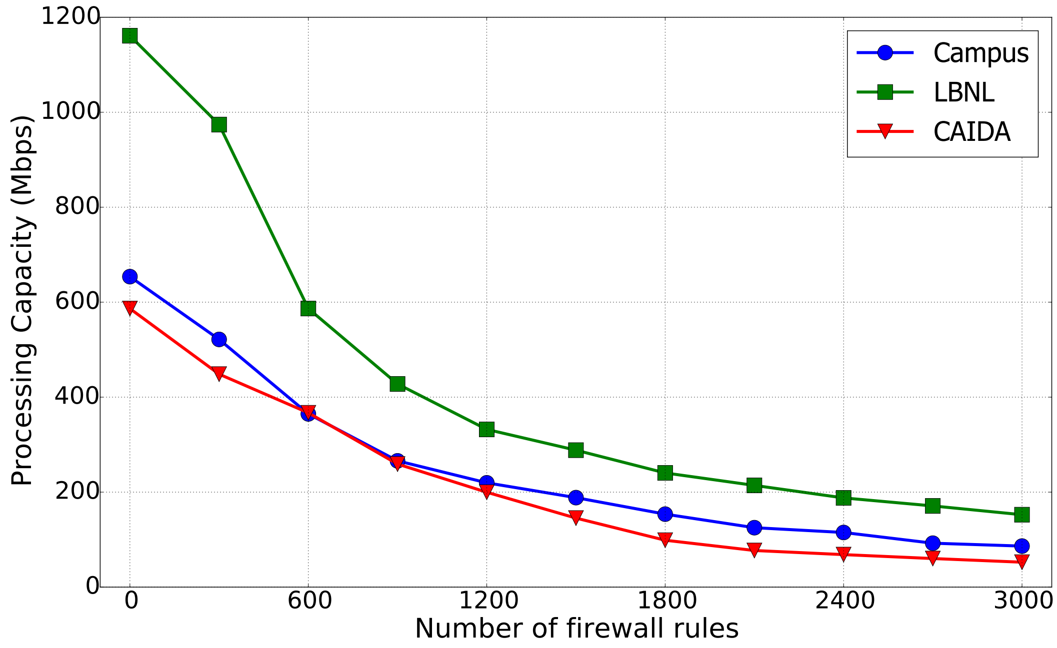
In the experiment, we also recorded the average time a virtual firewall instance spent to process a packet for each of the real-world dataset, as shown in Figure 3.5(b). As $S$ increases, the average processing time shows an linear increase, which eventually incurs the latency of the passed packets. These results of this experiment can be used for buffer cost calculation.

### 3.6.3 Rule Dependencies in Firewall Policies

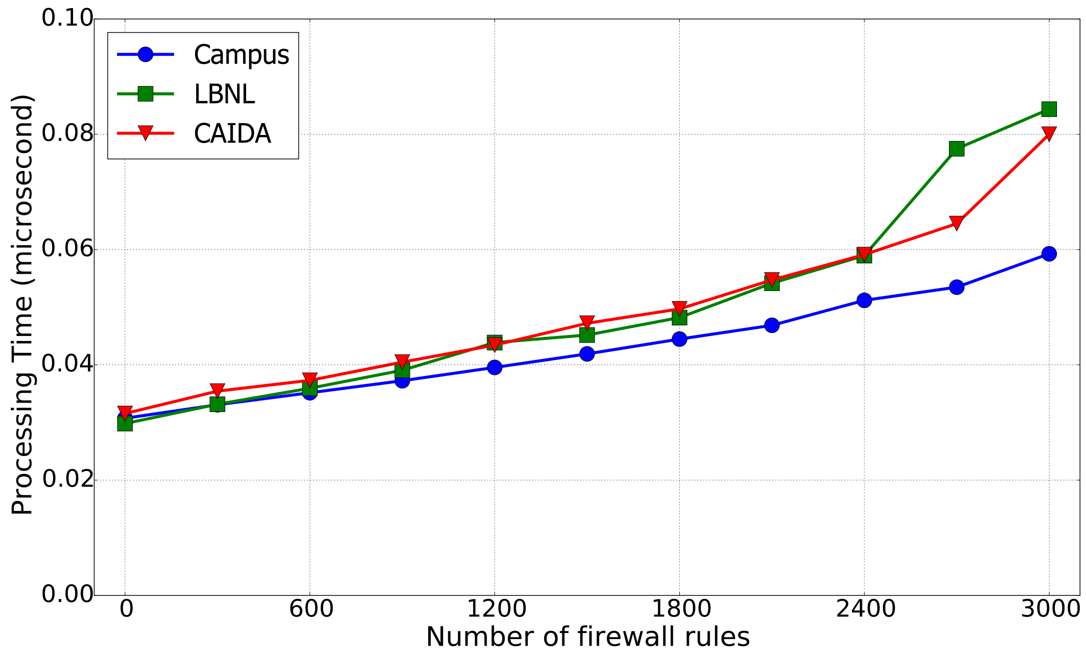
In this experiment, we studied 8 *real-world* firewall policies from different resources. Most of them are from campus networks and some are from major ISPs. We partitioned each policy to disjoint groups using the Algorithm 1. The experimental results are listed in Table 3.1. The first two columns of the table show the policies we used and their rule numbers. The third column gives the number of groups identified in each policy, and the fourth column shows the number of the firewall rules in the largest group of each policy. This experiment demonstrates that rule dependencies are common in *real-world* firewall policies. Therefore, the dependency analysis in `VFW Controller` is necessary. We also noticed from our study that the largest group we encountered contains only 18 firewall rules (from Policy H in Table 3.1).

### 3.6.4 Elasticity of Virtual Firewalls

In this experiment, we demonstrated `VFW Controller`'s ability to elastically scale out an overloaded virtual firewall. We designed three scenarios. In scenario (1), a single virtual firewall was created and configured with 400 firewall rules to process the traffic between the client and the server.




(a) Processing capacity.




(b) Processing time per packet.

Figure 3.5: Relationship between virtual firewall performance and rule size.

Table 3.1: Rule dependencies in real-world firewall policies.

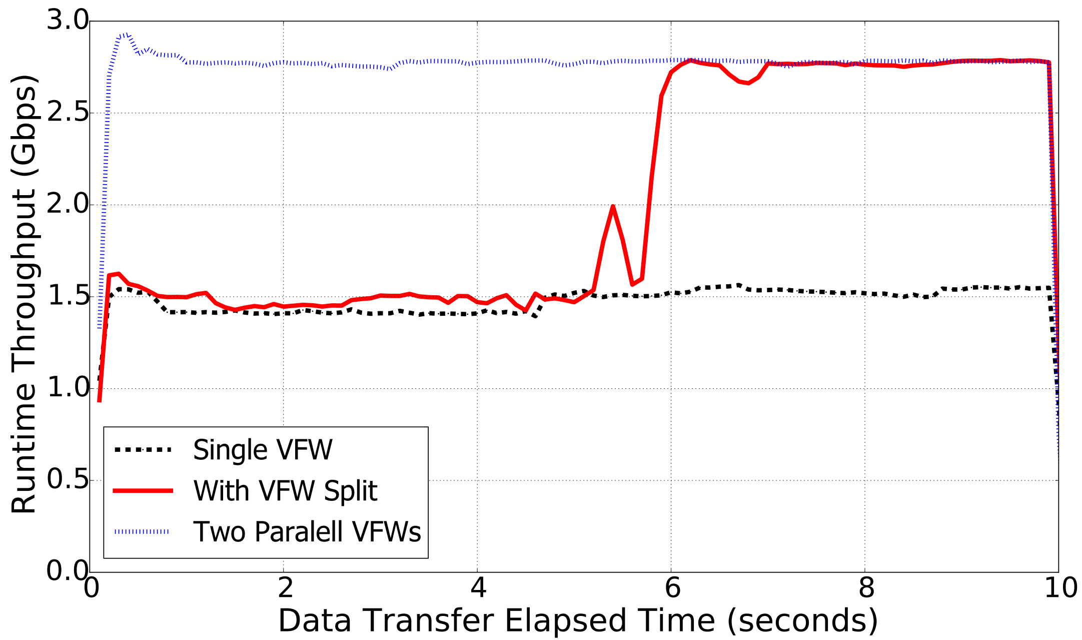
| **Policy** | **Rule** (#) | **Group** (#) | **Largest Group Member** (#) |
|---|---|---|---|
| A | 12 | 2 | 3 |
| B | 18 | 3 | 5 |
| C | 25 | 3 | 6 |
| D | 52 | 7 | 7 |
| E | 83 | 9 | 7 |
| F | 132 | 10 | 9 |
| G | 354 | 10 | 12 |
| H | 926 | 13 | 18 |

In scenario (2), two single virtual firewalls were created to work in parallel process the same traffic. The 400 firewall rules in scenario (1) were split and installed in those two virtual firewalls. In both scenarios, the virtual firewalls worked in standalone mode, which meant they were not connected to `VFW Controller` and no scaling would be performed. In scenario (3), a single virtual firewall with the same configuration as that in scenario (1) was created to work in standalone mode at first, then switched to connected mode, which meant the virtual firewall was connected to `VFW Controller` and was scaled out. We compared the runtime throughput of the virtual firewalls in the three scenarios as shown in Figure 3.6.

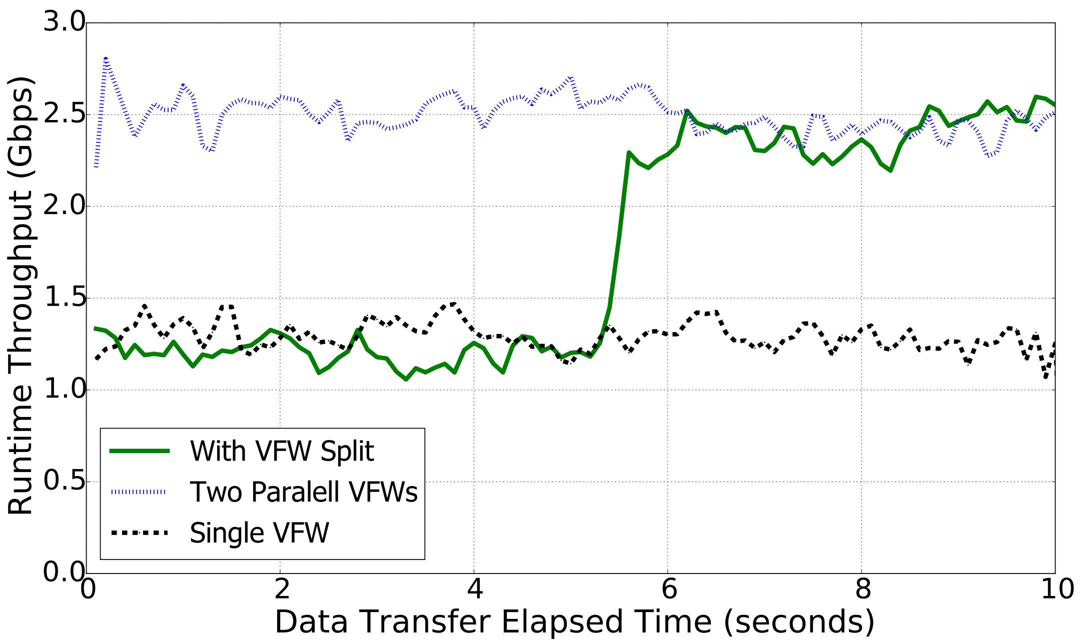
Figure 3.6(a) shows the runtime throughput of the three scenarios when the client generated 4 UDP flows with an aggregated traffic rate of about 2.8 Gbps. In scenario (1), the single virtual firewall achieved a maximum throughput (i.e. processing capacity) of about 1.5 Gbps. Significant packet loss was experienced. In scenario (2), the two virtual firewall were able to handle the incoming traffic, producing a runtime throughput of about 2.8 Gbps. No packet loss occurred. In scenario (3), we intentionally let a single virtual firewall to work from time $t = 0s$ to $t = 5s$. Packet loss occurred during this time period. Then we connected this virtual firewall to `VFW Controller` and it was scaled into two virtual firewall instances. The scaling-out took a short period of time ($< 1$ second), after which we observed the aggregated runtime throughput increased to around 2.8 Gbps.

We also evaluated `VFW Controller` against TCP traffic. Figure 3.6(b) shows the runtime throughput of the three scenarios when the client established 2 TCP connections with the server. We observed a boost of runtime throughput between $t = 5s$ and $t = 6s$.

In summary, the above results demonstrated that `VFW Controller` can quickly scaled out




(a) Split with UDP flow overload.




(b) Split with TCP flow overload.

Figure 3.6: `VFW Controller` for VFW elasticity.

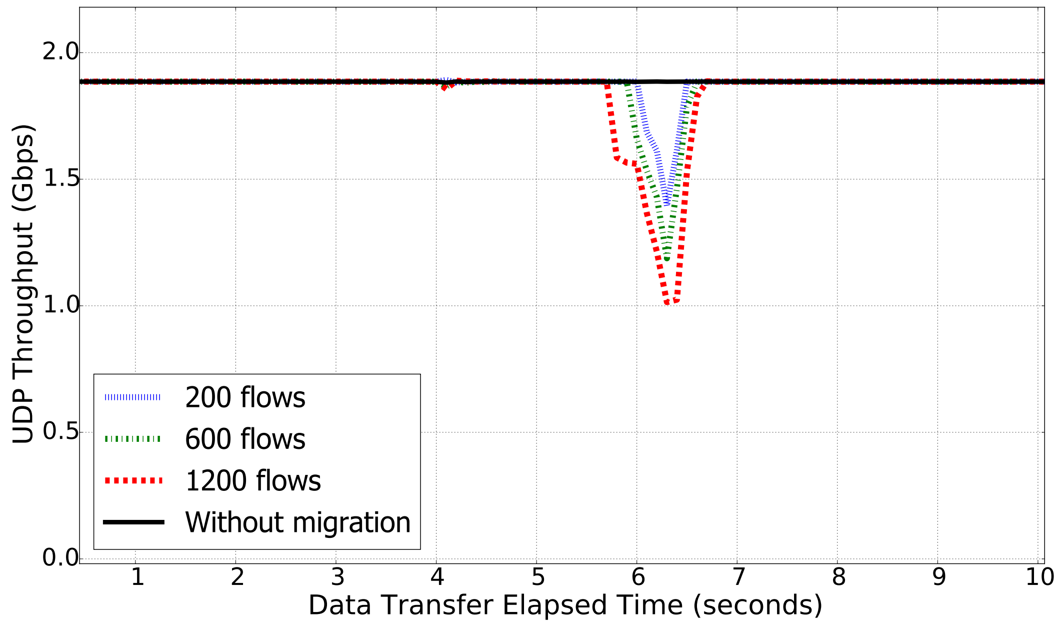
an overloaded virtual firewall and solved the overload condition.

### 3.6.5 Impact of Migration

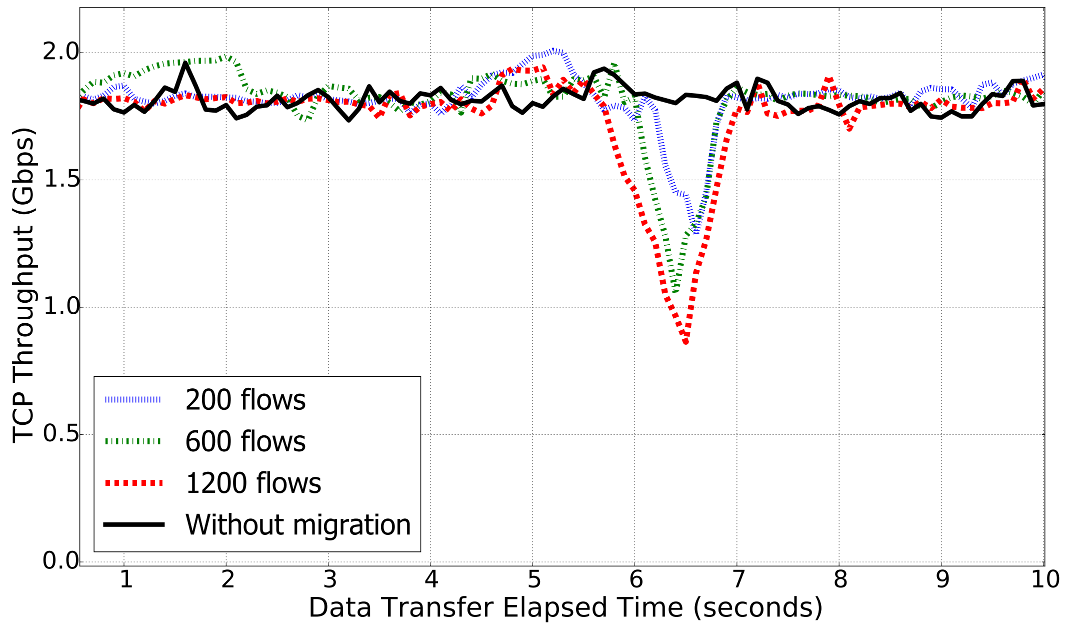
During the migration, the in-flight traffic are buffered until the migration completes, therefore the runtime throughput of a virtual firewall is expected to degrade during the migration. According to equation (3.1), if more traffic flows are associated with the migrated firewall rules, more in-flight traffic will be buffered. Therefore larger the degradation will be and longer the degradation will last. In this experiment, we quantified both the duration and magnitude of throughput degradation. We set up two scenarios to test UDP and TCP flows, respectively.

In scenario (1), the client kept sending UDP traffic destined to the server. The UDP packets were routed through a virtual firewall created by `VFW Controller`. In Figure 3.7 (a), the solid black line, as a base line, shows the throughput of the virtual firewall when no migration occurred. The dotted lines in Figure 3.7(a) show the runtime throughput of the virtual firewall when migrating 200, 600, 1200 firewall rules, respectively. The migrations were scheduled at $t = 6s$. We fabricated firewall rules so that each firewall rule has one matching flow. The duration and magnitude of throughput degradation increased as the number of migrated firewall rules (or the number of UDP flows) increased. However, the degradation lasted for a very short period of time (≈0.75s with 1200 flows) and the throughput bounced back very quickly. In scenario (2), the client sent TCP traffic. Figure 3.7(b) shows that the degradation of throughput of the virtual firewall when processing TCP traffic. As the number of firewall rules (or TCP flows) increases, the degradation lasted longer and was bigger. The degradation in the case of TCP traffic lasted slightly longer than that of UDP, because TCP depends on congestion avoidance mechanisms to control its traffic rate, which means TCP flows take more time to recover from a throughput degradation than UDP flows. For both UDP and TCP flows, the throughput began to bounce back in less than 0.1 second after it reaching the lowest point (see Figures 3.7(a) and (b)).

To sum up, the duration and magnitude of throughput degradation increase as the number of migrated firewall rules (or the number of flows) increase. And the degradation shows an bigger impact on TCP flows than UDP flows.




(a) Impact on UDP throughput.




(b) Impact on TCP throughput.

Figure 3.7: Performance overhead of migration.

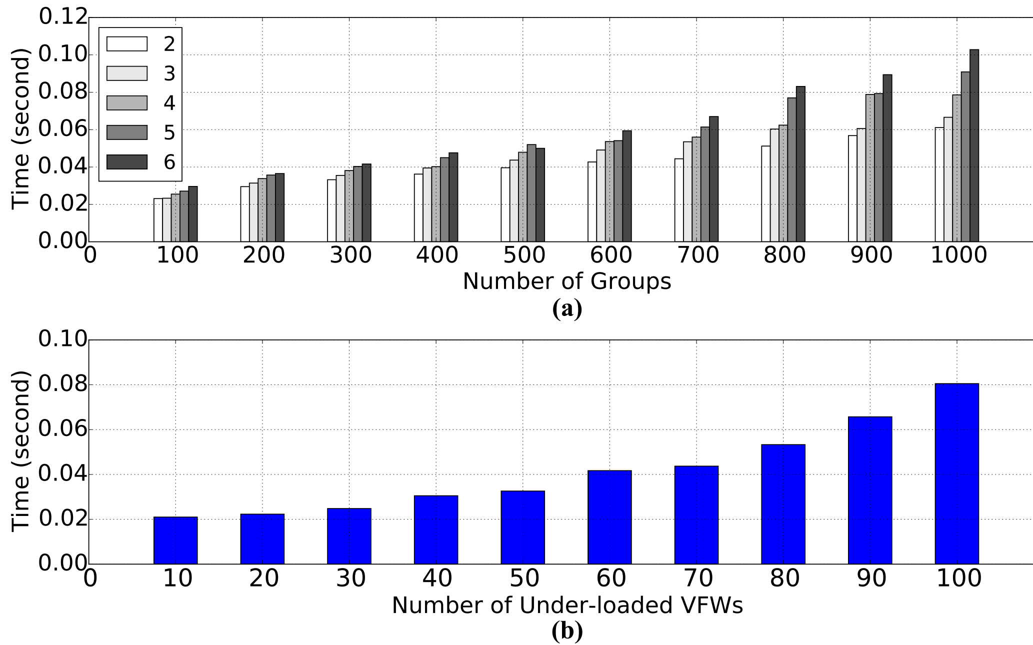



Figure 3.8: Performance of provision plan calculation.

### 3.6.6 Performance of Optimal Scaling Calculation

We introduced a three-step heuristic approach in `VFW Controller` to calculate an optimal solution for scaling out an overloaded virtual firewall. The performance of this approach depends almost fully on the ILP formulation and solving used in the second step. Therefore, we mainly tested the performance of our ILP formulation and solving. The ILP performance is influenced by (i) $m$, the number of firewall rule groups on the overloaded virtual firewall; and (ii) $n$, the number of new virtual firewall instances to be created. In our experiment, we measured the time to find the optimal solution for varied $m$ and $n$ values. We let $n$ range from 2 to 6, and $m$ range from 100 up to 1000. Figure 3.8(a) shows our experiment results. The time to find an optimal solution increases as $m$ or $n$ increases, however, the time is very short. Even for $m = 1000$, our approach needs less than 0.11 second.

Figure 3.8(b) depicts the time that our approach consumed to find an optimal solution for scaling in when the number of underloaded virtual firewall instances is changed. Our approach is very efficient. Even when 100 virtual firewall instances are detected to be underloaded, our approach takes less than 0.08 second to find an optimal scaling-in solution.

## 3.7 Conclusion

In this research task, we implement firewall as software instances that can be create and destroy quickly. To ensure the rapid live migration of virtual firewall instances, we design `VFW Controller`, a virtual firewall controller, which enables *safe*, *efficient* and *optimal* virtual firewall scaling. To demonstrate the feasibility of our approach, we have implemented the core components of `VFW Controller` on top of ClickOS. Our evaluation shows that the virtual firewall we developed in this task can safely and efficiently scale.

# Chapter 4

# Safe and Efficient NIDS Virtualization [1]

## 4.1 Introduction

Traditional NIDSes are generally implemented on vendor proprietary appliances or *middleboxes* with poor versatility and flexibility. The Emerging SDN and NFV technologies give the potential to virtualize NIDSes and elastically scale them to deal with attack traffic variations. However, such an elasticity feature must not come at the cost of decreased detection effectiveness and expensive provisioning. In this research task, we propose an NIDS architecture, vNIDS, to enable safe and efficient virtualization of NIDSes. vNIDS addresses two key challenges with respect to *effective intrusion detection* and *non-monolithic NIDS provisioning* in virtualizing NIDSes. The former challenge is addressed by detection state sharing while minimizing the sharing overhead in virtualized environments. In particular, *static program analysis* is employed to determine which detection states need to be shared. vNIDS addresses the latter challenge by provisioning virtual NIDSes as microservices and employing *program slicing* to partition the detection logic programs so that they can be executed by each microservice separately. Besides, we implement a prototype of vNIDS and design experiments to demonstrate the feasibility of our approach.

[1] This work has been published in Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS 2018).

## 4.2 Research Challenges

The two research challenges that must be addressed to achieve safe and efficient NIDS virtualization are as follows.

- **Effective Intrusion Detection:** NFV enables flexibly provisioning an NIDS as multiple instances that could run at different locations, and SDN can dynamically distribute network traffic to each instance. Some detection logics in NIDSes must take account of multiple network flows to identify malicious activities. For example, a scanner detector [28] must maintain a counter to count the number of flows generated by the same host to determine whether a host is a scanner. However, each NIDS instance may receive only a part of flows originated from the same scanner and maintains a counter individually. To ensure intrusion detection effectiveness, NIDS instances must share detection states with each other. Traditional NIDSes are usually provisioned with a fixed number of instances at a fixed location. This static setup mitigates the difficulty of detection state sharing between NIDS instances. In contrast, virtualized NIDSes pursue to place their instances flexibly at different locations and with different numbers. This flexibility feature makes sharing detection states among virtualized NIDS instances extremely costly and difficult. Therefore, ensuring the effectiveness of intrusion detection for virtualized NIDSes becomes more challenging than traditional NIDSes.

- **Non-monolithic NIDS Provisioning:** Existing virtualization solutions enabling elastic security [165] consider a virtualized network function as a *monolithic* piece of software running in a virtual machine or container. However, simply provisioning all the components of virtualized NIDSes within a virtual machine or container is not efficient. First, since a monolithically virtualized NIDS requires more resources than any one of its components, the virtual machine that runs a monolithically virtualized NIDS should be allocated with more resources than the one that runs only some components. Second, monolithically virtualized NIDSes lack the ability to scale each component individually, thus have to scale out the entire instance even though only one component is overloaded, resulting in over-provisioning of other components. Third, it is difficult to customize a virtualized NIDS if it is provisioned as monolith. However, NIDS customization is critical in terms of resource efficiency for advanced network attack defense [165, 167, 59]. However, due to the complexity of modern detection logics, decoupling

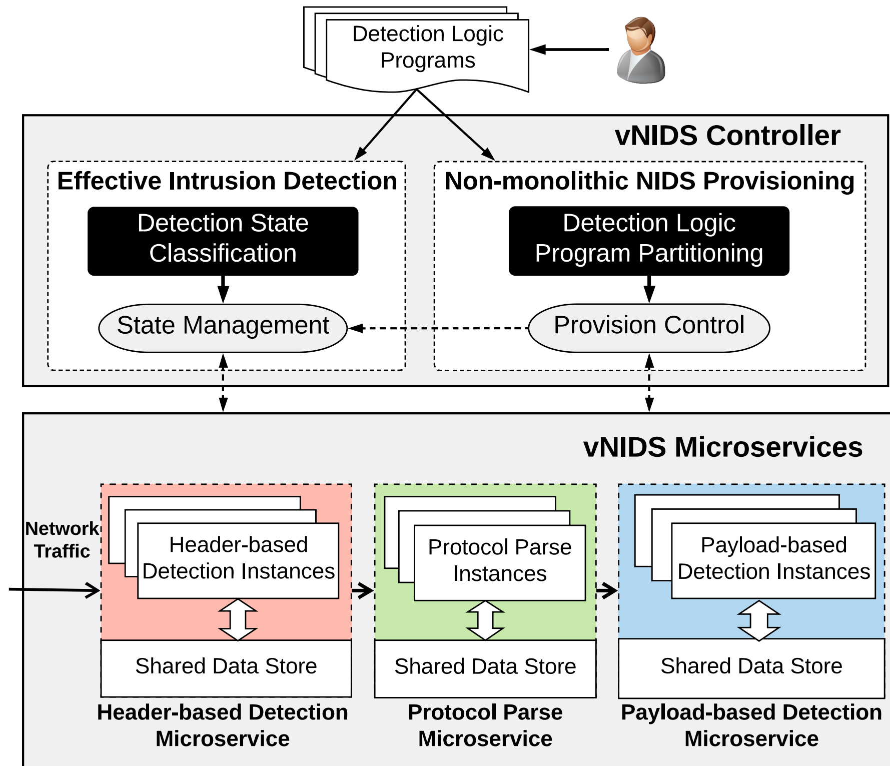



Figure 4.1: High-level view of vNIDS architecture.

NIDSes as non-monolithic pieces that are suitable to be deployed in virtualized environments remains challenging.

## 4.3 Methodology

### 4.3.1 High-level View of vNIDS

Our design of vNIDS is demonstrated in Figure 4.1. vNIDS consists of vNIDS *controller* and vNIDS *microservice instances*. The users interact with vNIDS through the controller. The microservice instances are responsible for processing the traffic.

**vNIDS controller:** There are two major modules in the vNIDS controller: the Effective Intrusion Detection module (EIDM) and Non-monolithic NIDS Provisioning module (NNPM). EIDM implements a detection state classification engine, which is run off-line to classify the detection states of DLPs into *global* and *local* detection states. The detection state taxonomy is sent to the State Management component (SMC). The SMC interacts with the shared data store and is responsible

for managing the detection states of instances. If the virtualized NIDSes scale-in/-out, the SMC should also address the state migration. NNPM implements the DLP partitioning algorithm, which is run off-line to partition a DLP into header-based and payload-based DLPs. The Provision Control component (PCC) in the NNPM takes charge of microservice instance provisioning, installs the DLPs into the instances, and demands the SDN controller to update the flow rules in network forwarding devices accordingly. Each time the PCC creates or destroys instances, it should inform the SMC to update the shared data store.

**vNIDS microservice instances:** The microservice instances in vNIDS are instantiated from the three microservices. Instances belonging to the same microservice share their *global* states through the shared data store. Instances belonging to different microservices can be chained up as "microservice chains". Each instance in a "microservice chain" only executes a part of the detection logic of the chain. This design enables more fine-grained customization of virtualized NIDSes by composing the "microservice chains" with different microservices. The results of a microservice can be attached to the traffic and passed to the next microservice.

Provisioning virtualized NIDSes with microservices is more efficient from the following perspectives. First, microservices are smaller than monolithic NIDSes and can be provisioned separately. Therefore, microservices can scale faster and independently. Second, since microservices require less resources to instantiate, they can make better use of fractional resources than monolithic NIDSes. Third, a microservice can be modified and updated without interfering other microservices. In addition, different microservices can be instantiated at different locations (for resource optimization [38]) and provisioned on demand based on different security contexts [165].

### 4.3.2 Effective Intrusion Detection

#### 4.3.2.1 Detection State Scope Analysis

Based on how a detection state is used, the scope of the detection state stored by a variable can be defined in three levels.

- *Per-packet* scope. We call detection states with this scope *per-packet* detection states. The *per-packet* detection states are only utilized within a single iteration of packet processing, thus are created and destroyed within a single iteration of packet processing. For example, a variable used to compute the checksum of a packet is considered as a *per-packet* detection state.

- *Per-flow* scope. We call detection states with this scope *per-flow* detection states. The *per-flow* detection states are utilized by multiple packets from the same flow. Therefore, these states persist beyond a single iteration of packet processing. However, these states must be created after a flow initiates and destroyed before a flow terminates. Otherwise there are memory leaks [2], since other flows can never access theses detection states. For example, a variable used to count the bytes of a flow is considered as a *per-flow* detection state.

- *Multi-flow* scope. We call detection states with this scope *multi-flow* detection states. The lifetime of *multi-flow* detection states persists beyond the duration of a single flow. Therefore, they are created before a flow initiates or destroyed after a flow terminates. For example, a variable used to count how many IP addresses have been scanned by a scanner is considered as a *multi-flow* detection state.

#### 4.3.2.2 Detection State Classification

We explain in detail the major four steps of our algorithm that infers the scope of detection states as follows.

*Step1: computing the CFG of the DLP and NE.* Every time the NIDS receives a packet, it starts from a packet receive function call (e.g, *pcap_next* or *recv*) and ends at the destroy statement of the packet. We therefore consider a packet receive function call as the entry statement for the CFG.

*Step2: identifying the packet and flow record destroy statements.* We manually check the code of IP reassembly implementation, which is available as standard libraries in many operating systems. For any specific NE, once identified, this information can be reused by all DLPs.

*Step3: computing dominators of the packet and flow record destroy statements.* Computing dominators of a given node in a CFG is a mature technique [91] in the graph theory domain.

*Step4: determining the scope of each detection state.* There are three cases of the destroy statement of a variable. (1) If the destroy statement of a variable is a dominator of the packet destroy statement, this variable is a *per-packet* detection state. (2) If the destroy statement of a variable is a dominator of the flow record destroy statement, this variable is a *per-flow* detection state. (3) If the destroy statement of a variable is neither a dominator of the packet destroy statement nor a dominator of the flow record destroy statement, this variable is a *multi-flow* detection state.

[2] An allocated memory location that will never be freed causes a memory leak.

After we infer the scope of each detection state by analyzing the CFG of the DLP and NE, we define the *per-packet* and *per-flow* detection states as *local* detection states and define *multi-flow* detection states as *global* detection states. We justify this definition in the next section.

#### 4.3.2.3 Ensuring Effective Intrusion Detection

To ensure effective intrusion detection, we must ensure that all the traffic under process can access to all the detection states relevant to the traffic. We achieve this goal by: 1) distributing the traffic of the same flow to the same instance, and 2) enabling *only global* detection states shared by multiple instances.

Since advanced network forwarding devices now have programmability thanks to SDN, dynamically redistributing traffic is feasible. When virtualized NIDSes dynamically scale (by creating or destroying instances), network traffic can be redistribute among the instances at *per-flow* granularity. That is, leveraging SDN, we can dynamically update the forwarding rules in the network forwarding device to always deliver the traffic of the same flow to the same instance.

Actually, we are trying to make a trade-off between the complexity of distribution algorithm (i.e., how to distribute the traffic to ensure detection effectiveness) and the overhead of detection state sharing (i.e., enabling information exchange among instances). On the one hand, we utilize the flexible *per-flow* forwarding capability of SDN to simplify the traffic distribution for virtualized NIDSes. On the other hand, the intrusion detection effectiveness is guaranteed by sharing *only multi-flow* detection states.

We can enable detection state sharing among instances by maintaining *global* detection states in the shared data store, such as RAMCloud [121], FaRM [50] and Algo-logic [17]. The authors of StatelessNF [82] have shown the feasibility of using RAMCloud as the shared data store for network functions. By classifying the detection states into *local* and *global* detection states in our approach, we only put *global* detection states on the shared data store to minimize the performance overhead.

### 4.3.3 Non-monolithic NIDS Provisioning

In this section, we first present how to identify the two partition points to decompose virtualized NIDSes into three *microservices*. Then we design an algorithm to partition the DLPs such that they can be enforced by non-monolithic virtualized NIDSes.

#### 4.3.3.1 Decomposing NIDSes as Microservices

**Logic structure of NIDSes:** Typically, the logic structure of NIDSes consists of three major components: the network protocol stack (NPS), application protocol parsers (APP), and the detection logic (DL). NIDSes firstly acquire and validate the network traffic through the NPS. This component is responsible for reading the packet from the driver, checking the checksum, reassembling the IP fragments, etc. It is implemented in the NE. Then, the outputs of the NPS are passed to the APPs. The APPs parse the payload of the traffic and extract the semantics of the conversation between two end points of the network. The outputs of the APPs are then processed based on various DLs.

For the purpose of reducing the development cost while maximizing the extensibility, the NPS and APPs are usually reused by different DLs. NIDS users are allowed to add their new APPs, while NPS is not likely to be changed by the users. For example, the Bro network security monitor [27] has integrated the IP/TCP network protocol stack and a number of predefined application protocol parsers[3]. Then, users only need to program their detection logic using a scripting language [24] provided by Bro. Users can also add their own APPs through specific program languages such as BinPAC [123]. Another example is Libnids [23], which has implemented a library for developing the NIDSes. The IP/TCP NPS has already been included in the library, but the users need to implement their own APPs and DLs through the C interface stubs. Therefore, for generality, we consider APPs as a part of NE and only DL is implemented by the DLPs.

**Types of DL:** We observed that there are three types of DLs based on what information the DL needs to conduct the detection task.

- `Type-I`: These DLs only need header information to complete their detection tasks. For example, the *Flow Monitor* implements the DL that only monitors the statistics of the flows, such as byte-per-flow and packet-per-flow statistics. Another example is the DL of *HTTP Monitor*, which first filters out the HTTP traffic by the port numbers and then compute the statistics of traffic generated by each Web server.

- `Type-II`: These DLs only need payload information to complete their detection tasks. For example, all the signature-matching DLs only need to check the payload of the packets.

- `Type-III`: Theses DLs need both header and payload information to complete their detection

[3] In the Bro community they are called as *protocol analyzers*.

| Cases | Microservice Chains | Detection Logics |
|---|---|---|
| **Case 1** | Header-based Detection → Protocol Parse → Payload-based Detection | **DNS Tunneling** and **Trojan** DLs |
| **Case 2** | Header-based Detection → Payload-based Detection | **Scanner** DL |
| **Case 3** | Protocol Parse → Payload-based Detection | **Cookie Hijacking** DL |
| **Case 4** | Header-based Detection | **Flow Monitor and HTTP Monitor** DLs |

Figure 4.2: Different DLs can be implemented by different microservice chains.

tasks. For example, the Cookie Hijacking DLP first looks at the cookie, which is payload information. Then it checks the IP address, which is a header information.

**Decomposing monolithic NIDSes:** Based on the logic structure of NIDSes and the classification of DLPs, we decompose NIDSes as three microservices: *header-based detection*, *protocol parse* and *payload-based detection*. The logic structure of NIDSes indicates three major processing of the packets. The first processing is executed by the NPS, which is the basic component of all network functions. We reuse this processing in all the microservices. The second processing is executed by APPs, which parse the traffic according to various application protocol specifications. We define the this processing as a dedicated microservice called *protocol parse*, which is devoted to parsing the traffic according to various application protocol specifications. The third processing, which is specified by users, implement various DLs. Based on the types of DLs, we can actually classify all the DLPs into header-based DLPs, which implement `Type-I` DLs, and payload-based DLPs, which implement both `Type-II` and `Type-III` DLs. We define the *header-based detection* microservice devoted to executing the header-based DLPs. an the *payload-based detection* microservice devoted to executing the payload-based DLPs.

#### 4.3.3.2 Microservice chaining

Instances instantiated from different microservices can be chained as “microservice chains”. A microservice chain implements a complete detection task, for example, detecting whether a specific

host is a scanner, or whether a cookie has been reused. Each instance in the chain takes over a part of the detection task. Intermediate results generated by the previous instance can be encapsulated in the traffic (using some encapsulation techniques such as FlowTags [60], NSH [127], Geneve [67], etc.) and passed to the next instance in the chain.

We have studied six different types of DLs, which can be summarized into 4 cases when composed as microservice chains. Their DLPs are listed in Table 4.1. Different DLs can be implemented with different microservice chains. Figure 4.2 depicts that the six DLs can be summarized into four cases. Each case can be implemented by one microservice chain. For example, the Cookie Hijacking DL (case 3) can be implemented with *protocol parse* and *payload-based detection* microservices, while Scanner DL (case 2) can be implemented with *header-based detection* and *payload-based* detection microservices.

Though microservice chains can be composed flexibly, there still some constrains of how the microservice chains can be composed. First, the *header-based detection* microservice can be located before the *protocol parse* and *payload-based detection* microservices, because *header-based* microservice does not need payload information. Second, the *protocol parse* microservice should be located before the *payload-based detection* microservice, since the latter needs the payload information gathered by parsing the traffic.

#### 4.3.3.3 Detection Logic Program Partitioning

As we have outlined, NIDSes can be decomposed as three microservices. However, it remains the challenge to partition the user defined DLPs into header-based and payload-based DLPs that can be executed separately by the *header-based detection* and *payload-based detection* microservices. In this section, we present an algorithm that can automatically partition the user-defined DLPs by utilizing the program slicing technology [74, 131, 158].

Program slicing is a program analysis technique that can determine: *1)* which statements are influenced by a given variable at a given point (forward slicing); and *2)* which statements can influence the value of a variable at a given point (backward slicing). The inputs of the program slicing are the program dependency graph (PDG), variables of interest, and the point where the variable is considered. The outputs of the program slicing are the statements of interest. PDG is a directed graph whose vertices represent statements and edges represent dependencies between statements. There are some program slicing tools available. In this work, we implement our DLPs in

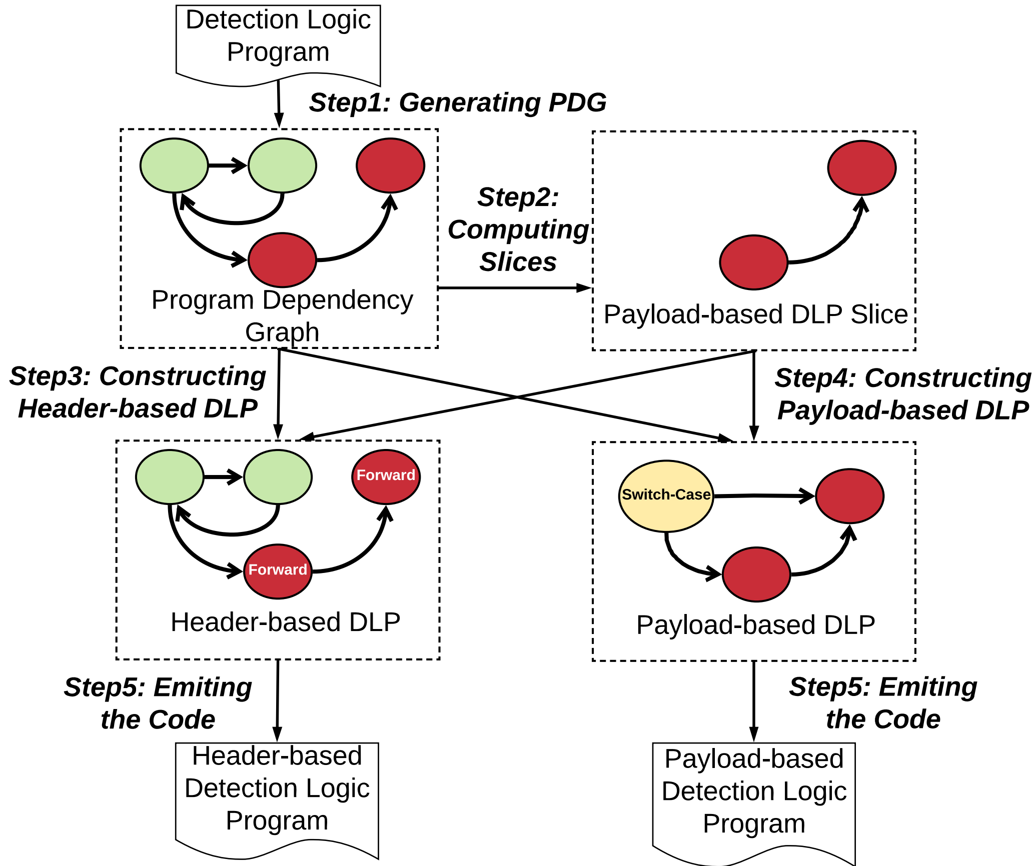



Figure 4.3: Major steps of the DLP partitioning. (1) Generating PDG of the DLP; (2) Computing payload-based DLP slice; (3) Constructing header-based DLP; (4) Constructing payload-based DLP; (5) Emit the code.

C and utilize Frama-C [20] to conduct program slicing on the DLPs. We achieve DLP partitioning through five major steps as shown in Figure 4.3.

*Step1: Generating PDG.* We conduct static program analysis to generate the program dependency graph of the inputed DLP. The PDG of a program involves two dependency information between the statements of a program–the control dependency and the data dependency. The control dependency, usually represented by control flow graphs, reflects the execution conditions and sequences between statements. The data dependency reflects the declaration and reference relationship between variables. There are tools that can be used to generate the PDG of a program. For example, Frama-C [20] can generate the PDG of C program and a commercial product CodeSurfer [19] can generate the PDG of C++ program.

*Step2: Computing slices.* After we get the PDG of the DLP, we utilize program slicing

tools to isolate the statements of payload-based DL from other statements. The key insight is that payload-based DL relies on payload information thus is influenced by inputs containing payload information. If an input is a direct reference of some header fields, this input only provides header information, otherwise it also contains payload information. Based on the above insight, we compute a forward program slice from the input of the DLP for the inputs that contain payload information. we call the resulting statements as the payload-based DLP slice. All the statements in the payload-based DLP slice are affected by the payload information. Therefore, we should let the *payload-based detection* microservice execute these statements.

*Step3: Constructing header-based DLP.* Since we have isolated the statements that are influenced by payload information, the remaining statements are independent to the payload information. These remaining statements can be executed by the *header-based detection* microservice. Algorithm 3 depicts the algorithm to identify the statements of the header-based DLP. Lines 1-4 identify the inputs containing payload information. Lines 5-7 compute the payload-based DLP slice. Line 8 computes the statements of the header-based DLP.

Then, one question remains: how can we address the packet if the processing of the packet reaches a point where the statements in the payload-based DLP slice must be executed? Note that those statements in the payload-based DLP slice cannot be executed in the *header-based detection* microservice, since payload information is not available in that microservice. We solve the problem by introducing the `forward` statement. We replace all the statements in the payload-based DLP slice with the `forward` statement. The `forward` statement is actually an interface to the NE, similar to the "system call" in the operating systems. Once the packet processing reaches the `forward` statement, it causes a trap into the NE. Then, the NE handlers the packet by encapsulating it with some metadata and sends it to the network. The metadata includes two pieces of information: *1)* the points where the execution is interrupted; and *2)* the intermediate results associated with the processing of this packet. We can get information *1)* by labeling the `forward` statement (e.g., using its line number). We can get information *2)* by tracking which variables have been updated by the current packet processing. Note that this information should not be too much, since we only need to track the variables that are also used by the payload-based DLP. Lines 9-12 of Algorithm 3 computes which variables should be tracked.

*Step4: Constructing payload-based DLP.* As we have already computed the payload-based DLP slice. We can use that slice as a starting point to construct the payload-based DLP. There

**Algorithm 3:** Identifying statements of header-based DLP and variables to be tracked by header-based DLP

**Input:** $F$: a set of header fields;
$prog$: origin detection program
**Output:** header-based DLP statements and variables to be tracked

```
1  PI = {} /* a set storing inputs containing payload information */
2  foreach input in Inputs(prog) do
3      if input ∉ F then
4          PI = PI ∪ {input}
5  Slice = {} /* a set storing statements in resulted slices */
6  foreach input in PI do
7      Slice = Slice ∪ ForwardSlicing(prog, entry, input)
8  S = Statements(prog) − Slice
9  V = {} /* a set storing variables that should be tracked by header-based
       DLP. */
10 foreach v in Variables(Slice) do
11     if v ∈ Variable(S) then
12         V = V ∪ {v}
13 return S and V
```

are two major concerns with the construction: *1)* how can we get access to the intermediate results produced by the header-based DLP? and *2)* how can we retrieve the execution from the previous break points? For concern *1)*, we add an initialization statement at the beginning of the payload-based DLP slice for every variable that are also used by the header-based DLP. Recall that we have a contract that the `forward` statement in the header-based DLP will encapsulate intermediate results of those variables used by both DLPs. As a result, those variables are initialized at the beginning of the payload-based DLP slice. For concern *2)*, we add a `switch-case` statement right after all the initialization statements. The `switch-case` statement checks the label that is set by the `forward` statement in the header-based DLP. According to the label, the `switch-case` statement jumps to the right points to start execution.

*Step5: Emitting the code.* This step is a reverse of *Step1*. Once we have constructed the PDG of header-based and payload-based DLPs, there is little challenge to emit the code for both DLPs. Many program analysis tools also support the reversing procedure to emit the code from a computed PDG.

### 4.3.4 Implementations

We have implemented a prototype of vNIDS based on Xen [13] hypervisor. The vNIDS controller is implemented in the *Dom0* of Xen, which has the ability to monitor and manage the virtual machines provisioned in Xen. We utilized Floodlight [9], an industry-scale SDN controller, and Open vSwitch [25] to construct our SDN environment. In particular, the vNIDS controller interacts with Floodlight to achieve per-flow traffic distribution. The three microservices are implemented based on the Click modular router software [88]. Click provides rich networking processing elements, which can be leveraged to construct a wide range of network functions.

**vNIDS controller:** We have implemented the DLP partitioning algorithms and detection state classification engine based on Frama-C [20], a static program analysis tool framework for C. The DLP partitioning algorithms and detection state classification engine are run off-line. Each time the user specifies new DLPs, the DLP partitioning algorithms will be run to partition the DLP into two header-based and payload-based DLPs that can be installed into the *header-based detection* and *payload-based detection* microservices, respectively. The detection state classification engine takes the two small DLPs and outputs DLPs with classified detection sates. We have developed a programing interface on top of XL to support provision control and state management.

**vNIDS microservice instances:** We have developed three new elements for Click to enable the three microservices. The major functionalities of the three microservice include *i)* handling the messages from the controller; *ii)* integrating RAMCloud [134] to support detection state sharing between instances; and *iii)* executing DLPs or protocol parsers (we have implemented HTTP, SSH, DNS, and FTP protocol parsers for our virtualized NIDS). We have written six DLPs as Click elements for various detection purposes. Table 4.1 shows the descriptions of the six DLPs that we have written for our virtualized NIDS. In particular, *DNS Tunneling DLP*, *Cookie Hijacking DLP*, *Trojan DLP* and *Scanner DLP* are representatives of the DLPs handling traffic of different application protocols. *Flow Monitor DLP* and *HTTP Monitor DLP* are commonly used traffic monitors that collect traffic statistics and conduct some detection tasks based on the statistics.

## 4.4 Evaluation

We evaluate our virtual NIDS with the following major goals:

| Detection Programs | Descriptions |
|---|---|
| DNS Tunneling DLP [18] | Detects potential DNS tunnels based on the DNS request frequency, packet length and domain name characteristics. |
| Cookie Hijacking DLP [15] | Detects whether an HTTP cookie has been reused by different users. |
| Trojan DLP [46] | Combines SSH, HTTP and FTP traffic to detect whether a host has been infected by Trojans. |
| Scanner DLP [28] | Detects whether a hosts has generated a large amount of small flows to different hosts in a short period of time. And searches signatures in the flows generated by scanners. |
| Flow Monitor DLP | Monitors the bytes, packets and average byte per packet information of each flow. |
| HTTP Monitor DLP | Monitors the data being transferred through HTTP traffic and sort the hosts based on the volume of HTTP traffic. |

Table 4.1: Detection programs that we have written for our virtualized NIDS.

- Demonstrating the intrusion detection effectiveness of vNIDS. We run our virtualized NIDS and compare its detection results with those generated by *Bro* NIDS based on multiple real-word traffic traces (Figure 4.4).

- Evaluating the performance overhead of detection state sharing among instances in different scenarios: 1) without detection state sharing; 2) sharing all detection states; and 3) only sharing *global* detection states. The results are shown in Figure 4.5.

- Evaluating how fast each microservice can execute and scale compared with monolithically provisioned NIDSes. We compare the processing capacity and launch time of each microservice with those of the monolithically virtualized NIDS (Figure 4.6).

- Demonstrating the flexibility of vNIDS regarding placement location. In particular, we quantify the communication overhead between virtualized NIDS instances across different data centers that are geographically distributed (Figure 4.8).

- Demonstrating the flexibility of vNIDS regarding processing capacity. We compare vNIDS with Bro Cluster with respect to processing capacity and resource usage when the network traffic volume is changed (Figure 4.9).

### 4.4.1 Data Collection & Attack Trace Generation

Our dataset consists of two parts, the background traffic and the labeled attack traffic. The background traffic are from three different sources: *i)* the CAIDA UCSD anonymized Internet trace [16], which is a representative of Internet traffic; *ii)* the LBNL/ICSI enterprise trace [5], which is a typical traffic trace collected from an enterprise network; and *iii)* the campus network trace that was collected from our campus network gateway. The labeled attack traffic is generated by conducting penetration tests in an isolated environment. We have generated four attack traces for our DLPs.

- The `DNS.trace` contains DNS tunneling traffic generated by 20 different hosts. Therefore, we count the number of "malicious activity" in this trace as 20. We use scapy [26], a powerful interactive packet manipulation program, to generate the traffic.
- The `Cookie.trace` contains HTTP traffic with 100 different cookies that have been reused by multiple hosts. Therefore, we count the number of "malicious activity" in this trace as 100. We use Firesheep [14], an extension of FireFox, to generate the traffic.
- The `Trojan.trace` contains 20 victim hosts intruded by our penetration. The penetration follows the description in [46]. Basically, an attacker connect to the victim through SSH; then the attacher downloads a ZIP toolkit from a website; finally, the attacker escalates the privilege and upload a ZIP data file to a remote FTP server. Therefore, we count the number of "malicious activity" in this trace as 20.
- The `Scanner.trace` contains the traffic from 20 hosts that try to scan other hosts in the network. Therefore we count the number of "malicious activity" in this trace as 20. We use Nmap [21] to generate the scanning traffic.

We merged the above four traces as the `Attack.trace`. Then, we merged the `Attack.trace` with three different real-world background traffic as: `CAIDA+Attack.trace`, `LBNL+Attack.trace`, and `Campus+Attack.trace`. The merged traces are then replayed to the NIDS. We observe the output logs to determine how many attacks have been detected by the NIDS.

### 4.4.2 Detection Effectiveness of vNIDS

In this experiment, we evaluate the detection effectiveness of vNIDS. Our evaluation was based on the DNS Tunneling, Cookie Hijacking, Trojan, and Scanner DLPs. For each dataset, we have compared the detection results in four scenarios:

- We used the outputs of Bro NIDS as our base line of the experiment results (*Bro*).
- We implemented a version of the four DLPs that shares all the detection states and use them in our virtualized NIDS. Each DLP is run with multiple instances (*Share All*).
- We ran the detection state classification component in the vNIDS controller to generate another version of the four detection programs that enable state sharing between instances. Each DLP is run with multiple instances (*vNIDS*).
- We implemented a version of the four DLPs without state sharing between instances and directly run each DLP with multiple instances (*No Sharing*).

In our experiments, we first ran the *Bro* NIDS over the three datasets and take the outputs of Bro as the baseline of the detection results. We didn't find DNS, Cookie or Trojan malicious activities in any of the three real-world traces. The malicious activities detected by these DLPs are all from our `Attack.trace`. We found malicious activities for Scanner DLP in all three real-world traces. The malicious activities detected by the Scanner DLP are from real-world traces and our `Attack.trace`.

Then, we ran the *Share All* NIDS over the three datasets. The traffic is distributed to two instances randomly at per-flow granularity. The number of the malicious activities reported by the NIDS is the same as *Bro* NIDS. This implies that, by sharing all detection states, our virtualized NIDS can ensure the effectiveness.

Next we ran the *vNIDS* over the three datasets. The traffic is distributed to two instances randomly at per-flow granularity. We also observed the same number of malicious activities reported by the NIDS. This implies that vNIDS can also ensure the detection effectiveness by only sharing the *global* detection states.

Finally, to demonstrate the limitations of NIDSes without sharing detection states, we run the *No Sharing* NIDS. The traffic is distributed to two instances randomly at per-flow granularity. At this time, we observed a degradation of detection rate for all the three datasets. The comparison of

detection rates of NIDSes in four scenarios is shown in Figure 4.4. Figures 4.4-a, 4.4-b, and 4.4-c are based on the three traces, `CAIDA+Attack.trace`, `LBNL+Attack.trace` and `Campus+Attack.trace`, respectively.

We further examined the reasons for detection rate degradation in this scenario. We found that it was caused by distributing traffic relevant to the same malicious activity to different instances. For example, for the Scanner DLP, we totally observed 58 hosts that are conducting scanning through the Bro NIDS in the `CAIDA+Attack.trace`. The *No Sharing* NIDS only reported 45 hosts. We manually checked the flow rules used to distribute the traffic and examined the flows generated from the 13 hosts that are not detected. We found that those 13 hosts were not reported because their flows are almost evenly distributed to the two instances. For each instance, the scanning speed of those hosts is below the threshold.

As a summary, this experiment demonstrated that vNIDS can address the effective intrusion detection challenge for virtualized NIDSes and achieve equivalent outputs as traditional NIDSes.

### 4.4.3 Detection State Sharing Overhead

In this experiment, we have evaluated the overhead of detection state sharing in three scenarios described in the previous section: *Share All*, *vNIDS* and *No Sharing*.

We replayed the same traffic trace to the virtualized NIDS in three scenarios and observed the processing time of each packet. We chose the processing time as the evaluation metric because this time reflects how fast a packet can be processed. If a packet takes too much time to be processed, the NIDS instance cannot keep up with the packet transmission rate thus may drop packets.

Figure 4.5 (a) presents the packet processing time of each DLP. On the one hand, the results of this experiment demonstrated that sharing all detection states introduced non-trivial performance overhead to the virtualized NIDSes. On the other hand, for DNS Tunneling, Cookie Hijacking, Trojan, and Scanner DLPs, we observed that VNIDS saved 70.71%, 54.22%, 70.34% and 69.96% of the processing time compared with the approach that shares all detection states, shown in Figure 4.5 (b). Specially, VNIDS saved 92.23% and 99.55% packet processing time for the Flow Monitor and HTTP Monitor DLPs, respectively, because all the detection states employed by these two DLPs are *local* detection states. vNIDS keeps all of their detection states locally to each instance to significantly reduce the processing time of each packet. The Evaluation results demonstrated that sharing all detection states introduces non-trivial performance overhead and VNIDS enables only

*global* detection state to be shared to minimize performance overhead significantly.

### 4.4.4 Microservice Efficiency

In this experiment, we have evaluated the efficiency of virtualized NIDSes provisioned as microservices. First, we evaluated how fast each microservice instance can launch with various detection programs installed and compared them with the launch time of the monolith NIDS instance. Then, we compared the processing capacity of each microservice instance with that of monolithic NIDS instance. The launch time is used to estimate how fast the microservice instance can scale out by launching new instances. The processing capacity reflects the number of CPU cores required to achieve a specific performance requirement. The greater the processing capacity, the less CPU core is required to achieve a specific performance requirement, therefore, the instance requires less resources to run.

We have implemented our microservices based on Click elements. For the monolithic NIDS, we include all the elements of three microservices and instantiate them within a single instance. Notice that, there are four cases (Figure 4.2) with respect to composing microservice chains. In our experiment results, if a microservice is not included in microservice chain, we set the processing time of that microservice as 0.

Based on the evaluation results shown in Figure 4.6 (a), we observed that, for all the detection programs, the launch time of each microservice instance was less than the launch time of the monolithic NIDS instance, though the sum of the launch times of all microservice instances was greater than the launch time of monolithic NIDS instance. This is because each microservice instance includes less Click elements than the monolithic NIDS instance but some elements have been reused by multiple . The results of this experiment demonstrated that microservice instances can launch faster than monolithic NIDS instances thus can scale faster.

We also compared the processing capacity of each microservice instance with that of the monolithic NIDS instance. Figure 4.6 (b) shows the comparison results. We observed that all the microservice instances have greater processing capacity than the monolithic NIDS instance. That means, every microservice instance requires less resource than the monolithic NIDS instance to achieve equal processing capacity. Therefore each microservice instance can be provisioned with less resource than the Monolithic NIDS instance.

### 4.4.5 Flexibility of Placement Location

In this experiment, we evaluated the flexibility of vNIDS regarding placement location. The experimental environment was built on CloudLab involving two data centers (*Site-1*) and (*Site-2*) that are geographically distributed. In the experiment, we wanted to demonstrate that under the situation where some applications (Applications B) have migrated from *Site-1* to *Site-2*, vNIDS have advantages over traditional NIDSes due to their flexibility of placement location. Figure 4.7 demonstrates the different behaviors of two types of NIDSes.

Originally, all applications are running *Site-1* and the NIDS is provisioned at *Site-1* to protect those applications. If Applications B are migrated to *Site-2* due to resource management or optimization purpose, as traditional NIDSes are difficult to migrate, in this case the network traffic relevant to Applications B must be rerouted from *Site-2* back to *Site-1* for processing. In contrast, vNIDS can flexibly provision new instances at *Site-2* to process the traffic of Applications B. The only concern is the communication traffic, the volume of which we have quantified in this experiment, between virtualized NIDS instances located at different sites.

During the experiment, we divided each of our dataset into two parts, representing the traffic of Applications A and Applications B respectively. Then we replayed the two parts of each dataset at two sites respectively. In the traditional NIDS case, traffic replayed at *Site-2* was rerouted to *Site-1* and we evaluated the bandwidth required by the rerouting. In the vNIDS case, traffic was replayed at *Site-1* and *Site-2* separately and was processed by instances at corresponding sites. We evaluated the bandwidth required by the communication between instances in the two sites.

The required bandwidth between *Site-1* and *Site-2* for traditional NIDS and vNIDS respectively is shown in Figure 4.8. We tested the three real-world datasets (without synthetic attack traces injected) and found that the bandwidth required by vNIDS is much less than traditional NIDS. Especially, for `Campus.trace`, traditional NIDS requires 1000x more bandwidth than vNIDS. This is because vNIDS only needs to exchange *global* detection states between the two sites, while traditional NIDS needs to reroute all the network traffic between the two sites. Another important observation is that for all three real-world datasets, the communication bandwidth between virtualized NIDS instances surprisingly low – only hundreds of Kbps – for the three datasets. This is because the amount of *global* detection states is relatively small.

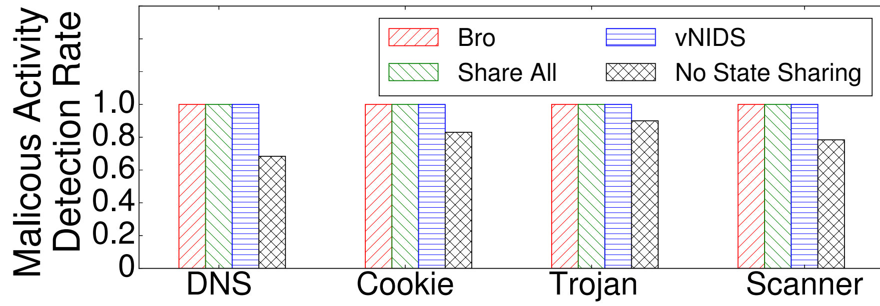
### 4.4.6 Flexibility of Processing Capacity

In this experiment, we evaluated the flexibility of vNIDS regarding processing capacity and compared it with Bro Cluster. We only used `Campus.trace`, which has the highest traffic rate, for this experiment. We ramped the traffic sending rate gradually from 0 up to 10Gbps in 100 seconds and observed the throughput of Bro Cluster and vNIDS. We deployed 5 instances for Bro Cluster in the experiment because each Bro Cluster instance, estimated against our dataset, can roughly handle 1Gbps traffic. In contrast, vNIDS scaled according to the traffic volume. Once existing virtualized NIDS instances are about to overload, we created a new instance and redirected some traffic to the new instance.

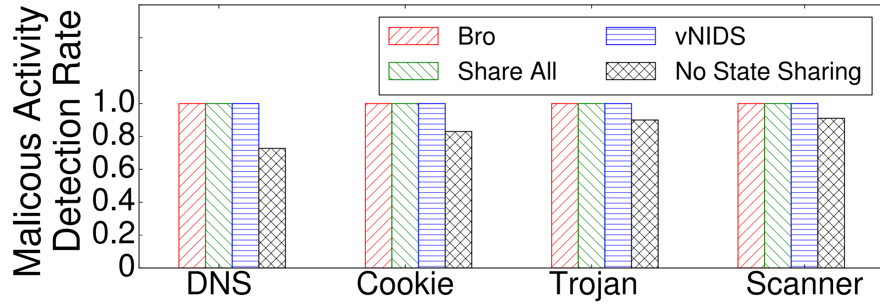
Figure 4.9 depicts our observation of this experiment. Figure 4.9-a indicates that the throughput of Bro Cluster and vNIDS steadily increased as the traffic volume grew. Once the traffic volume reached the maximum processing capacity of Bro Cluster, the throughput of Bro Cluster is limited to less than 6Gbps. For vNIDS, as it can quickly launch new instances, its throughput grew consistently with the traffic volume. Figure 4.9-b depicts the number of provisioned instances for Bro Cluster and vNIDS over time. Bro Cluster provisioned static number of instances. Virtualized NIDS, in contrast, provisioned its instances according to the traffic volume. As the traffic volume grew, vNIDS provisioned an increasing number of instances to handle the traffic. We saw that when the traffic volume was under 5Gbps, the Bro Cluster is *over-provisioning*, while the traffic volume exceeded 5Gbps, the Bro Cluster became *under-provisioning* (overloaded in Figure 4.9-a).

## 4.5 Conclusion

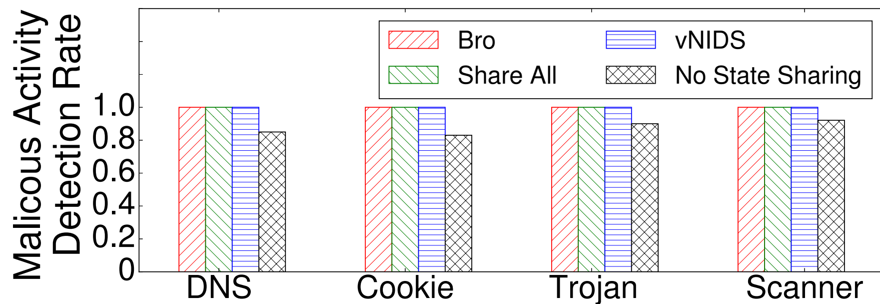
In this task, we have propose vNIDS, a new NIDS architecture, to achieve safe and efficient virtualization of NIDSes. We have carefully designed solutions in vNIDS to address two key challenges including *effective intrusion detection* and *non-monolithic NIDS provisioning* in NIDS virtualization. We have implemented a prototype of vNIDS, involving three microservices and the vNIDS controller, and evaluated them with six detection logic programs based on real-world traffic traces. Our evaluation results have demonstrated the safety and efficiency of vNIDS for NIDS virtualization.



(a)



(b)



(c)

Figure 4.4: Detection rate of known malicious activities for Bro, vNIDS, Sharing all states, and No sharing solutions. The experiments are based on three different real-world traffic with generated attack traces: (a) CAIDA+Attack.trace; (b) LBNL+Attack.trace; and (c) Campus+Attack.trace.

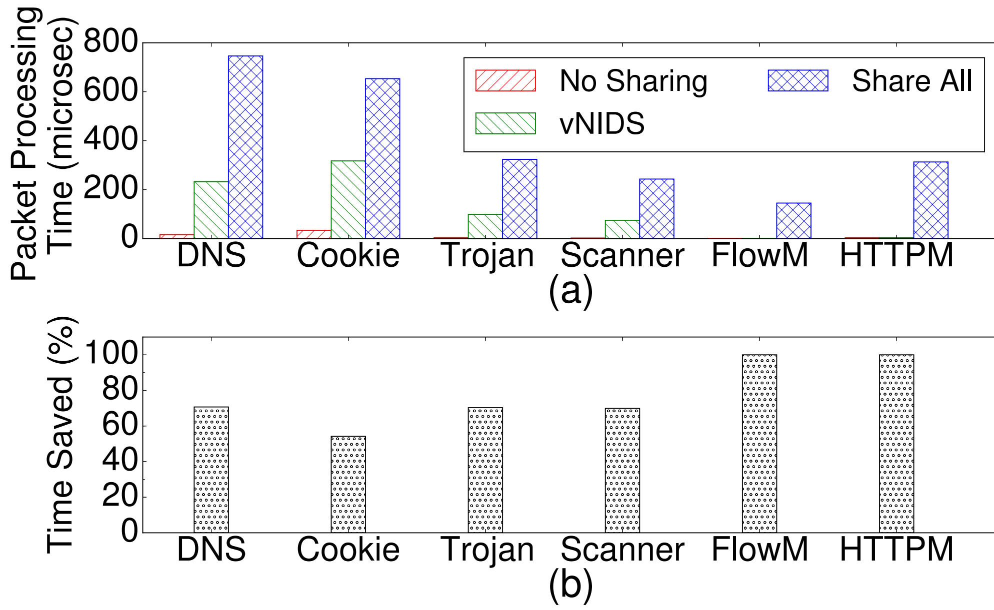



Figure 4.5: (a) Packet processing time of each detection program in three scenarios. (b) Packet processing time reduced by vNIDS compared with sharing all detection states.

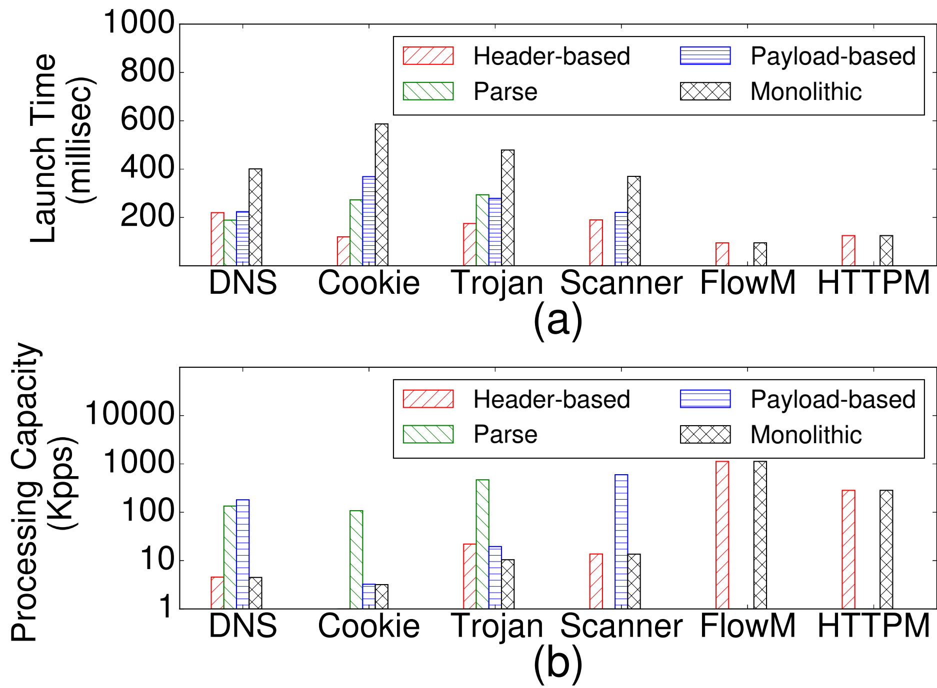



Figure 4.6: Launch time (a) and processing capacity (b) of *header-based detection*, *protocol parse*, *payload-based detection*, and Monolithic NIDS instances.

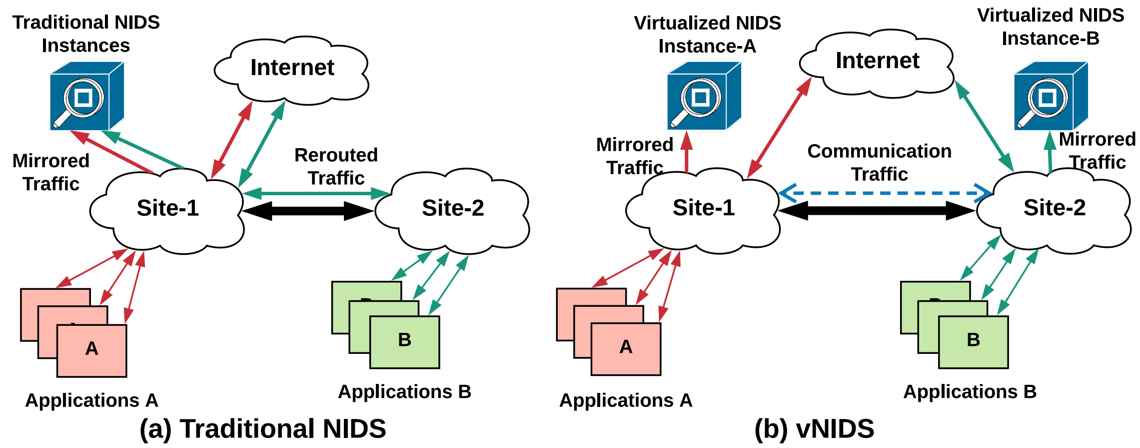



Figure 4.7: (a) Traditional NIDS requires rerouting traffic of Applications B back to *Site-1* for processing. (b) vNIDS can provision its instances flexibly at *Site-2*, if Applications B have migrated to *Site-2*.

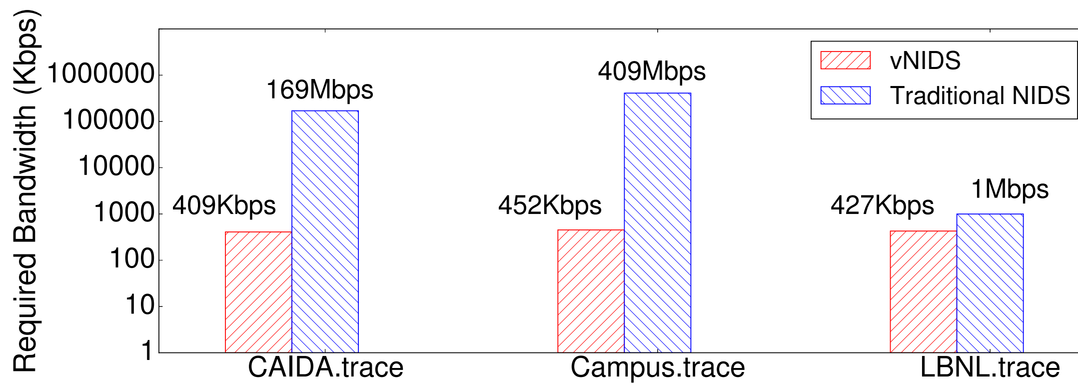
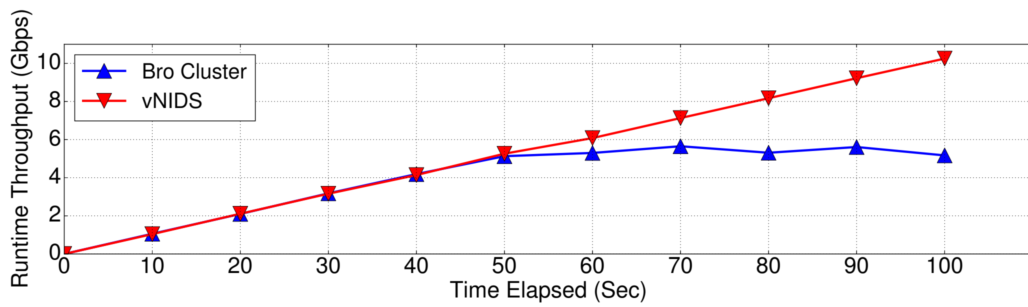



Figure 4.8: Required bandwidth between *Site-1* and *Site-2* for vNIDS and traditional NIDS respectively.



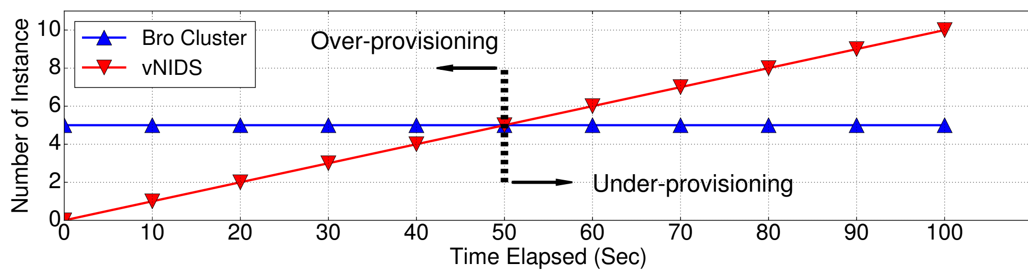




Figure 4.9: (a) Runtime throughput of vNIDS and Bro Cluster. (b) Number of instances of vNIDS and Bro Cluster.

# Chapter 5

# Expressive Security Directives for Host Security Functions

## 5.1 Introduction

we have altered the design of two critical security functions, firewalls and NIDSes, in the network level by leveraging the advances of SDN and NFV. In the host level, the situation seems better as security functions are already implemented in software. Nevertheless, the shortcoming of the security functions in the host is that most of them can only support simple binary access control actions, such as allowing or denying. This calls for the work that provides more expressive security directives for network security functions.

Inspired by the SDN flow rules, we define the host security directives in a "Match-Action" paradigm called system flow rule. A system flow rule consists of two fields, match field and action field. The match field will match against an event. The action field describes the response action of the security directive. The system flow rule is stored in a system flow rule table, which is implemented in the kernel of operating systems.

To allow those expressive security directives to be used by various host security functions, we abstract the events in the host with a flow-based model called system flow. The events that are covered in this dissertation are resource access including the following two types.

- A process accesses a file. This type of events will be able to capture any illegal file operations.

- A process accesses a network socket. This type of events will be able to capture any in-going and out-going traffic of a process.

The events are formalized with a three-tuple, including the source, destination, and operation. The source indicates which process initiate this event. The destination indicates which file or network socket this event targets to. The operation depicts how the process access a file or socket. In this dissertation, we will cover six operations, *read*, *write*, *open*, *close*, *create*, and *delete* a file or socket.

There are two research challenges for this research task.

- **C1:** How to abstract system activities and model system capabilities for *unified programmability*?
- **C2:** How to enable complicated, practical *security functions* to the flow model for *diverse threat response*?

## 5.2 Methodology

### 5.2.1 System Flow Overview

Inspired by SDN, we introduce a novel flow-based model for system activity abstraction, called system flow. An SDN data plane utilizes a flow table to match network packets characterized by various patterns (e.g., src/dst IP and src/dst port) and to provide aggregated views on the network traffic. It facilitates communication management amongst a set of (heterogeneous) network devices. Whereas, system flow defines a set flow rules that manage the access to various system resources, such as file, memory, pipe, socket, etc., during a certain time interval. System flow constructs a flow table with a flow match in an entry characterized by src/dst system objects/resource, system operations, and metadata. Note that, based on this flow definition, system flow provides system-oriented aggregated views on the system activities, just as SDN supplies common low-level network views and building blocks via a southbound interface upon which developers can write arbitrary controller applications that can modify the underneath flow tables.

Figure 5.1 depicts the overview of system flow model. In general, the whole system flow model works in the kernel space. At the very top, it provides OS-specific hooks for system calls, which are the only ways user-space processes interact with the system resource. Once those hooks capture the interactions (system calls), they extract the parameters and send them to an Event

Generator. The Event Generator then encapsulates the interactions with events, which is then sent to the Flow Table Manager. The Flow Table Manager maintains a flow rule table and matches the events against the flow rules. After that, Flow Table Manager notifies the Action Scheduler with the matched flow rules. The Action Scheduler responses to the actions in those flow rules according to the priorities of them. At this moment, there are two possible fates of the actions. One is that the action is delivered back to the hooks from where the events are generated so that the hooks will determine whether they should proceed. The other is that the action is delivered to host security functions where the actions receives further processing, which is defined by the logic of host security functions. In this case, security functions may update the flow rule table via Flow Table Manager.

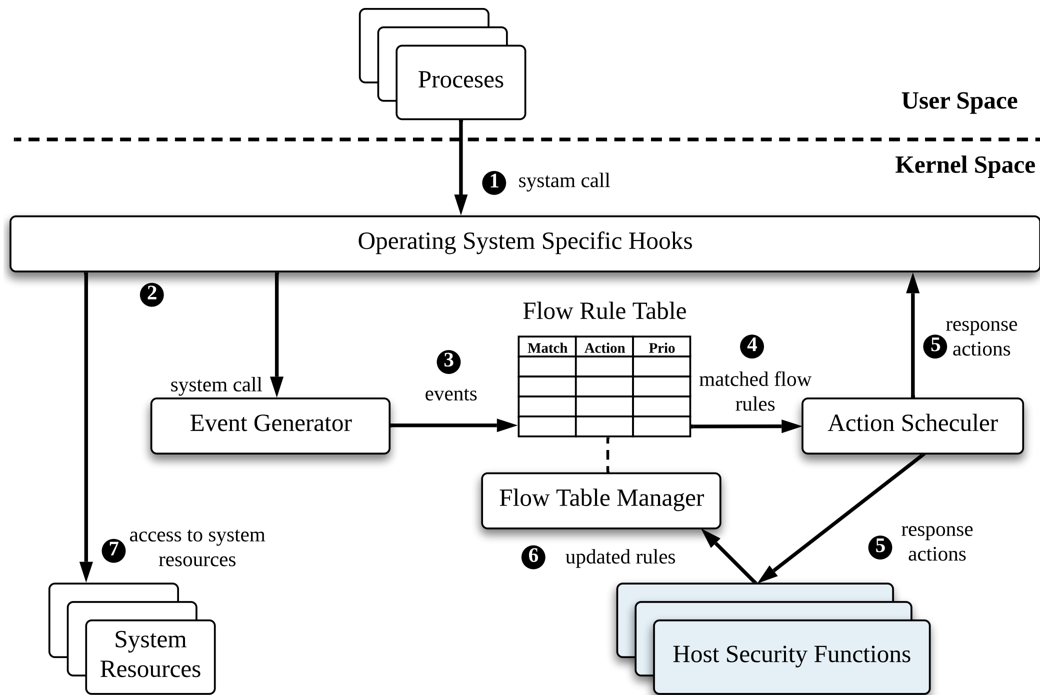



Figure 5.1: Overview of system flow.

### 5.2.2 Event Generator

In infrastructure, host systems may include a variety of critical activities, which should be monitored or controlled according to different security policies. Hence, we define a general concept of *system events* to model such critical system activities. In this section, we particularly adopt system events to cover interactions between processes and different system resources (e.g., file, socket, and

IPC) as listed in Table 5.1.

Table 5.1: Supported system events

| **Type** | **Description** |
|---|---|
| file_ioctl | control attribute of file |
| file_read | read file |
| file_write | write to file |
| file_append | append to file |
| file_create | create file |
| file_delete | delete file |
| file_getattr | get file attribute |
| file_setattr | set file attribute |
| file_lock | lock file |
| file_unlock | unlock file |
| file_open | open file |
| file_close | close file |
| socket_ioctl | control attribute of socket |
| socket_read | read socket |
| socket_write | write to socket |
| socket_append | append to socket |
| socket_getattr | get socket attribute |
| socket_setattr | set socket attribute |
| socket_create | create socket |
| socket_delete | delete socket |
| socket_bind | bind to socket |
| socket_connect | connect to socket |
| socket_listen | listen to socket |
| ipc_create | create ipc object |
| ipc_delete | delete ipc object |
| ipc_read | create ipc object |
| ipc_write | delete ipc object |
| ipc_getattr | get ipc object attribute |
| ipc_setattr | set ipc object attribute |
| memory_map | create a shareable memory region |
| memory_protect | modify protection flags of memory |
| memory_allocate | allocate new virtual memory mapping in mm structure |
| process_create | create a new process |
| process_execve | load executable program for a program |

In the case where developers need extra events, they can also define their own system events. The Event Generator component is used to generate system events by intercepting system-level activities (e.g., system calls) from hooks in host systems. It also interprets the semantics of system events by parsing parameters from those hooks if necessary. In addition, Event Generator also provides an interface, called *sysflow_generate_event*, to allow users to generate their own system

events.

In many cases, the security application developers encounter semantic gaps for system objects, i.e., they can hardly specify the identifier of system objects. For example, a security application developer may not know the identifiers (i.e., *UUID* and *inode number*) of tax files in the file system of the victim host when they want to write security apps to prevent the ex-filtration attacks. Instead, they may be aware of the file name and possibly the path. To bridge the semantic gap, Event Generator enables an identifier binding and resolution service for system objects.

At run-time, Event Generator maintains the profile table for system objects (Table 5.2). For example, Event Generator will keep the executable name of a process in addition to the process identifier. When receiving the requests to update flow rules that include attributes of system objects instead of identifiers, the Event Generator will refer to the profile table to retrieve the identifier of the system object. We note that, in some cases, the map from the attributes of a system object to its identifier is not unique. For example, the process name may map to multiple running processes. In this case, we will install flow rules for each of them. In addition, the binding service will also monitor the change of the mapping from non-identifier profiles to the identifier (e.g., from name to *UUID* and *inode number* for a file) at run-time and update the inductive flow rule accordingly.

Table 5.2: Example system object attributes.

| **System Object** | **Example Profile** |
|---|---|
| Process | pid, ppid, pname, stime, etime, etc |
| File | uuid, inode number, fname, path, etc |
| Socket | uuid, inode number, local/remote addr/port, etc |
| Memory | virtual address, physical address, etc |

### 5.2.3 Flow Table Manager

#### 5.2.3.1 System Flow Rule

We introduce *system flow rules* to model system security capabilities upon a sequence of system events. The system flow rule is formally defined by the syntax in Figure 5.2. System flow rules are used to capture system security intents, which include *match*, *action*, *priority*. A *match* is a predicate to match a sequence of system events that have the same attributes, i.e., *source*, *destination*, or *type*. The source of a system flow is an identifier for the initiator of the flow, which is an identifier of a system application. The destination of a system flow is an identifier for the receiver

```
Notations: Integer n, Wildcard *, Text string

Match                       ma     ::= <src, dst, type>
Source ID                   src    ::= id | *
Destination ID              dst    ::= id | *
Resource ID                 rid    ::= id | *
Operation Type              type   ::= string | n
ID                          id     ::= {n1, n2, …, nk}

Action                      act    ::= pa | (act | act) | (act >> act)
Primitive Actions           pa     ::= allow | deny | report | message |
                                       log | encode(tag) | decode(tag) |
                                       redirect(dst1, dst2) | quarantine(n)|
                                       external(id)
Tag                         tag    ::= string

Priorities                  pri    ::= n

System Flow Rule            rule   ::= <ma, act, pri>
```

Figure 5.2: Syntax of system flow rule.

of the flow, which is an identifier of system resources, such as files, memories, etc. The type of system flow is used to classify the different interactions between system applications and resources, e.g., writing a file. Note that a system flow can be used to represent an exact system event or a group of system events with the same pattern by using a wildcard notation (*). For example, a system flow can be specified as $\{src : *,\ dst : file_1,\ type : file_op_write\}$ to match system events representing any process writes to $file_1$.

Table 5.3: Security actions that we define.

| Name | Descriptions |
|---|---|
| Allow | permit matched system events pass through |
| Deny | prevent matched system events pass through |
| Encode | push contextual tag into outgoing packets |
| Decode | check contextual tag from incoming packets |
| Redirect | redirect matched system events to a new destination |
| Quarantine | restrict a process under a specific running context |
| Report | report alerts to administrators |
| Log | log matched events based on conditions |
| Message | pop up a prompting window to notify the host user. |

A system flow rule uses a list of *primitive actions* (as listed in Table 5.3) to specify how the system events should be processed. The intuitive primitive actions are *allow* and *deny*, which are used to enforce explicit access control upon matched system events. *quarantine* aims to constraint/isolate a specific process under a fine-grained resource context and access. Also, *log* can be used to record system events for further analysis, e.g., digital forensics. *encode* is to push contextual

tag into outgoing network packets, and *decode* is to check the contextual tag from incoming network packets, which can be used to enforce cross-host information flow tracking. Furthermore, *redirect* aims to change the system flow to a new location, which is helpful to enforce a deception-based defense. Besides, *alert* and *message* are proposed to notify administrators or host users of suspicious/abnormal events. We note that the defined security actions are basic security primitives that we abstract from many existing system security applications. In addition, a system flow can also use *external* primitive action that is customized by a security application developer. An integer-based priority is used to disambiguate rules with overlapping patterns. If a system event matches multiple system flow rules, only the highest-priority rule is applied. In this case, the priority can be used to explicitly specify the matching order to resolve potential conflicts from a set of system flow rules.

The system flow rule can be used to enforce a wide variety of basic security intents. For example, a system flow rule "*src: *, dst: file_1, type: file_open, priority:0, actions: block*" can be used to disallow any open operations on a specific file.

#### 5.2.3.2 Efficient Flow Rule Management

The low-level security intents is embedded in a table of system flow rules, called system flow rule table. We note that the flow rule in system flow rule table can have two types of match patterns, i.e., *exact match* and *wildcard match*. In order to support both of them, an intuitive solution for system flow rule table is to use a bit-wise classification, i.e., we compare the incoming system events with all system flow rules in the table bit by bit. However, in such a solution, the time complexity for flow table lookups and updates is $O(R)$, where $R$ is the number of system flow rules resided in a running system. As a result, security intent enforcement will incur a high latency.

In order to support an efficient flow table update and query, we adopt Tuple Space Search (TSS) classification algorithm [141]. The key insight of TSS classification is to realize a flow table as a set of hash tables. In each hash table, it stores the hashed key for each specific system flow rule with the same mask. Suppose all the flows in a system flow rule table matched on the same fields in the same way, e.g., all flows match the source and destination system object but no other fields. In such a case, TSS implements a flow table as a single hash table. If a new flow with a different match is added, TSS generates another hash table that handles the new match for the flow. To denote matched fields, we use a 3-bit mask to specify the range of each hash table. Based on the TSS algorithm, Flow Table Manager can provide an efficient flow table management, e.g., flow rule

update/lookup, with time complexity of $O(1)$. [1]

### 5.2.4 Action Scheduler

#### 5.2.4.1 Built-in Security Actions

We next discuss the implementation of built-in security actions in our System Flow model (listed in Table 5.3). We must note that the implementation of built-in security actions is OS-specific, depending on how the host OS can support certain operations, such as access to files, memory, and socket. For discussion purpose, we assume that our built-in security actions will be implemented on the Linux OS.

The *allow* and *deny* directives allows and denies, respectively, certain operations, including *open*, *read*, and *write* a file. These two directives can be easily enforced through the *file_permission* or *inode_permission* hooks in Linux Security Module (LSM). The *quarantine* directive is used to set up a group of system flow rules with *allow* and *deny* actions. The *redirect* directive is intended for redirecting operations to another object. For example one can redirect the "open file" operation to a decoy file so that a process will open the decoy file instead of the original file. However, the enforcement of *redirect* action is quite challenging due to only relying on the existing hooks of LSM. Therefore, we place an additional hook to handle the *redirect* action. Since a process in the user space manipulates a file through a file descriptor, we place a hook (*fd_bind*) before the file descriptor is bound to a specific inode. By invoking the *fd_bind* hook, one can enforce the *redirect* action via replacing the inode of the original file with the inode of any other files according to the security intents. The *encode* directive is for tagging the outgoing packets, and we can get the tagging information from the incoming packets by *decode* directive. In order to associate process identifiers with their outgoing packets, we embed two new hooks in LSM from Linux kernel functions (i.e., ip_queue_xmit() for *encode* and ip_rcv() for *decode*), which can help correlate process identifiers with their outgoing packets. Then, the *encode* and *decode* leverage packet tags (using IP TOS field in outgoing or incoming packets) to exchange contexts cross multiple hosts. The *alert* directive is implemented as *flow rule status report* messages that notify security administrators. The *log* directive is implemented to write to a system log file, "/var/log/messages", in Linux, and other system log files for other operating systems.

[1] The flow rule lookup with TSS needs $T$ hashed memory accesses, where $T$ is a constant value (i.e., 8 in the work) of the number of tuples. The flow rule update with TSS needs 1 hashed memory accesses.

#### 5.2.4.2 Security Action Scheduling

For a specific incoming system event, System Flow can specify multiple security actions based on user-defined control logic. By default, the Action Scheduler will response to all of those actions in parallel to improve its efficiency. That is, the Action Scheduler will create multiple copies of action responses and feed them to different destinations. If the action is for access control only and can be enforced by the hooks directly, Action Scheduler will feed this action response back to the hooks. In this case, the hooks will determine whether they should go forward to allow the access or return a permission error to block the access. In another case, if the action response needs further processing with certain host security functions, Action Scheduler will feed this action response to the corresponding security functions simultaneously.

### 5.2.5 Host Security Functions

The insight of Host Security Functions is to provide a dedicated mechanism that can maintain a global view of different events across the system. The security functions can maintain states between events, implement comprehensive logic that is necessary for sophisticated security applications. For example, many system security applications may locate suspicious/malicious incidents by inspecting a sequence of system calls, e.g., detection of TOCTTOU attacks [36, 42, 47, 147] and Link Following attacks [154, 40]. In those case, security functions can be developed to maintain statistics of a serials of system calls. And if if necessary, those security functions can dynamically update the system flow rules via Flow Table Manager. To update the system flow rules in the system flow rule table, security functions must specify the flow rules that they want to update by following the flow rule syntax defined in Figure 5.2.

## 5.3 Evaluation

In this section, we present our experimental results from evaluating the performance of our System Flow model for micro-benchmark tests and scalabillity tests. The tests are conducted under Linux OS. In the following evaluations, we leverage *baseline* to refer to systems running an unmodified Linux kernel.

### 5.3.1 Micro-Benchmark Results

We used LMBench [7] and Unix-Bench [138] to evaluate the run-time performance of system calls, file operations, memory latencies, and socket I/O throughput. Table 5.4 and Table 5.5 depict the comparison between the baseline and System Flow with a variety of applications deployed.

Based on the results of LMBench, two of the three primary file operations (i.e., *read*, and *write*) introduce reasonably low overhead (with less than 4%). Based on the results of Unix-Bench, most of the operations introduce low overhead (with less than 4%) except the system call overhead, which is 10.38%. This is because the hooks are implemented in the system calls. Overall, the results indicate that System Flow mainly impacts the file open/close operations, mmap latency and system calls. But for other file or socket operations, System Flow only introduces negligible overhead.

Table 5.4: LMbench results for System Flow.

| **System Operation** | **Baseline** | **System Flow** |
|---|---|---|
| Latency of system operations in ms (smaller is better) | | |
| file read | 0.1913 | 0.1979 (+3.45%) |
| file write | 0.1879 | 0.1914 (+1.86%) |
| file open/close | 0.4875 | 0.5612 (+15.12%) |
| file create (0k) | 2.9717 | 3.0629 (+3.06%) |
| file create (10k) | 6.6968 | 7.1788 (+7.20%) |
| file delete (0k) | 4.9388 | 5.0660 (+2.58%) |
| file delete (10k) | 3.4770 | 3.6923 (+6.19%) |
| syscall | 0.0359 | 0.0367 (+2.23%) |
| mmap latency | 3418.0 | 3940.0 (+15.27%) |
| pipe latency | 2.0840 | 2.1471 (+3.03%) |
| Socket throughput pps (larger is better) | | |
| socket I/O | 885 | 883 (-0.22%) |

### 5.3.2 Scalability with Flow Rules

Moreover, we tested the scalability of our System Flow model using different numbers of system flow rules. Figures 5.3, 5.4, and 5.5 depict the cumulative distribution function (CDF) of the performance of three most frequently used operations, file *read*, file *write*, and socket I/O. We increased the number of system flow rules from 1000 up to 100,000 in our experiments. The actions of these rules are all *allow* and most of them will only match specific system objects (e.g. specific file). The testing results show that the number of system flow rules does not make significant differences to the performance of file read, file write, and socket throughput since the flow table is implemented

Table 5.5: UnixBench resulted sources for System Flow.

| **Benchmarks** | **Baseline** | **System Flow** |
|---|---|---|
| Dhrystone 2 using register variables | 4380.1 | 4379.2 (-0.02%) |
| Double-Precision Whetstone | 447.5 | 440.3 (-1.62%) |
| Execl Throughput | 1565.8 | 1548.4(-1.12%) |
| File Copy 1024 bufsize 2000 maxblocks | 2867.2 | 2839.3 (-0.97%) |
| File Copy 256 bufsize 500 maxblocks | 1922.8 | 1912.0 (-0.56%) |
| File Copy 4096 bufsize 8000 maxblocks | 5259.3 | 5092.7 (-3.18%) |
| Pipe Throughput | 1595.0 | 1582.0 (-0.92%) |
| Pipe-based Context Switching | 1202.9 | 1177.7(-2.10%) |
| Process Creation | 1611.6 | 1569.1(-2.65%) |
| Shell Scripts (1 concurrent) | 2935.2 | 2903.8(-1.07%) |
| Shell Scripts (8 concurrent) | 2697.9 | 2679.1(-0.70%) |
| System Call Overhead | 3837.5 | 3439.1(-10.38%) |
| System Benchmarks Index Score | 2122.5 | 2097.1(-1.20%) |

through a hash table.

In addition, we tested the memory overhead introduced by the System Flow through the *top* Linux command with different numbers of system flow rules. The result shows the memory usage is about 400 KB for 1000 flow rules and it grows linearly with the number of system flow rules inserted. Hence, System Flow is scalable to contain system flow rules for various system security intents.

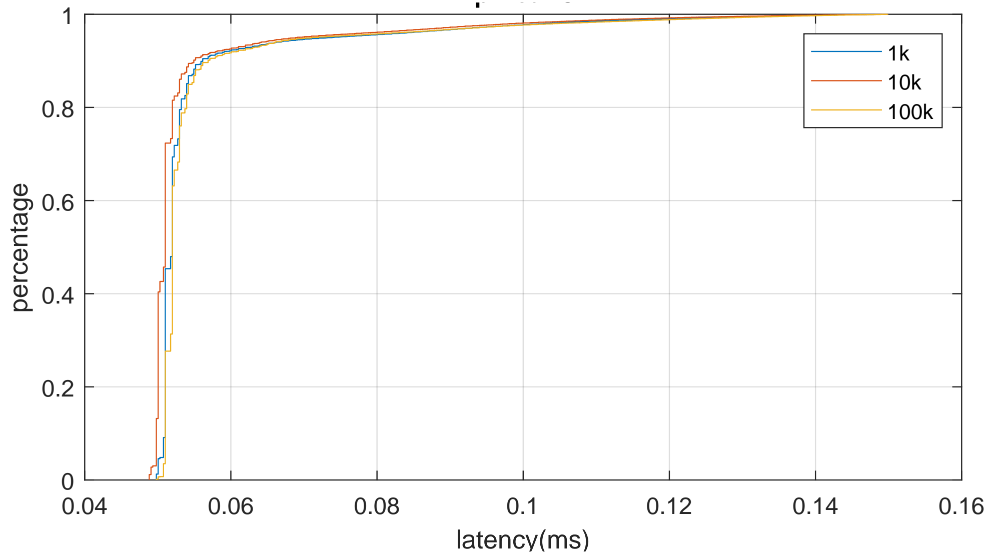



Figure 5.3: CDF of the latency of file read operation with various numbers of system flow rules installed.

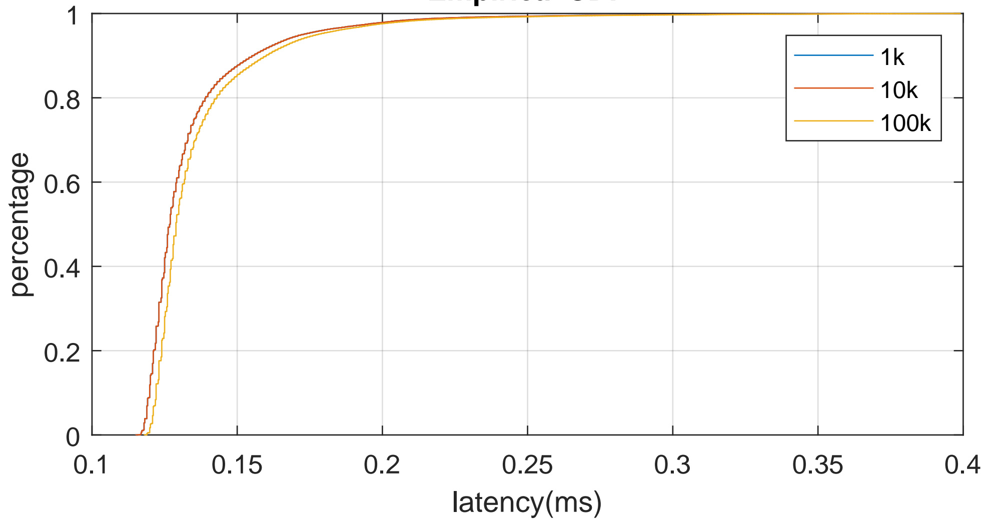



Figure 5.4: CDF of the latency of file write operation with various numbers of system flow rules installed.

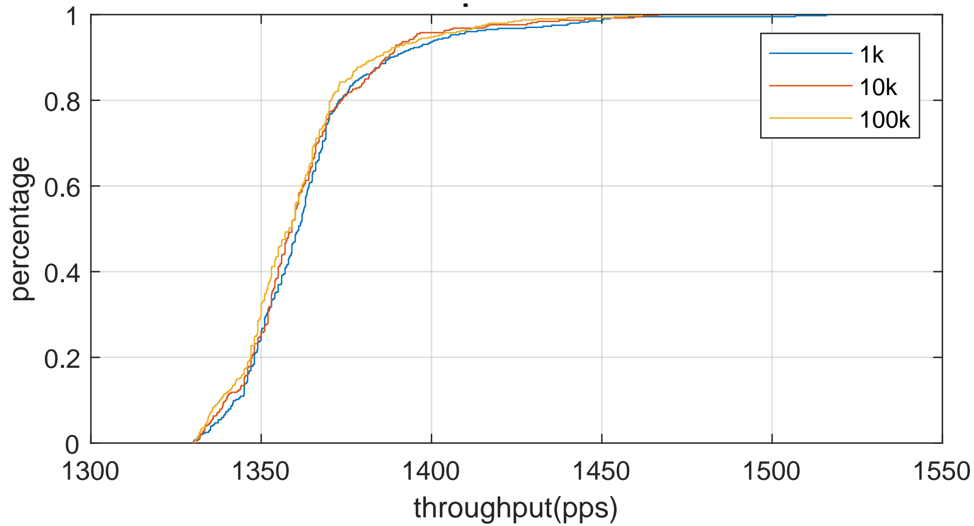



Figure 5.5: CDF of the socket throughput with various numbers of system flow rules installed.

## 5.4 Conclusion

In this task, we present a novel system security development framework for programmable security control of host system activities at run time. System flow model abstracts low-level system activities into a flow-based abstraction, provides dynamic flow-level control at run time, and offers a unified programmable interface for developers to develop various host security functions. The evaluation shows that our system flow model is useful to design diverse system security apps and

only introduces minor run-time overhead.

# Chapter 6

# Case Study: Defending malware in IoT with ProSec

## 6.1 Introduction

Linux has been widely used, especially, nowadays by the overwhelming number of the so-called "Internet of Things" (IoT) devices. The rocketed population, poor security, and 24/7 online properties make Linux-based IoT devices ideal targets for attackers. We have witnessed a number of catastrophes caused by large groups of compromised IoT devices in the past years. However, due to the budget constraints and security vulnerabilities of Linux-based IoT devices, protecting them from being compromised presents one of the greatest challenges. Recent research has recognized three main stages of IoT device compromise–intrusion, infection, and monetization.

In this chapter, we develop an approach to Linux-based IoT malware defense as a use case utilizing ProSec. This use case will demonstrate the usefulness of ProSec by showing how some domain-specific requirements can be satisfied easily through ProSec. First, the majority of IoT malware intrusion is based on vulnerable passwords, which makes it difficult if not impossible to distinguish legitimate login from intrusion by only looking at the network traffic. Therefore, it is critical to consider both network-level and host-level information to detect IoT malware. Second, most IoT devices are resource and budget constrained. As a result, state-of-the-art host-based security approaches can hardly be deployed due to their high-performance demands. Third, IoT malware

monetizes in various ways, such as launching DDoS attacks, stealing data, and cryptocurrency mining, which may cause real-world damages or financial loss. What makes matters worse, it presents a great challenge to detect and block those harmful activities before they actually take place, which requires the so-called early detection.

Our defense approach is developed based on ProSec. In the first place, we try to detect and prevent malware in the early stage, which is before the compromised IoT devices are monetized. To achieve this goal, a host-based infection detector and infection prevention mechanism are used to detect infection attempts and prevent detected infections from proceeding, respectively. As a second line to defense IoT malware, a virtual NIDS and virtual firewall are deployed to detect and block network traffic that is generated during by malware in the monetization stage.

To evaluate our defense approach, we have deployed it to high-fidelity software (virtual) IoT devices (as honeypots) on public clouds across several geographically different locations for one week. And we also quantify the performance overhead of our defense approach on real IoT devices.

## 6.2 Host-based IoT Malware Infection Detection

### 6.2.1 Understanding IoT Malware Infection

#### 6.2.1.1 Identifying IoT Malware Infection Process

A recent research on IoT malware provides a deep understanding on Mirai [33]. Authors of [33] revealed that Mirai comes with a separate loader program, which asynchronously infects vulnerable IoT devices by logging in through brute-force attempts, determining system environment, downloading and executing architecture-specific binaries, and finally deleting downloaded binaries. We further investigate the source code of Mirai and recognize that a number of shell commands are used to perform the above operations as shown in Fig. 6.1 (left). Another research devoting to a different IoT malware, Hajime [73], also reveals similar infection behaviors, which include logging in, and downloading and executing malicious binaries in the target system via shell commands. Besides, a prior study based on an online honeypot system, IoTPot [122], discovers that during the infection stage, IoT malware executes a sequence of shell commands to detect system architecture, download and execute malicious binaries, and remove the downloaded binaries.

Based on those key investigations, we find that a general IoT malware infection process can

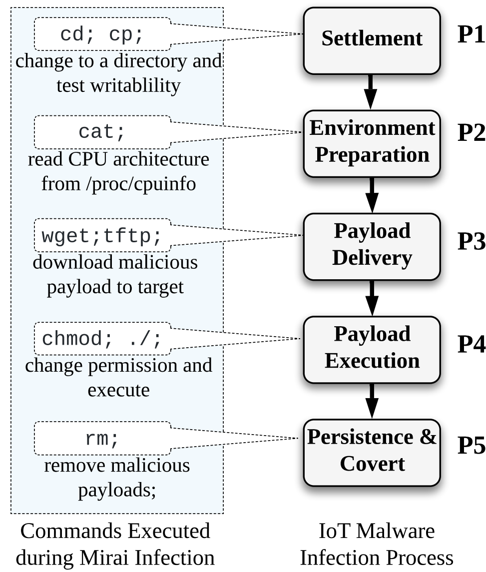



Figure 6.1: Shell commands executed during Mirai infection (left), and the five phases of IoT malware infection process that we identified (right).

be divided into 5 basic phases – *Settlement*, *Environment Preparation*, *Payload Delivery*, *Payload Execution*, and *Persistence & Covert*. Fig. 6.1 (right) illustrates the 5 phases in sequence. In the first phase, Settlement (1), malware tries to find a writable place as the working directory. Then, during the Environment Preparation (2) phase, malware collects necessary information from the system. After that, during the Payload Delivery (3) phase, malware delivers malicious payload to the target system. The following phase is Payload Execution (4) where malware executes the delivered payload on the target system. After the payload has been successfully executed, malware seeks ways to persist and covert footprint in the Persistence & Covert (5) phase. More importantly, we also observe that an IoT malware infection process is accomplished by executing a serials of shell commands and these commands must be executed in a certain sequence. For instance, as shown in Fig. 6.1 (left) for commands executed during the Mirai infection, `wget` must be after `cd` because the malware needs to enter a writable directory before it can download a file; `chmod` must be after `wget` or `tftp` because malicious payloads need to be delivered before changing their permissions; and `rm` should be after `./` because malicious payloads must be executed before they are removed. To verify the infection process identified in Fig. 6.1 (right), we collect the source code of 3 IoT malware and reverse-engineering reports of 18 different families of IoT malware online and compare the infection process with them. A summary of the comparison is listed in Table 6.1, which illustrates that the

Table 6.1: Summary of the phases of infection process involved in 3 open-source IoT malware and 18 reverse-engineering reports.

| Source | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| Wifatch [161] | × | ✓ | ✓ | ✓ | ✓ |
| LightAidra [92] | × | ✓ | ✓ | ✓ | × |
| Mirai [109] | ✓ | ✓ | ✓ | ✓ | ✓ |
| Psyb0t [78] | × | × | ✓ | ✓ | ✓ |
| Chuck Norris [78] | ✓ | × | ✓ | ✓ | × |
| Kaiten [71] | ✓ | ✓ | ✓ | ✓ | ✓ |
| Darlloz [48] | ✓ | ✓ | × | ✓ | × |
| BASHLITE [77] | ✓ | ✓ | ✓ | ✓ | × |
| XOR.DDoS [101] | ✓ | ✓ | ✓ | ✓ | ✓ |
| IRCTelnet [103] | × | ✓ | ✓ | ✓ | ✓ |
| LizKebab [102] | × | ✓ | ✓ | ✓ | × |
| Remaiten [100] | × | ✓ | ✓ | ✓ | ✓ |
| TheMoon [99] | ✓ | ✓ | ✓ | ✓ | ✓ |
| NyaDrop [104] | ✓ | ✓ | × | ✓ | ✓ |
| Hajime [151] | ✓ | ✓ | ✓ | ✓ | × |
| Amnesia [163] | × | × | × | ✓ | ✓ |
| BrickerBot [129] | × | ✓ | × | × | ✓ |
| PERSIRAI [146] | × | × | ✓ | ✓ | ✓ |
| IMEIJ.A [145] | × | × | ✓ | ✓ | × |
| DvrHelper [22] | × | ✓ | × | ✓ | × |
| Okiru [41] | × | × | ✓ | ✓ | × |

infection of these malware can be well covered by the infection process we identified.

Unfortunately, the leaked source code [161, 109] and reverse-engineering reports [71, 77] of IoT malware are still very limited. Prior studies on IoT malware infection process are also rare. What's worse, they either lack details of how the infection process work [109, 161, 71, 77] or dedicate on specific IoT malware [33, 73]. To fully understand how IoT malware infection proceeds, it is crucial to know in detail what commands are executed and in what sequence they are executed during each infection phase. This motivates us to seek a more comprehensive study over large-scale data that could contain a rich number of shell commands used for IoT malware infection.

#### 6.2.1.2 Large-scale Data Collection and Analysis

We further collect a large-scale dataset from VirusShare [12], which is a repository of malware samples to provide security researchers, incident responders, and forensic analysts with access to samples of live malicious code. The repository contains a large number of diverse malware samples, whose active time ranges from 2012 to 2020. In VirusShare, we find two types of files that can offer shell commands, which are needed in our comprehensive study. One type is the malicious

ELF (binary) file, which may invoke shell commands. The other type of file containing shell commands is the Linux shell script. More interestingly, after manually checking the shell scripts we extracted from VirusShare, we find that most of them are for infection purposes and can be aligned with the infection process we identified in § 6.2.1.1.

*1) Data Retrieval:* We collect *all* the files on VirusShare with timestamp spanning from 2012-06-15 to 2020-01-08. We leverage the `file` utility tool to extract ELF files. To extract shell scripts, we parse all downloaded files using `base64`, `file`, and `bashlex` [34], an open-sourced bash shell parser. We then query VirusTotal [156], a hub of more than 70 antivirus scanners and URL/domain blacklisting services, for each ELF file and shell script to make sure that they are reported as malicious. We consider the report as malicious if there is at least one scanner or service indicating it as malicious. As a result, we finally obtain 48,099 valid ELF files, out of which **40,318** are malicious ELF files, and 3,620 valid Linux shell scripts, out of which **3,439** are infection shell scripts.

*2) Shell Script Analysis:* We manually review all the shell scripts in our dataset to understand the logic of each script. In particular, we extract and get the statistics of the shell commands invoked by the scripts. We also refer to online manual pages [95, 39] to understand the capability of each shell command. As such, we confirm that all the shell scripts in our dataset can align with a part or all of the phases of IoT malware infection we have identified. An example of the shell commands invoked by an infection shell script is shown in Fig. 6.2 (upper). We can clearly recognize that there are five different shell commands being invoked in this shell script. According to the capabilities of those commands, we are able to align them with four phases (1, 3, 4, and 5) of the infection process sequentially.

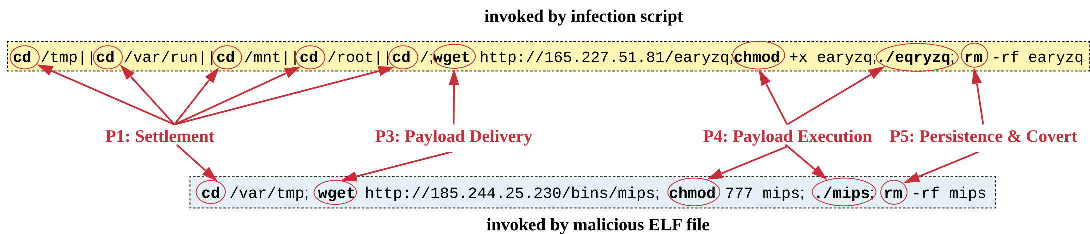



Figure 6.2: Examples of shell commands invoked by an infection shell script (upper) and a malicious ELF file (bottom) in our dataset.

*3) Malicious ELF File Analysis:* To extract the commands from malicious ELF files, we take a three-step approach. First, we leverage `radare2` [128] to disassemble each ELF file and extract the

Table 6.2: Top 20 commands invoked by malicious ELF files, infection scripts, and overall files in our dataset.

| ELF Files | Percentage | Scripts | Percentage | Overall | Percentage |
|---|---|---|---|---|---|
| **sh** | 18.56% | **cd** | 51.12% | **cd** | 33.89% |
| **cd** | 17.21% | **chmod** | 10.69% | **rm** | 10.33% |
| **rm** | 12.92% | **./** | 10.50% | **chmod** | 9.76% |
| **chmod** | 8.86% | **wget** | 7.74% | **sh** | 9.43% |
| **wget** | 6.31% | **rm** | 7.67% | **wget** | 7.01% |
| **tftp** | 5.91% | **cat** | 3.31% | **./** | 5.94% |
| **pkill** | 3.18% | **tftp** | 2.63% | **tftp** | 4.30% |
| **history** | 3.02% | **curl** | 1.40% | **cat** | 1.77% |
| **busybox** | 2.14% | **busybox** | 1.09% | **pkill** | 1.63% |
| **service** | 2.10% | **echo** | 1.02% | **busybox** | 1.62% |
| **ftpget** | 2.09% | **cp** | 0.37% | **history** | 1.55% |
| **./** | 1.51% | **chattr** | 0.37% | **ftpget** | 1.14% |
| **iptables** | 1.48% | **mv** | 0.23% | **service** | 1.07% |
| **sshd** | 1.02% | **ulimit** | 0.22% | **curl** | 1.00% |
| **perl** | 0.97% | **grep** | 0.17% | **iptables** | 0.76% |
| **su** | 0.69% | **touch** | 0.16% | **echo** | 0.75% |
| **cc** | 0.63% | **ftpget** | 0.16% | **sshd** | 0.52% |
| **curl** | 0.60% | **killall** | 0.13% | **perl** | 0.50% |
| **kill** | 0.56% | **md5sum** | 0.08% | **cp** | 0.40% |
| **killall** | 0.53% | **mkdir** | 0.08% | **su** | 0.35% |

strings from the disassembled code. This yields 32 million valid strings. Second, we review the strings from 800 different malicious ELF files manually and compare them with the commands extracted from infection scripts to identify patterns of command strings. Third, we automate the command extraction using these identified patterns. We finally find that 46.64% malicious ELF files invoke shell commands and 40.99% execute Linux shell scripts. In addition, command strings extracted from malicious ELF files are very similar with those in infection scripts. Fig. 6.2 demonstrates two examples extracted from infection scripts (upper) and malicious ELF files (bottom), respectively. They share the same infection phases but with slightly variants within each phase. Next, we compare the similarities of commands invoked by malicious ELF files and infection scripts.

*4) Command Distribution Comparison:* We compare the distribution of shell commands found in malicious ELF files and infection shell scripts. We find that the commands invoked by both types of files are quite concentrated. Table 6.2 lists the top-20 commands of both types of files in our dataset. We calculate the invocation percentage of each command by counting the times of each command being invoked divided by the total times of all commands being invoked. The percentages drop to 0.53%, 0.08%, and 0.35% for the 20th command in malicious ELF files, infection shell scripts, and overall files, respectively. In Table 6.2, commands shared by malicious ELF files and infection

scripts take 75.32% of all the command invocations in the overall dataset. This result implies that malicious ELF files and infection shell scripts invoke the same set of shell commands in most times. One major difference is that malicious ELF files invoke the `sh` command extensively but the infection scripts don't. We then look into how the `sh` is invoked by malicious ELF files and find that most of them are trying to execute some shell scripts. This means malicious ELF files in some cases may also try to execute a shell script to complete the infection process.

#### 6.2.1.3 Understanding IoT Malware Infection Process

Based on the analysis of malicious ELF files and infection scripts in the large-scale dataset, it turns out that each malware infection phase may invoke multiple shell commands and one shell command may also be invoked in multiple phases. It is the capability of a shell command that determines *when* and *how* the command is invoked in malware infection. To better understand when and how shell commands are used by each malware infection phase, we first introduce the notion of *infection capability*: an abstraction of a sort of shell commands that can achieve certain goals during the malware infection. Then we explain the malware infection process based upon the infection capabilities exploited by each infection phase. Focusing on the infection capability rather than each specific command makes our understanding more general. This generality allows future work to add new commands or remove outdated commands to keep the knowledge base up-to-date.

We abstract the shell commands in our dataset into 25 *infection capabilities*, each of which aims to achieve certain goals in malware infection process. For instance, lots of malware will download files from a server on the Internet in 3. There are a set of shell commands, such as `wget`, `tftp`, `curl`, and `git`, that can achieve this goal. Then we abstract these commands, which have the capability to download files from a remote server, as *Download* (**DW**) infection capability. Table 6.3 summarizes the infection capabilities we abstracted along with corresponding explanations. To further verify the infection capabilities we have abstracted, we thoroughly investigate 3 open-source IoT malware and 18 reverse-engineering reports of IoT malware listed in Table 6.1. We found that those infection capabilities can cover all the IoT malware we investigated.

We next provide more details about how infection capabilities are exploited in each phase of malware infection. We hope that the knowledge we compile and offer here can help the security community to better understand how the infection process works and will serve as a reference for future research focusing on the malware infection.

Table 6.3: A list of infection capabilities we abstracted and corresponding explanations.

| Capabilities | Abbr. | Explanation |
|---|---|---|
| Change Permission | **CH** | Change the permission of files |
| Remove History | **RM** | Remove activity history to evade forensic |
| Disable Security | **DS** | Disable security mechanisms |
| Download | **DW** | Download files from the Internet |
| Find Place | **FP** | Try to find a working place |
| Copy File | **CP** | Copy files |
| Create and Write | **CW** | Create and write to new files |
| Decompress | **DCP** | Decompress files |
| Decode | **DCD** | Decode from encoded files |
| Compile Code | **CC** | Compile source code |
| Process Text | **PT** | Search, cut, sort texts |
| Kill Process | **KILL** | Kill processes |
| Exclude Others | **EXO** | Avoid others to infect or login |
| Network Probe | **NP** | Probe internal or external networks |
| Implant Backdoor | **IBD** | Launch a deamon to enable access later |
| Execute | **EXE** | Execute files |
| Collect Information | **CI** | Collect information from the system |
| Manage System | **MSYS** | Change users and environment variables |
| Manage Software | **MSOFT** | Update or install software |
| Manage Resource | **MRES** | Set/reset resource hard and soft limit |
| Get Time | **GT** | Get timestamp of files |
| Change Time | **CT** | Change timestamp of files |
| Programming | **PR** | programming commands, e.g., break; continue |
| Agnostic | **AGN** | Any commands do not align to the above |
| Unrecognized | **UN** | Any commands not including in our base |

Table 6.4 lists the 5 malware infection phases and the infection capabilities exploited by each phase along with exploitation percentages. In the table, we also list the top commands (up to 5) that are abstracted as each infection capability. For a complete list of shell commands abstracted as each infection capability, please refer to Table **??** in Appendix **??**. We further visualize the relationships between infection phases and infection capabilities in Fig. **??** in Appendix **??**. There are three special infection capabilities:

- *Unrecognized* (**UN**). We use *unrecognized* infection capability to abstract any new commands that are not included in our dataset. Thus, this infection capability is not exploited in any malware infection phase.
- *Agnostic* (**AGN**). We use *agnostic* infection capability to abstract the commands that may be executed in any phases whenever it is necessary. Among the examples are `sleep` (wait for certain time), `pwd` (print working directory), `clear` (clear screen contents), etc.

Table 6.4: Infection capabilities exploited in each phase, exploitation percentage, and top commands (up to 5).

| Phase | Capabilities | Percentage | Top Commands |
|---|---|---|---|
| **P1** | **CW** | 2.10% | echo open printf mkfifo |
| | **AGN** | 0.70% | lp md5sum |
| | **FP** | 89.49% | cd ls |
| | **CP** | 6.77% | cat cp mv read head |
| | **PR** | 0.67% | enable exit set unset test |
| **P2** | **CI** | 26.91% | ps id du df file |
| | **PT** | 31.61% | tr sed grep wc awk |
| | **AGN** | 10.74% | lp md5sum yes clear sleep |
| | **NP** | 0.30% | nmap zmap dig |
| | **PR** | 10.31% | enable exit set unset test |
| | **MRES** | 4.45% | ulimit |
| | **DS** | 0.08% | ufw, accton |
| **P3** | **AGN** | 1.55% | lp md5sum yes clear sleep |
| | **DCD** | 0.15% | base64 |
| | **CP** | 14.94% | cat cp mv read head |
| | **DW** | 78.38% | wget tftp ftpget curl ftp |
| | **DCP** | 0.25% | tar gzip unzip gunzip |
| | **CC** | 2.12% | cc gcc make ldconfig |
| | **PR** | 1.49% | enable exit set unset test |
| **P4** | **AGN** | 0.85% | lp md5sum yes clear sleep |
| | **CH** | 31.57% | chmod chattr chown umask usermod |
| | **EXE** | 58.77% | sh ./ busybox service perl |
| | **MSYS** | 3.95% | iptables su sudo reboot defaults |
| | **PR** | 0.81% | enable exit set unset test |
| | **MRES** | 0.35% | ulimit |
| | **MSOFT** | 1.46% | mkdir ln yum mktemp export |
| **P5** | **PT** | 4.67% | tr sed grep wc awk |
| | **AGN** | 1.59% | lp md5sum yes clear sleep |
| | **MSOFT** | 2.74% | mkdir ln yum mktemp export |
| | **IBD** | 3.12% | sshd httpd squid |
| | **GT** | 0.19% | date |
| | **CT** | 0.50% | touch |
| | **EXO** | 0.86% | passwd chpasswd iptables-restore |
| | **PR** | 1.52% | enable exit set unset test |
| | **DS** | 0.01% | ufw accton |
| | **KILL** | 13.37% | pkill killall kill |
| | **RM** | 70.71% | rm history |

- *Programming* (**PR**). We use *Programming* infection capability to abstract the commands that are used for general programming purpose. Among the examples are `break`, `continue`, `local`, `set`, etc. Commands with this infection capability may also be used in any phases whenever it is necessary.

Next, we will go through the 5 phases of the infection process and explain how each infection capability is exploited in each phase.

*1) Settlement:* The initial step of the infection process is to find a working place as a playground where succeeding commands can be executed. Usually the default working place is the home directory of the login user. Other than the default home directory of the login user, we also find many infection attempts seek different working places. Some of the most commonly trials are `/tmp`, `/var`, `/etc`, `/run`, and `/`.

There are three infection capabilities being exploited in this phase. First, malware may exploit the *find place* (**FP**) infection capability to check the existence of a path, for example, using `cd` and `ls` commands. As is shown in Table 6.4, **FP** is the most widely exploited infection capability taking 89.49% command invocations. Second, malware can exploit the *copy file* (**CP**) infection capability to copy or move a file to a certain place to check whether a place is writable. Finally, malware may also exploit the *create and write* (**CW**) infection capability to create and write to a file at a specific path so as to determine the write permission of that place. It is worth noting that malware finds the proper working place via a trial and error manner, which is quite suspicious because a legitimate administrator should have the idea where the best working place is.

*2) Environment Preparation:* Once malware gains a foothold of a system, the next step is to prepare the target environment so that succeeding actions can proceed. During this phase, 3 infection capabilities are exploited, involving *collect information* (**CI**), *process text* (**PT**), and *network probe* (**NP**). In particular, the **CI** and **NP** are usually followed by **PT**, which achieves the goal of trimming, searching, sorting, or counting on the text information. Table 6.4 shows that **CI** and **PT** dominate the infection capabilities exploited in this phase, taking 28.19% and 33.11%, respectively. We further categorize the information collected during this phase into 7 categories:

- *Operating System.* This information includes the type and version of the operating system, system variables, and system up time.

- *File System.* This information includes the type of file system, existence of certain paths, and

permissions of certain paths and files.

- *Processes.* This information includes process identification (PID), files opened by a process, running services, and how much processing units available in the system.
- *User Groups.* This information includes user id and user group id. Sometimes, if the command is not running with the root privilege, malware will terminate the infection.
- *Networking.* This information includes IP address, MAC address, host name, reachability of another network, opening ports, and opening sockets in the system.
- *Hardware.* This information includes CPU architecture, available memory and disk volume, and peripherals.
- *Security.* This information includes availability of `iptables` and `ufw`.

*3) Payload Delivery:* After the environment preparation, malware seeks ways to deliver malicious payloads to the target system to make the system suitable for various monetization purposes. There are two major steps to deliver malicious payloads to the target system. First, malware downloads some files from a server on the Internet. Then, malware releases or makes the payloads from the downloaded files. To achieve those two steps, 5 infection capabilities are exploited in this phase.

To download files from a remote server, malware exploits the *download* (**DW**) infection capability, which takes 78.38% of command invocations in this phase as listed in Table 6.4. Furthermore, we find that there are three types of files downloaded to the target system, including *binaries*, *scripts*, and *source codes*. To release or make the payloads, 4 infection capabilities are exploited, including **CP**, *decompress* (**DCP**), *decode* (**DCD**), and *compile code* (**CC**). In this phase, **CP** is the second commonly used (14.94%) infection capability that allows malware to copy the downloaded files to proper places. In addition, we find three ways that malware releases or makes the payloads from downloaded files.

- *Decompressing.* Compression is a method that can bypass naive detection approaches. Commonly seen compressed files include `tar`, `gz`, `zip`, and `gzip`. Malware exploits **DCP** to release the payloads from compressed files.

- *Decoding.* Another more sophisticated technique is encoding the payloads, such that they can bypass some of the detection approaches. The encoding/decoding tool we find in our dataset is `base64`. Malware exploits **DCD** to release payloads from encoded files.

- *Compiling.* We find a few malware downloads source code files and call compiling tools such as `gcc` and `ldconfig` on the target system to make the payload. We abstract those commands as **CC**, which is exploited by malware to make payloads from the source code files.

*4) Payload Execution:* The core of the infection process is to execute malicious payloads that will serve various monetization purposes. In the Payload Execution phase, malware exploits 5 infection capabilities, including *manage software* (**MSOFT**), *manage system* (**MSYS**), *manage resource* (**MRES**), *change permission* (**CH**), and *execute* (**EXE**).

Before malware starts to execute malicious payload, it may prepare the environment by exploiting **MSOFT** to install dependencies, exploiting **MSYS** to create folders and user accounts, and exploiting **MRES** to raise the limit of the resources that a process can utilize. Once the malicious payloads are available and the environments are ready, malware exploits **CH** to change the permission of the malicious payload to make it executable or belonging to specific users. We observe that 31.57% command invocations exploit **CH** as listed in Table 6.4. Malware finally exploits **EXE** to execute malicious payloads in the target system. There are 58.77% command invocations that exploit **EXE**. The methods to execute malicious payloads are categorized into four categories:

- *Direct Execution.* The most direct way is to run the executable via the `./` syntax. This method applies to malicious payloads that can be executed directly from a shell.

- *Interpreters.* Some malicious payloads are loaded and run by interpreters, such as `perl`, `Python`, `php`, `Go`, etc. This method requires that proper interpreters are installed and available in the system.

- *Background Utilities.* Some utilities are designed to allow programs to be executed in background or to be launched from a remote server. Those utilities include `nohup`, `screen`, `sshpass`, `ssh`, and `nc`.

- *Scheduling Utilities.* There is another body of tools that may be used to load and run malicious payloads in a more stealthy way. Those utilities are designed to schedule a program to run in certain time frames. Those utilities are `service`, `nice`, and `crontab`.

*5) Persistence & Covert:* In this phase, malware tries to persist in the system and hide its footprint.

For persistence, malware exploits five infection capabilities, including *implant backdoor* (**IBD**), *exclude others* (**EXO**), *kill process* (**KILL**), *disable security* (**DS**), and **MSOFT**. There are four strategies commonly used by malware to persist. First, malware may exploit **IBD** to run legitimate services, such as `httpd` and `sshd`, as daemons in the background. Malware may exploit **MSOFT** to install those services if they are not installed. Second, malware may exploit the **EXO** to prevent other malware from accessing the device. A typical operation is to change the credentials using `chpasswd` and `passwd`. Third, malware may exploit **KILL** to terminate competitors and benign processes to make more system resources available. Notably, there are 13.37% command invocations are for this purpose as listed in Table 6.4. Finally, some malware may exploit **DS** to disable security mechanisms, including `iptables` and `ufw`.

To hide its footprint, malware exploits 3 infection capabilities, *get time* (**GT**), *change time* (**CT**), and *remove history* (**RM**). There are two strategies used by malware to hide their footprints. The most widely used strategy is to remove the history by exploiting **RM**, which takes 70.71% of all command invocations in this phase. Malware may remove the payloads after execution. We also observe that malware may remove the command history in the system. Another strategy is to modify the time stamps of certain files. Those files are under `/bin`, `/usr/bin`, and `/sbin`. The reason is that some malware may access or modify those files. Changing the time stamps of those files will keep this kind of operations undiscovered.

### 6.2.2 Modeling IoT Malware Infection Process

In order to use the knowledge of malware infection that we have built for detecting malware infection, we need to formalize the infection process with a model. However, shell commands extracted from malicious ELF files lose the execution logic information, and thus we cannot obtain the exact invocation sequence of the extracted commands [1]. Therefore, we leverage the infection scripts, which contain complete execution logic, from our dataset to model the IoT malware infection process.

Since our model will be used for malware infection detection, a baseline is required to

[1] We also tried to run malicious ELF files in existing Linux-based sandboxes [43, 96, 72]. Unfortunately, they could not be correctly run on those sandboxes.

distinguish infection activities from benign activities. To determine the baseline between infection and benign activities, we collect benign shell scripts from GitHub [65] and use them to assist our modeling. In particular, we develop a GitHub crawler, which collects 244,162 repositories that have more than 50 stars as of 2020-01-22. Since our focus is Linux shell commands, we reduce our scope to 1,987 repositories that are labeled with the shell scripting language. We finally extract 9,341 shell scripts and further verify – through `base64`, `file`, `bashlex`, and VirusTotal – that 3,890 are valid and benign shell scripts.

An overview of our modeling approach is illustrated in Fig. 6.3. The output of our modeling is a weighted infection state machine (WISM) that models the malware infection. Our modeling approach consists of 3 major steps. First, we develop our own tool to generate command flow graph (CFG) for all the infection scripts and benign scripts in our dataset. Second, we build an infection state machine (ISM) based upon the CFGs of all the infection scripts. Finally, we conduct a correlation analysis based on the CFGs of infection scripts and benign scripts. The correlation analysis tracks the command flows in all CFGs and assigns a weight to each state transition in the ISM. The weights are maximized for infection scripts while minimized for benign scripts. As a result, we obtain a WISM.

#### 6.2.2.1 Generating Command Flow Graphs

The CFG is a representation, using graph notation, of all paths containing shell commands in sequence that might be traversed through a shell script during its execution. In our generated CFGs, each node represents a shell command and each directed edge represents a transfer from one command to another one. Unfortunately, we cannot find any mature tools that can be used to generate CFGs for Linux shell scripts. We develop our own tool to generate the CFGs based on `Bashlex` [34], which is an open-sourced parser for bash scripts.

In this work, we generate a CFG for each individual infection or benign script. Fig. 6.4 illustrates two infection scripts (script-a and script-b) and their corresponding CFGs. Each CFG is stored as a file using the `networkx` Python library. Those CFGs will be referenced when we build ISM and WISM.

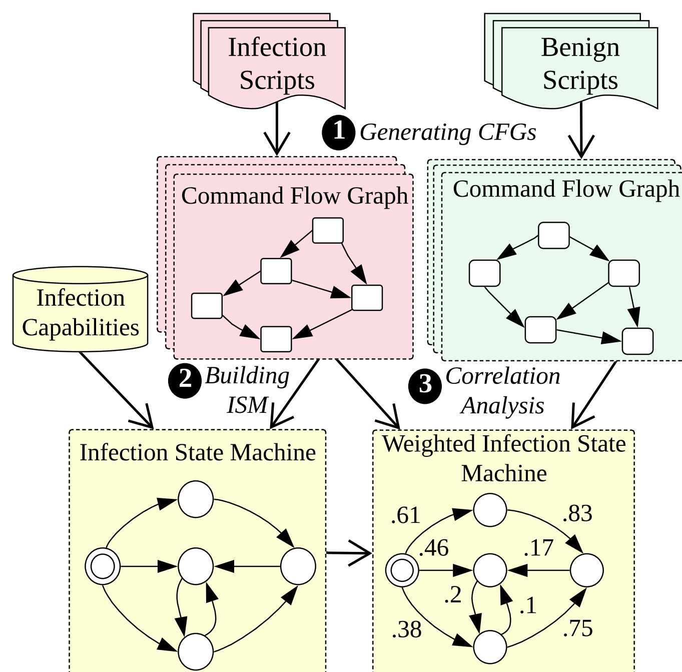



Figure 6.3: Modeling approach overview. Firstly, we generate the CFGs of infection and benign scripts in our dataset (1). Then we build an ISM that represent the behavior patterns of malware infection (2). After that, we assign weights to the ISM through a correlation analysis (3). The generated WISM works as a general model of malware infection.

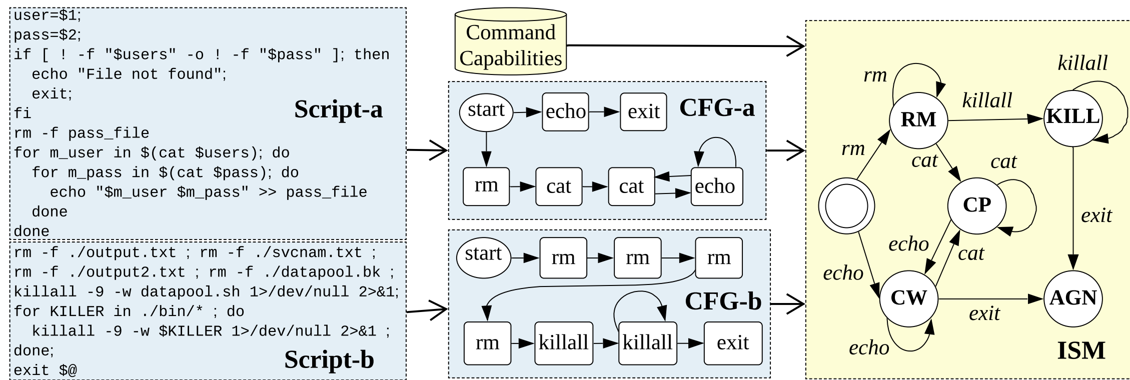



Figure 6.4: Building the ISM from shell commands in infection scripts. Refer to Table 6.3 for abbreviations of the capabilities in the ISM.

#### 6.2.2.2 Building Infection State Machine

We use an ISM to represent the behavior patterns of malware infection. We formally define our ISM as a 5-tuple $(\Sigma, S, s_0, \Delta, F)$ where:

- $\Sigma$ is the set of all the shell commands that we cover;
- $S$ is the set of states, each of which is mapped to an infection capability in Table 6.3;
- $s_0$ is the initial state, which also belongs to $S$ but is not mapped to any infection capability;
- $\Delta$ is the state-transition function: $\Delta : S \times \Sigma \rightarrow S$; and
- $F$ is the set of final states.

We only build a single ISM from all the infection scripts. As an example, Fig. 6.4 depicts the CFGs of two infection scripts and the ISM built from those two scripts.

In the ISM, we consider each node as a state where a specific infection capability has been exploited during the infection. Each state in our ISM is mapped to an infection capability except for the initial state, $s_0$. For example, a state that maps to **RM** means that if the infection goes into this state, the **RM** infection capability has been exploited. Therefore, some shell commands, such as `rm` and `history`, must be invoked. We associate a directed edge with a command, which is abstracted as certain infection capability. For example, a directed edge coming into the **RM** state is associated with the `rm` command in Fig. 6.4 because in the infection scripts we observe the **RM** infection capability is exploited by invoking the `rm` command.

**Algorithm 4:** Building ISM from CFGs

**Input:** $C$: a list of all CFGs;
$Trans$: translation table mapping commands to capabilities;
**Output:** $ISM$: the built ISM;

```
1 ISM = {} /* a set containing the edges along with the nodes of ISM */
2 foreach c in C do
3     foreach edge in c do
4         if Trans[edge] ∉ ISM then
5             ISM = ISM ∪ {Trans[edge]}
6 return ISM
```

To automatically build the ISM from all the infection scripts, we develop a tool that takes the CFGs as inputs and yields an ISM. The algorithm we used is presented in Algorithm 4. The *Trans* is a hash function that translates an edge in the CFG to an edge in the ISM. The core of the hash function is the infection capabilities we identified. Formally, the edge in the CFG is defined as a tuple $(src, dst)$ where $src$ and $dst$ are two consecutive commands. The edge in the ISM is defined as a 3-tuple $(s_1, \sigma, s_2)$ where $s_1 \in S$ and $s_2 \in S$ are two neighboring states and $\sigma \in \Sigma$ is the label of the transition between the two states. *Trans* is a function: $Trans : (src, dst) \rightarrow (s_1, \sigma, s_2)$. By using the *Trans* function, we can translate an edge in the CFG into an edge in the ISM. In *lines 4 and 5* of Algorithm 4, we add all edges (which are translated from the CFGs) that are not in the ISM into the ISM. This procedure is applied to the CFG of each infection script (*line 2*). Each state in the ISM is mapped to an infection capability and each edge in the ISM is associated with a command. A state transition in the ISM means an infection capability is exploited. To trace what infection capabilities have been exploited and in what sequence they are exploited, a sequence of state transitions must be considered.

#### 6.2.2.3 Correlation Analysis

*1) Key Idea:* Our correlation analysis tracks a sequence of state transitions in the ISM to determine whether there is an infection. The key idea is to assign each state transition in the ISM with a weight and maintain a counter for the weights over a temporal window. Once the counter exceeds a threshold, an infection is detected. To determine the weight for each state transition, we employ a hill-climbing algorithm [37].

*2) Risk Score:* Let's denote a WISM as $\theta$, a serials of state transitions that are triggered by a sequence of $N$ commands as $T = (t_1, t_2, ...t_{N-1})$, and $0 \leq \theta_{t_i} \leq 1$ as the weight that will be

assigned to the state transition $t_i$ in WISM. Then, the risk score of a sequence of $N$ commands is defined as

$$R(T, \theta) = \frac{1}{N} \sum_{i=1}^{N} \theta_{t_i} \tag{6.1}$$

With the above definition, we can calculate a risk score given a WISM and a sequence of commands.

*3) Assigning Weights:* To obtain a WISM that best models the infection process, we *minimize* the risk score for a sequence of commands invoked by benign scripts and *maximize* the risk score for a sequence of commands invoked by infection scripts. Therefore, we can formulate the weight assignment as an optimization problem that maximizes the following objective function:

$$f(\mathbf{M}, \mathbf{B}, \theta) = \sum R(T_i^{\mathbf{M}}, \theta) - \sum R(T_j^{\mathbf{B}}, \theta) \tag{6.2}$$

where $\mathbf{M}$ is a set of sequences of commands invoked by infection scripts, $T_i^{\mathbf{M}}$ is a serials of state transition triggered by the *ith* sequence of commands invoked by infection scripts, $\mathbf{B}$ is a set of sequences of commands invoked by benign scripts, $T_j^{\mathbf{B}}$ is a serials of state transitions triggered by the *jth* sequence of commands invoked by benign scripts.

We then employ a hill-climbing algorithm to solve the optimization problem. The hill-climbing algorithm relies on a feedback loop to gradually improve the quality of weights in the WISM. For each iteration, the algorithm adjusts all the weights of the WISM with a small amount (0.01 in our case). The initial weights of each state transition are set to the frequency of the corresponding consecutive commands in the infection scripts. Over a sufficiently large number of iterations (100,000 in our case), we observe the convergence and obtain the weights for our WISM.

*4) Command Sequences:* The command sequences used by the hill-climbing algorithm are generated from our infection scripts and benign scripts. However, it is infeasible to enumerate all possible paths of a CFG because of the path explosion issue. To address the path explosion issue, we relax our model by ignoring the paths that cover multiple iterations of the loops. First, we utilize the *simple_paths* methods in the `networkx` library to extract the paths from the entry point to each endpoint in the CFG. Then, for each loop, we add extra paths that involve only two iterations of the loop. We finally extract 4 million sequences of commands from all the CFGs. We then use these command sequences for the correlation analysis.

### 6.2.3 Infection Detector Development

We develop an infection detector, SOTER, based on the infection model we build. The design objective of SOTER is *lightweight*, so that it can be deployed in resource-constrained Linux-based IoT devices, such as a D-Link IP camera [44] that only equips with 32MB random access memory (RAM) and 4MB flash memory. Specifically, SOTER should meet the following requirements:

- **Trivial Modification.** Since the operating systems for IoT devices are often customized to fit a variety of IoT device specifications, SOTER should add as little code as possible to maximize its compatibility. In addition, SOTER should not depend on external tools or security frameworks, such as *KProbe*, *inotify*, and *Linux Security Module (LSM)*, because those dependencies are not always supported by IoT devices.

- **Storage Efficiency.** Unlike commodity PCs and servers, IoT devices usually equip with limited storage. Even worse, the amount of free memory that can be used for security purposes is extremely small. For instance, the D-link DCS-932L IP camera has 4MB flash memory installed. But less than 400KB free flash memory is available after installing some standard packages, such as light sensor and MJPG streaming. Therefore, SOTER should be able to be deployed in such a limited space.

- **Computation Efficiency.** Due to the limited computation capability, IoT devices are not suitable for computational-intensive tasks. SOTER should utilize as fewer computation resources as possible to accomplish the detection task.

#### 6.2.3.1 Design Choices

To achieve trivial modification, we design SOTER as two logically separated parts, *hook* and *classifier*. The *hook*, which is embedded into the source code of the Linux kernel, aims to capture the commands that are executed. We modify the `execve` system call to intercept its parameters. From the parameters, we can determine which commands are executed and what options are passed to the commands. We skip the shell built-in commands, such as `break`, `continue`, `set`, and `declare`, because it is difficult to trace those built-in commands efficiently and they are for general programming purposes. Therefore, we only need to add a callback function in the `SYSCALL_DEFINE3` function in `fs/exec.c`. To raise an alert, we invoke the `printk` function to emit alerts to the log system (e.g.,

Table 6.5: A breakdown of space consumed by each asset of the infection detector.

| **Assets** | **Size (bytes)** | **Description** |
|---|---|---|
| command.h | 108 | Encode all commands as enum type. |
| capability.h | 700 | Encode all capabilities as enum type. |
| hash.h | 2,275 | Mapping between commands and capabilities. |
| transitions.h | 10,512 | Encode states and edges of IWSM as an array. |
| Total | 13,595 | Space consumed by all above data structures |

`/var/log/syslog`). In order to maximize the compatibility, we implement the *classifier* as a Loadable Kernel Module (LKM), which can be modified, compiled, and installed without recompiling the operating system kernel. To update *classifier*, one only needs to modify and compile the LKM to generate a *.ko* file. Then the *.ko* file can be distributed to Linux-based IoT devices and deployed via `insmod`. Neither a bootstrap nor a reboot is required to deploy *classifier*, which includes all the detection logic of SOTER.

To achieve storage efficiency, we encode the WISM, infection capabilities, and corresponding commands as C arrays in *.h* files. To further reduce the size of SOTER, transitions in WISM with 0 weights are not encoded. As such, SOTER finally consumes less than 14KB to store the WISM, infection capabilities, and corresponding commands. A breakdown of the assets and corresponding sizes are listed in Table 6.5. To minimize the run-time footprint, we neither use any memory from the heap (e.g., allocated via `kmalloc`) nor employ recursive procedures in SOTER. The memory allocated to SOTER only includes those hard-coded C arrays. An extra array is used to maintain a temporal window of command sequence with a length of 34. In summary, the estimated space complexity of SOTER is $O(1)$.

To reduce computational overhead, the detection logic of SOTER should run in the kernel space as an LKM, which avoids context switching between the user space and the kernel space. The major challenge of implementing the detection logic in the kernel space is that not all CPU architecture supports floating-point calculation in the kernel space. Even worse, a significant body of CPU architecture does not support 64-bit multiplication. In order to allow SOTER to run on as many CPU architectures as possible while to avoid introducing too much implementation complexity, we make a trade-off between the precision and the complexity. We scale all the weights – which are floating-point numbers between 0 and 1 – of the WISM to integer numbers that are between 0 to $2^{32}$. This transformation loses some precision of the weights, but it allows the weights to be presented as integers that can be multiplied in the kernel space.

#### 6.2.3.2 Implementation

SOTER is implemented and runs completely in the kernel space and detects malware infection in real-time. Fig. 6.5 provides an overview of SOTER. The *hook* is implemented as a C function pointer that points the entry point of the *classifier*. The *classifier* consists of the WISM, a threshold tester, and an alert function. Every time a command triggers the *hook* through `execve` system call, SOTER will recalculate the risk score of commands within a temporal window based on WISM. If the risk score exceeds a threshold, an alert is raised.

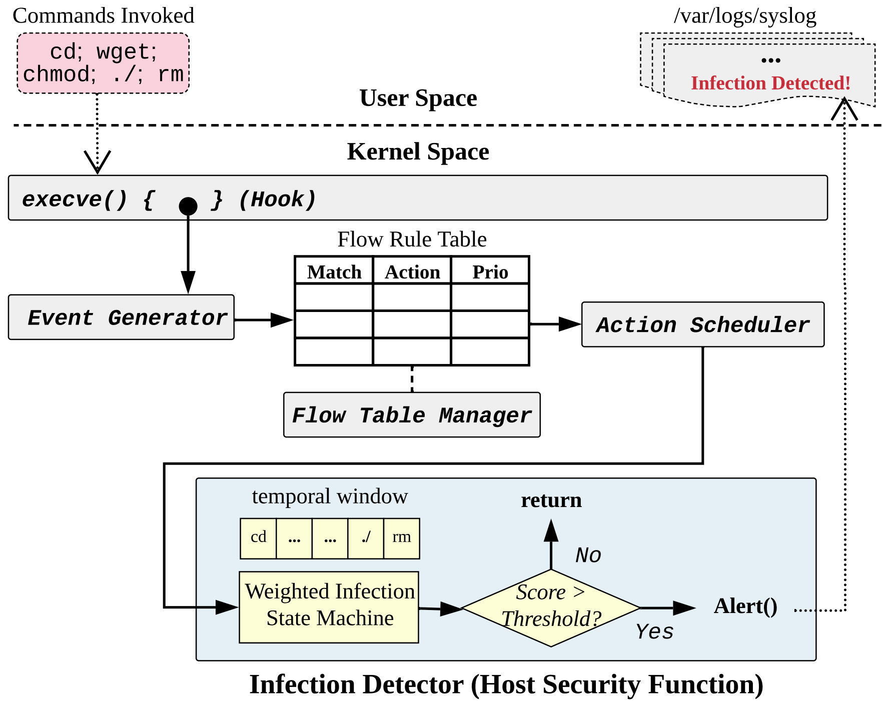



Figure 6.5: Overview of SOTER implementation.

There are two parameters that need to be defined for SOTER: *i)* the threshold; and *ii)* the temporal window size. When we decide the threshold, we make use of infection scripts and benign scripts in our dataset. For each script, we compute the risk scores of all the command sequences in that script. Then, we choose the highest risk score as the risk score of that script. After that, we compute the Cumulative Distribution Function (CDF) of risk scores for both infection scripts and benign scripts. The resulted CDF is shown in Fig. 6.6. By observing the CDFs, we estimate a threshold that satisfies a certain level of true alert rate while maintaining a relatively low false alert rate. For our implementation, we choose 0.67 as the threshold to achieve a reasonably high

true alert rate while remaining relatively low false alert rate. To determine the temporal window size, we count the length of all the command sequences we extracted from our infection samples. We observe that all the command sequences are shorter than 34. As a result, we decide to use 34 as the maximal length for SOTER. When SOTER is running, a 34-length window will slide over the command sequence. This parameter is adjustable by reloading SOTER.

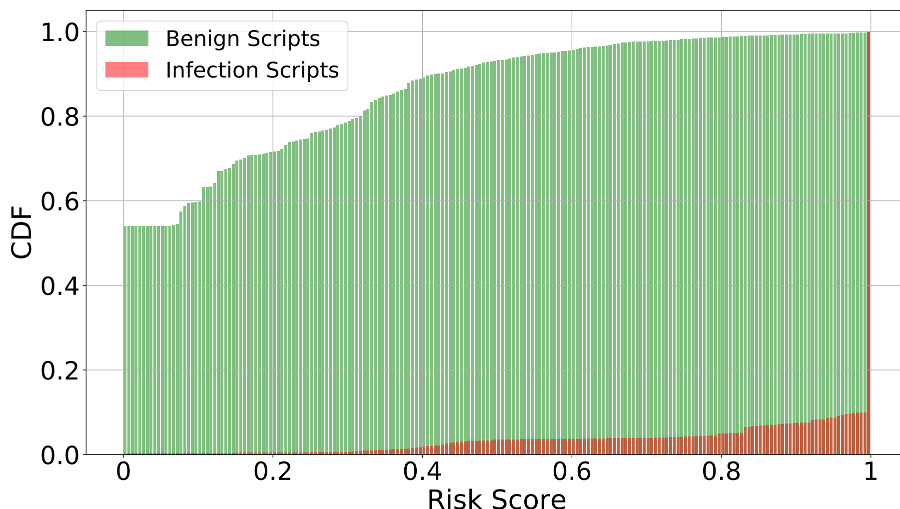



Figure 6.6: CDFs of risk scores of all benign scripts and all infection scripts.

### 6.2.4 Evaluation

In this section, we evaluate SOTER and our infection detection modeling from following aspects:

- **Effectiveness of SOTER.** We deployed a large scale of software IoT devices as honeypots across the globe and tested SOTER in those software IoT devices. (§ 6.2.5)
- **Generalization of our modeling approach.** We evaluated how general our modeling approach is by testing the trained model with the samples that have *not* been used for training and investigated the trade-offs between false positive and false negative. (§ 6.2.6)
- **Performance overhead of SOTER.** We measured the CPU and memory usage of SOTER on three types of physical IoT devices, which are representatives of *low-end*, *mid-end*, and *high-end* IoT platforms (§ 6.2.7).

### 6.2.5 Effectiveness Evaluation of SOTER

#### 6.2.5.1 Online Deployment

In order to evaluate the effectiveness of SOTER in realistic environment, we would like to run SOTER in IoT devices at scale and observe the detection performance of SOTER for infections in the wild. Unfortunately, it turns out deploying hardware IoT devices in large scale is challenging due to two major reasons. First, the Internet access fee for IoT devices is expensive because each individual device must be assigned with a public IP to attract infections. Second, ensuring the reliability of low-level infrastructure introduces high maintenance overhead, such as periodically checking the underlying Internet connections, power supply, and hardware devices.

As an alternative, we turn to software (virtual) IoT devices rather than hardware IoT devices for large-scale deployment. A body of work [122, 45, 62] has proved that software IoT devices are feasible and cost-efficient ways for IoT device deployment at scale. For example, HoneyCloud [45] demonstrates how to deploy high-fidelity software IoT devices on public clouds across the globe in large scale.

To prevent attackers from identifying our software IoT devices as honeypots, we enhanced the fidelity of software IoT devices from three aspects. First, since we use QEMU emulator to provision the software IoT devices, we specifically tuned the CPU and memory capabilities to best match that of real IoT devices, the Raspberry Pi in our experiments. Second, we used OpenWrt, one of the most popular Linux-based operating systems for IoT devices, in our software IoT devices. Furthermore, we forge `/proc/cpuinfo` to make it look like a commercial CPU used by real IoT devices. Third, we enabled the `httpd` service and installed the MJPGE streaming [112] and Motion package [115] in the software IoT devices to emulate essential functionalities of IP cameras. To preserve the detection results and the logs necessary for our analysis, we employed the Rsyslog [133] utility to transfer and save all the detection results and logs from the software IoT devices to a remote log server in real time. To prevent our software IoT devices from being used by attackers to launch attacks or infect more IoT devices, we ran Snort on the host virtual machines (outside of software IoT devices) to monitor inbound and outbound traffic. We setup forwarding rules to block dangerous outbound traffic, which is determined according to the Snort log.

To evaluate the effectiveness of SOTER, we deployed software IoT devices as honeypots at scale across the globe following the guidance introduced in [45] and installed SOTER in those software

Table 6.6: Software IoT device deployment statistics.

| Region | Site | Deployed Device | Valid Login |
|---|---|---|---|
| **North America** | Ashburn | 9 | 3,866 |
| | New York | 6 | 2,603 |
| | Los Angeles | 6 | 2,613 |
| | Clemson | 6 | 875 |
| | Utah | 6 | 91 |
| | Wisconsin | 6 | 697 |
| | Des Moines | 1 | 1,834 |
| | Seattle | 1 | 578 |
| | Chicago | 3 | 1,619 |
| | Toronto | 4 | 4,936 |
| | Washington | 1 | 1,650 |
| **Europe** | Helsinki | 3 | 2,981 |
| | Brussels | 3 | 1,777 |
| | London | 4 | 3,612 |
| | Berlin | 3 | 2,331 |
| | Amsterdam | 4 | 3,367 |
| | Zurich | 3 | 2,486 |
| | Paris | 1 | 537 |
| | Dublin | 1 | 2,362 |
| **Asia** | Tokyo | 4 | 3,397 |
| | Hong Kong | 4 | 5,027 |
| | Taipei | 3 | 2,226 |
| | Seoul | 1 | 1,982 |
| | Singapore | 4 | 5,247 |
| **Australia** | Sydney | 18 | 7,666 |
| **South America** | Sao Paulo | 17 | 7,504 |
| **Total** | | 122 | 73,864 |

IoT devices. As listed in Table 6.6, we deployed 122 software IoT devices on 4 public clouds – Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, and CloudLab¸ – distributing at 26 different sites (shown in Fig. 6.7) for 30 days. All the software IoT devices attract 73,864 successful logins.

#### 6.2.5.2 Real-time Detection Results

Among all successful logins, 59,506 execute at least one Linux command after login thus are considered as valid logins. To further verify that a valid login is actually a valid infection process, we parsed the logs generated by the software IoT devices. We found that not all valid logins are followed by a valid infection process because those logins execute files that do not exist in the system. There are two main reasons resulting in this phenomenon. The first reason is that the malware is running incorrectly on our software IoT devices due to wrong assumptions of the target system (e.g., malware

Figure 6.7: Geo-distribution of our deployed software IoT Devices.

authors may assume certain configurations are made and certain utilities are available). The second reason is that since we rebooted and reset our software IoT devices periodically, the malware thinks it has already infected the target thus is trying to execute files it has downloaded before. Therefore, we excluded cases where a file is executed before it is downloaded.

Table 6.7 lists the detecting results of SOTER over the valid infections. We observed 55,949 valid infections, among which SOTER raises 55,155 alerts. Since those software IoT devices are deployed as honeypots, all access to those devices is considered illegitimate. As a result, the FNR is 1.41% and TPR is 98.58%. We further investigated the false negative samples and found some new infection patterns that are not in our dataset[2]. For example, we observed some malware samples download malicious payloads and execute the same payloads for multiple times. Between two consecutive executions, they invoke `wget` to download more files. Finally, they remove all downloaded files. We thought the consecutively downloaded files may convey fragmented information that will be used by the malicious payloads. Among those false negative samples, we also found incomplete infections. For example, we found some malware samples login to invoke `ls`, `cat`, or `uname`, and then logout without invoking other shell commands. We thought this happens because the malware samples check the system information and may find that the system is not their target.

In addition, we explored how many unique infections have been detected by SOTER by looking at the risk score of each infection alert. We observed 873 different risk scores in our alerts, which means SOTER detects 873 unique infections. For a false negative sample, since the detector

[2]We will integrate these new infection patterns into our infection model. A more detailed discussion is in § **??**.

Table 6.7: Real-time detection results of SOTER. FN: False Negative; FNR: False Negative Rate; TPR: True Positive Rate.

| | **Total** | **Alert** | **FN** | **FNR** | **TPR** |
|---|---|---|---|---|---|
| **Valid Infections** | 55,949 | 55,155 | 794 | 1.42% | 98.58% |
| **Unique Infections** | 917 | 873 | 44 | 4.80% | 95.20% |

Table 6.8: Details of each sample set used in our evaluation.

| **Sample Sets** | **Volume** | **Description** |
|---|---|---|
| training_b | 3,112 | benign scripts for weight and threshold assign. |
| training_m | 2,751 | infection scripts for weight and threshold assign. |
| testing_b | 778 | unseen benign scripts to test our model |
| testing_m | 688 | unseen infection scripts to test our model |
| Total_b | 3,890 | all benign scripts use for the evaluation |
| Total_m | 3,439 | all infection scripts used for the evaluation |

does not raise an alert, we considered the highest risk score that it can reach instead. As a result, we observed 44 different risk scores. SOTER achieves 4.80% FNR or 95.20% TPR w.r.t. unique infections as shown in Table 6.7.

### 6.2.6 Generalization Evaluation of Modeling Approach

In this evaluation, we evaluated how general our modeling approach is by testing the trained model against *unseen* samples. In particular, we split the shell scripts in our dataset randomly into *training* set (80%) and *testing* set (20%) and made sure samples in the *testing* set do not appear in the *training* set. During the correlation analysis and threshold determination (a.k.a. training process), only the *training* set is used and the *testing* set always remains unseen to the detection model. During the testing process, the *testing* set is used to test the prediction performance of the trained model. Table 6.8 lists the details of all the sets we used in this evaluation. The *total_b* and *total_m* are sets of benign and infection scripts, respectively. The *training_b* and *training_m* are sets of benign and infection scripts used for training. The *testing_b* and *testing_m* are sets of benign and infection scripts used for testing.

The testing results are presented in Fig. 6.8(a). Most of the infection scripts yield a high risk score close to 1 while the majority of benign scripts gain a low risk score close to 0. Using the threshold 0.67, which is determined during the training process, our model achieves 1.79% false positive rate (FPR) and 95.45% true positive rate (TPR), with an overall accuracy of 96.92% and F-score of 0.97.

Next, we manually investigated the scripts that are falsely classified by our model. We found that the false positive samples are caused by extensively downloading and executing binaries, which is similar to the behavior of infection scripts. If the remote servers from where the binaries are downloaded are malicious, those samples may actually be for infection purposes. The false negative samples are mainly caused by extensive usage of commands – such as `declare`, `continue`, and `break` – that are abstracted with the *programming* (**PR**) infection capability. This capability allows the script to implement complex programming logic, which is not necessary to a successful infection.

Furthermore, we performed the receiver operating characteristic (ROC) analysis of our model based on the testing sets. The ROC curve shows the relationship between the FPR and the TRP. It provides a means of reviewing the performance of a model in terms of the *trade-off* between FPR and TPR. In our case, the ROC curve shown in Fig. 6.8(b) implies that the TPR grows rapidly when the FPR still remains low. The speed slows down when TPR is higher than 0.9 where the corresponding FPR is around 0.005. The area under the curve (AUC) for our model is 0.973, which indicates a good balance between false positives and false negatives.

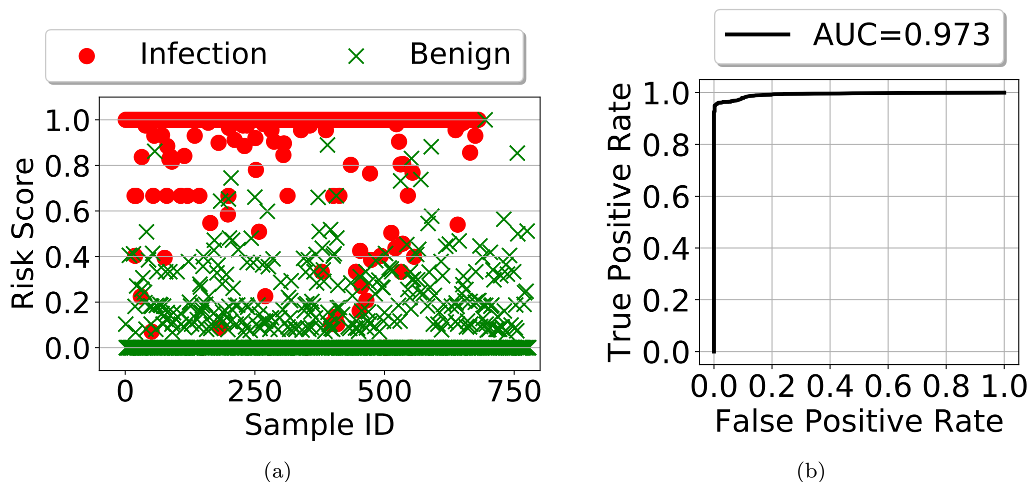



Figure 6.8: The testing results **(a)** and the ROC curve **(b)** of our infection detection model over infection scripts unseen during the training process.

### 6.2.7 Performance Overhead Evaluation of Soter

In this experiment, we quantified and evaluated the performance overhead of Soter on three popular IoT platforms:

- D-Link DCS-932L IP Camera (DCS-932L), equipping with MIPS 24KEc processor, 32MB SDRAM and 4MB flash memory, representing a *low-end*, resource-constrained platform.

- Raspberry Pi Compute Model 3 (CM3), equipping with ARM Cortex-A53 quad-core processor, 1GB DRAM and 8GB micro-SD card, representing a *mid-end*, generic multi-purpose platform.

- SolidRun HummingBoard Edge (HBE), equipping with ARM Cortex-A9 quad-core processor, 2GB DRAM and 16GB micro-SD card, representing a *high-end*, powerful computing platform.

We installed OpenWrt on DCS-932L, Raspbian on CM3, and Debian on HBE. All those Linux versions are officially supported by the device vendors, respectively. In addition, we installed and ran different applications on the three devices to emulate a regular workload. We installed light sensor [93] and MJPGE streaming [112] packages on DCS-932L that allow it to capture and send frames continuously to a remote receiver. We installed the Motion package [115] on CM3 that allows it to work as a Digital Video Recorder (DVR). We configured HBE to make it as an edge node that forwards network traffic. As a baseline, we first set up all the devices without SOTER and observed the CPU and memory usage. As a comparison, we then installed SOTER in the devices and observed the CPU and memory usage again.

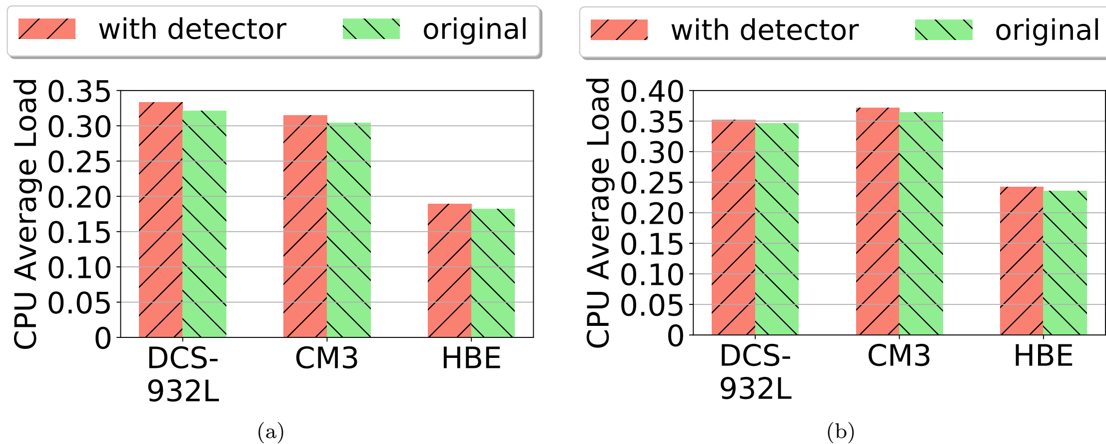



Figure 6.9: Average CPU loads of three types of IoT platforms without human interaction **(a)** and with human interaction **(b)**.

We quantified the CPU overhead using the CPU load average [70] within one minute. The CPU load average represents the total queue length of the active processes within a recent past time period (e.g., one minute in our experiment). If this number is greater than 1, it means at least one

active process on average is in the queue waiting to use CPU. In contrast, if this number is less than 1, it means less than one active process on average is in the queue waiting to use the CPU. Fig. 6.3.2.1 presents the results of the CPU load average in two scenarios. Fig. 6.9(a) shows the results when the devices are running without human interaction and Fig. 6.9(b) shows the results when users login to the system and perform regular operations such as copying files, removing files, opening files, etc. In the first scenario, SOTER introduces 3.73%, 3.61%, and 3.85% CPU load average for DCS-932L, CM3, and HBE, respectively. In the second scenario, SOTER introduces 1.73%, 2.20%, and 2.54% CPU load average for DCS-932L, CM3, and HBE, respectively. In both scenarios, SOTER introduces no more than 3.85% CPU load average, meaning that the length of the active process waiting queue only increases 3.85% on average due to SOTER.

To measure the memory overhead, we added extra code in the kernel to trace how much memory is allocated to the process of SOTER. Then, we compared that with free memory information obtained from `/proc/meminfo`. The results from different types of IoT platforms are quite similar, showing that SOTER occupies 2.7MB memory. This amount is only a small portion of the available memory even for low-end IoT platforms like DCS-932L, which has around 12MB free memory in our experiments.

## 6.3 IoT Malware Infection Prevention

We develop an infection prevention approach, PROCESS GAURD, as a host security function that can prevent infection process from being proceed. Similar to SOTER, the design objective of PROCESS GAURD is also *weightweight* so that it can also be deployed in resource-constrained Linux-based IoT devices, working togather with the infection detector.

### 6.3.1 Design Choices

To achieve trivial modification, PROCESS GAURD only includes a single component, the *flow rule generator*. The *flow rule generator* does not rely on any external libraries or frameworks except our system flow model. Since PROCESS GAURD will be invoked after the execution of SOTER, taking the results of SOTER as its input, it is registered as one of the host security functions. During the execution, PROCESS GAURD will generate corresponding system flow rules and update the system flow table via Flow Table Manager.

To achieve storage efficiency, Process Gaurd is implemented as a stateless security function without storing any states. The only memory space required by Process Gaurd is on the stack to create a system flow rule, which is then send to Flow Table Manager and destroyed after execution.

The computation overhead is also very limited since there is no loop or any computationally intensive task is conducted. The only computation task conducted by Process Gaurd is the extract the information that is required when it generates the system flow rule.

### 6.3.2 Implementation

Process Gaurd is implemented and runs completely in the kernel space and update the system flow table in real-time. Figure 6.10 provides an overview of Process Gaurd. Note that, Process Gaurd must be scheduled after Soter. The scheduling is achieved via Action Scheduler in our system flow model. To implement such a scheduling sequence, a system flow rule needs to be defined and installed in the system flow table. The system flow rule we used for this purpose is depicted in figure 6.11. For the match field, the source and the destination are wildcard meaning that this rule applies to all processes to all system resources and the operation type is *process_op_execve* meaning that this rule applies to `process_execve` event. For the action field, there are two actions specified and they are connected with the $\gg$ operator, which means they must be response sequentially and the results from the first action are passed to the second action processing. The priority of the rule is set to 1000, which is a default priority for system flow rules.

Once is scheduled, Process Gaurd will generate system flow rules to prevent the process, which has triggered the infection detector raise an infection alert, from being accessed any other system resources. The corresponding system flow rule for this intention is depicted in figure 6.12. For the match field, the source is extracted from the information passed in from Soter. The destination and operation types are wild card meaning that this rule applies to any system resources regarding to any operation types. For the action field, there is only a *deny* action meaning that this rule will block any process that generate the matched events. In this case, the block action response is directly responded to the hooks by Action Scheduler.

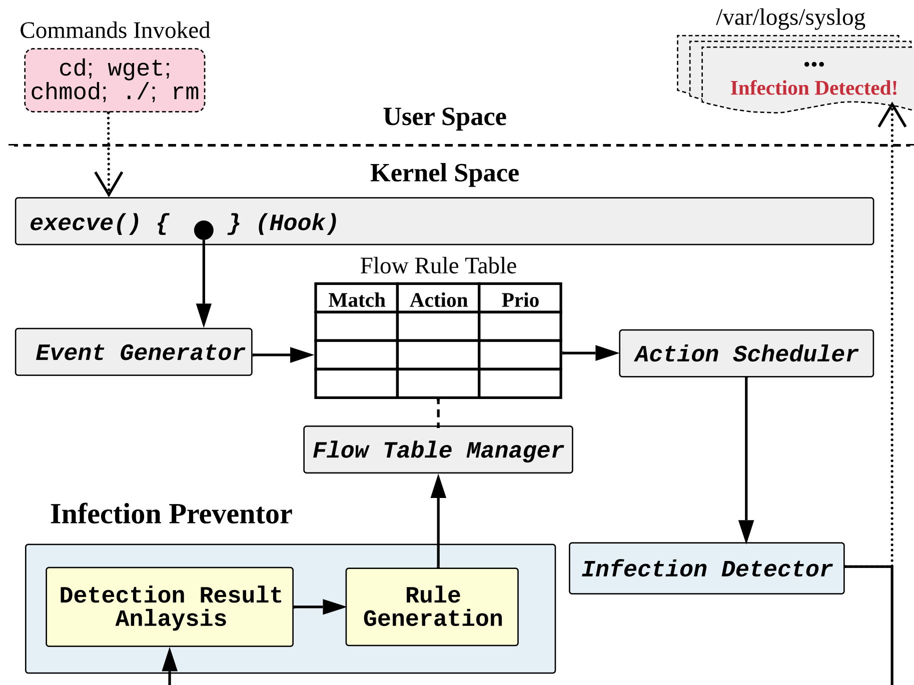



Figure 6.10: Overview of infection preventor implementation.

```
match=<*, *, process_op_execve>; actions=(external(1)>>external(2)); priority=1000
```

Figure 6.11: The system flow rule that is defined to implement the scheduling sequence that Process Gaurd is invoked after Soter.

```
match=<$pid, *, *>; actions=(deny); priority=10000
```

Figure 6.12: The system flow rule that will be generated by Process Gaurd. $pid is the process id passed from Soter.

#### 6.3.2.1 Evaluation

We conducted experiments using the configurations and real IoT devices described in § 6.2.7 to evaluate the performance overhead of both infection detector (Soter) and infection preventor.

We quantified the CPU overhead using the CPU load average [70] within one minute. The CPU load average represents the total queue length of the active processes within a recent past time period (e.g., one minute in our experiment). If this number is greater than 1, it means at least one active process on average is in the queue waiting to use CPU. In contrast, if this number is less than 1, it means less than one active process on average is in the queue waiting to use the CPU.

Fig. 6.3.2.1 presents the results of the CPU load average in two scenarios. Fig. 6.13(a) shows the results when the devices are running without human interaction and Fig. 6.13(b) shows the results when users login to the system and perform regular operations such as copying files, removing files, opening files, etc. In the first scenario, our infection detector and infection preventor introduce 5.29%, 4.93%, and 5.49% CPU load average for DCS-932L, CM3, and HBE, respectively. In the second scenario, our infection detector and infection preventor introduce 4.05%, 4.12%, and 5.08% CPU load average for DCS-932L, CM3, and HBE, respectively. In both scenarios, our infection detector and infection preventor introduce no more than 5.49% CPU load average, meaning that the length of the active process waiting queue only increases 5.49% on average due to our infection detector and infection preventor.

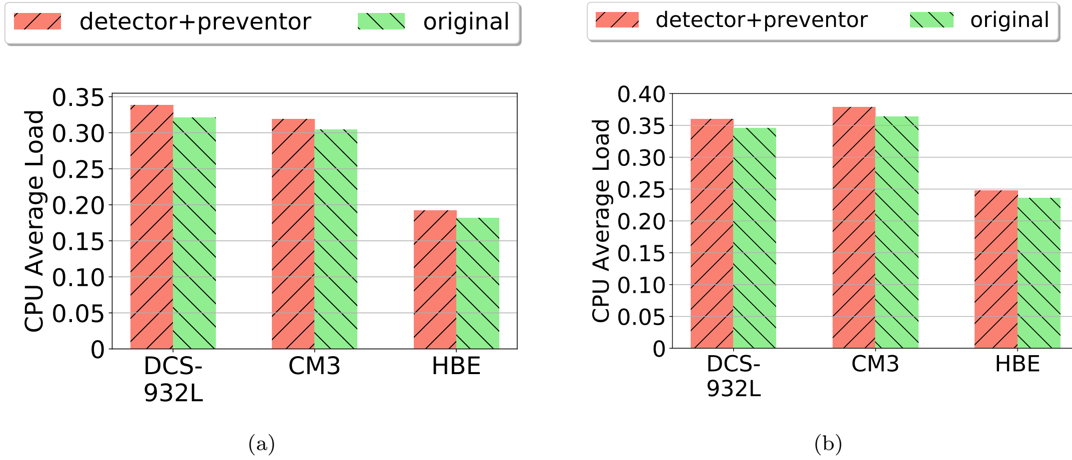



Figure 6.13: Average CPU loads of three types of IoT platforms without human interaction **(a)** and with human interaction **(b)**.

## 6.4 Cross-level IoT Malware Defense

In the experiments we have conducted in our evaluation of host-based IoT malware infection detector and prevention approach, we find that there are some cases where the host-based IoT malware prevention security function fail to stop the infection process. In this section, we develop a cross-level IoT malware defense approach that makes use of both network and host security functions. The objective of the cross-level IoT malware defense is to mitigate the real-world damage and financial loss caused by compromised IoT devices.

### 6.4.1 Problem Statement

The host-level IoT malware infection detection and prevention security functions feature *early* and *lightweight* detection and prevention. Because they will be deployed in resource-constrained IoT devices. However, we found that the host-level IoT malware infection detection and prevention security functions cannot always guarantee block of all malicious operations in the early stage.

As we further investigate the reasons behind, we can identify two major reasons. The first reason is that some infection process cannot be captured by our infection detector because they are emerging infection patterns and are not included in our dataset. As a result, the infection prevention security function cannot prevent such infection process. The second reason is that the prevention security function will only block the process that raises infection alerts. For those processes, which don't trigger infection alerts, our infection prevention security function will not block. However, our infection detector cannot raise an alert for *every* process that may potentially be related to an infection process because some of those processes may also be utilized by legitimate users. Blocking all related processes may result in a system crash. As such, some of those processes may be exploited by malware infection to conduct malicious activities, such as port scan.

In order to further mitigate real-world damage and financial loss, and at the same time, without impacting the system, network-level detection and prevention mechanisms are necessary. Unfortunately, solely relying on network-level detection and prevention mechanisms can hardly achieve *early* detection because they lack the detailed activities taking place a the host-level. Therefore, the host-level detection and prevention security functions and network-level detection and prevention security function need to collaborate with each other to achieve *early* detection as well as maximize the prevention rate.

### 6.4.2 Infrastructure-wide Architecture

A cross-level IoT malware defense security function is deployed in a centralized controller to enable collaboration between host-level security functions (Soter and Process Gaurd) and network-level security functions (virtual NIDS and virtual firewall) as is shown in Figure 6.14. In the network, an SDN switch is used to support dynamic traffic delivery for network-level security functions. Two network security functions are employed by the cross-level IoT malware defense. The virtual NIDS is used to detect any network traffic that appears to be port scanning. The virtual

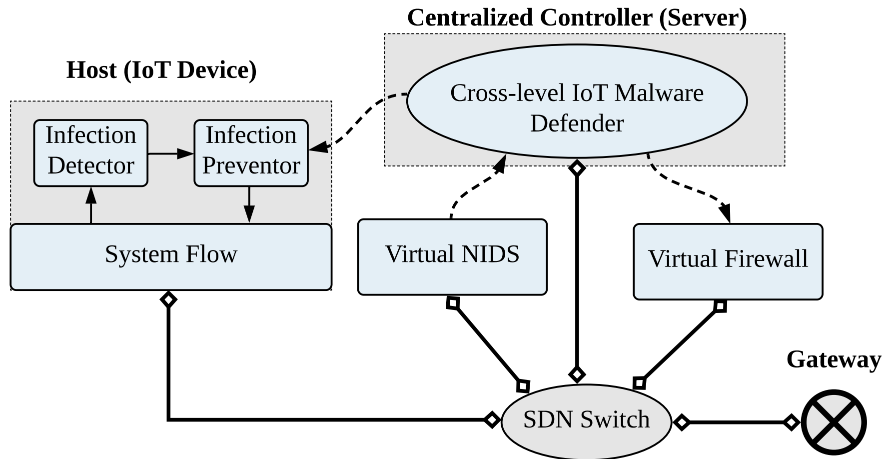



Figure 6.14: Infrastructure-wide architecture of cross-level IoT malware defense approach.

firewall is used to enforce access control rules on the network traffic. Once the virtual NIDS detects any port scan traffic, it will raise an alert to the cross-level IoT malware defense security function (1 in Figure 6.14). The latter then install a firewall rule into the virtual firewall to block the port scan traffic (2 in Figure 6.14). In the host, in this case, an infection detector (Soter) and a infection prevention mechanism (Process Gaurd) are employed by the cross-level IoT malware defense. The Soter is used to detect any infection attempts and Process Gaurd is used to enforce access control rules on processes related to malware infection. In the case where cross-level IoT malware defense security function receives alerts from virtual NIDS, it updates the system flow rule table by installing a new flow rule with an action that quarantines all processes, which access the sockets for a certain period of time (3 in Figure 6.14).

### 6.4.3 Implementation

The cross-level IoT malware defense security function is implemented on a dedicated server through high-level programming languages, in our case, Python and Java. The cross-level IoT malware defense security function consists of three key components, *alert parser*, *network flow rule generator*, and *system flow rule generator* as is shown in Figure 6.15. The northbound interface is responsible for receiving NIDS alerts and sending system or network flow rules to the system flow and virtual firewall respectively. Once a NIDS alert is received, it is firstly parsed by the *alert parser* to extract traffic information, such as IP addresses and port numbers. Those information is

then send to the *network flow rule generator* and *system flow rule generator* to construct proper flow rules. The generated network flow rule and system flow rule are shown in Figure 6.16 and Figure 6.16 respectively. In Figure 6.16, the source IP and destination port of the match field is provided through the alert parser. In Figure 5.1, the quarantine parameter of the action field is a predefined value indicating how long the quarantine will last.

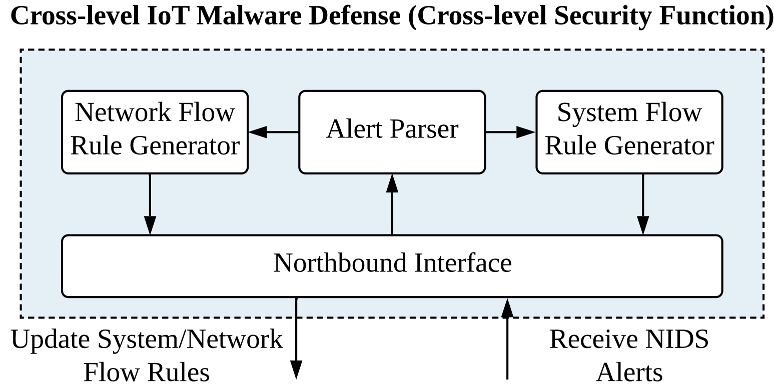



Figure 6.15: Key components of cross-level IoT malware defense security function.

```
match=<$sip, *, *, $dport>; actions=(deny); priority=1000
```

Figure 6.16: Network flow rule generated by cross-level IoT malware defense security function to block scanning traffic.

```
match=<*, *, socket_write_operate>; actions=(quarantine(3600)); priority=1000
```

Figure 6.17: System flow rule generated by cross-level IoT malware defense security function to quarantine all processes that access sockets.

### 6.4.4 Evaluation

In this section, we evaluate the effectiveness of our cross-level IoT malware defense security function. In particular, we will investigate whether the network-level security functions can collaborate with host-level security function to improve the prevention rate compared with that only deploying host-level security functions.

We used the honeypot environment developed in § 6.2.5 to provision software IoT devices as hosts. We deployed zeek NIDS as the virtual NIDS and Open vSwitch (OVS) as the virtual firewall to achieve network-level access control. The corss-level IoT malware defense security function is implemented using Java and Python and is run on a dedicated server connected to the same OVS. In the experiments, we deployed 43 IoT devices on CloudLab at 15 different sites (shown in Fig. 6.18) for 14 days and attracted 11157 valid IoT malware logins. The detailed statistics of our deployment is listed in Table 6.9.

Table 6.9: Software IoT device deployment statistics.

| **Region** | **Site** | **Deployed Device** | Valid Login |
|---|---|---|---|
| **North America** | Lowa | 12 | 3398 |
| | Los Angeles | 3 | 708 |
| | South Carolina | 3 | 761 |
| | Oregon | 3 | 834 |
| | Northern Virginia | 3 | 759 |
| **Europe** | Finland | 3 | 749 |
| | London | 3 | 691 |
| | Zurich | 3 | 703 |
| | Belgium | 3 | 662 |
| **Asia** | Taiwan | 2 | 523 |
| | Singapore | 1 | 212 |
| | Tokyo | 1 | 179 |
| | Osaka | 1 | 572 |
| | Seoul | 1 | 206 |
| **Australia** | Sydney | 1 | 200 |
| **Total** | | 43 | 11157 |

The host-level infection detection and prevention performance are listed in Table 6.10. As a result, among the 8534 valid infections, the cross-level IoT malware defender successfully defends 8392 IoT malware. In particular, the host-level infection detector achieves a FNR of 0.76% or TPR of 99.24%. The host-level infection preventor achieves a prevention rate of 98.34% We deterime the host-level prevention failure by counting the number of port scans that have occur because port scan happens after an infection. We totally observed 77 port scan alerts or 0.90% scan rate. The infection activities related to those port scans have been detected by our infection detector but the infection preventor fails to prevent the port scans. This is because the infection preventor can only prevent commands from executions after the infection detector makes decision. We found that those failures are caused by commands executed before infection detector can raise an alert. In summary, 1.66% of infections are failed to be prevented at the host level, including undetected infections, which takes

Figure 6.18: Geo-distribution of our deployed software IoT Devices.

Table 6.10: Real-time detection results of cross-level IoT malware defender. FN: False Negative; FNR: False Negative Rate; TPR: True Positive Rate; HP: Host-level Prevention; HPR: Host-level Prevention Rate; PS: Port Scan; PSR: Port Scan Rate; CP: Cross-level Prevention Rate

| | **Total** | **Alert** | **FN** | **FNR** | **TPR** | **HP** | **HPR** | **PS** | **PSR** | **CP** | **CPR** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Valid Infections** | 8,534 | 8,469 | 65 | 0.76% | 99.24% | 8,392 | 98.34% | 77 | 0.90% | 8,469 | 99.24% |
| **Unique Infections** | 145 | 133 | 12 | 8.28% | 91.72% | 133 | 91.72% | - | - | 133 | 91.72% |

0.76% of all valid infections.

To test the response time of cross-level IoT defender, we measure the time between a port scan starts and the time a system flow rule has been installed into the host. The response time to all the port scan cases is shown in Figure 6.19. All the responses are less than 6 seconds with the minimal of 2.524 seconds and the maximum of 33.954 seconds. Among all detected port scan cases, the average time our cross-level IoT malware defender takes to response is 5.32 seconds.

## 6.5 Conclusion

In this task, we deeply studied IoT malware infection process to build a knowledge base for IoT malware infection detection, which can achieve *early* and *lightweight* detection. We further developed IoT malware infection detection and prevention security functions based upon the system

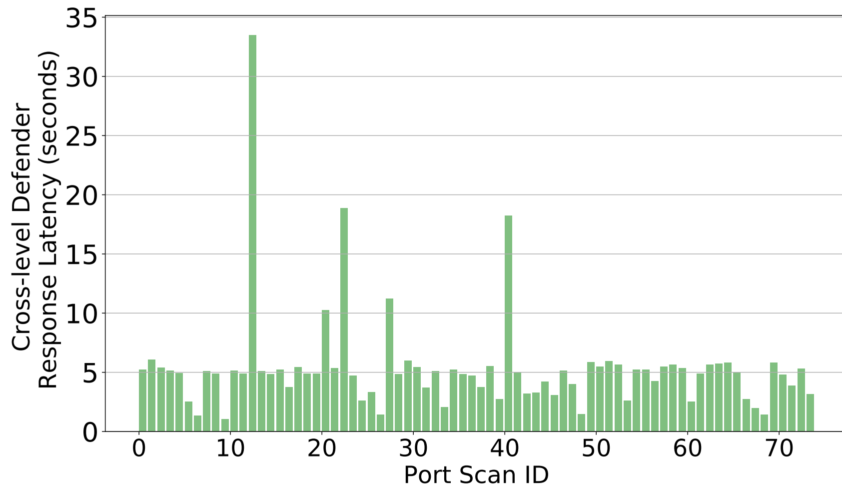



Figure 6.19: Cross-level defender response latency to each port scan incident.

flow model. Finally, we developed a cross-level IoT malware defense approach that allows host-level IoT malware infection detection and prevention security functions to collaborate with network-level port scan detection an prevention security functions. Our evaluation shows that the cross-level IoT malware defense security function can take advantages of network-level and host-level programmable security at the same time.

# Chapter 7

# Discussion

In this chapter, we will discuss potential limitations of the research conducted in this dissertation and point out the promising directions for this work.

## 7.1 Network-level Security Functions

In the network level, two security functions are redesigned to support programmability. In this dissertation, the techniques behind to support such programmability are SDN and NFV. However, the redesign of current network security functions presented in this dissertation addresses challenges that are independent to specific SDN and NFV implementations (e.g., OpenFlow and ClickOS). Instead, those challenges are related to specific network security functions. For the virtual firewall elasticity control, splitting firewall rules with firewall rule semantic consistency and updating flow rules correctly are two major challenges closely related to the virtual firewall while avoiding buffer overflow and optimal scaling are two key challenges to enable programmable network security functions. For the virtual NIDS, how to maintain effectiveness and efficiency of the new (programmable) NIDS requires domain knowledge of NIDS to choose appropriate designs.

In addition, the features stressed by the new design are also general to any network security functions, not limiting to the firewall and NIDS. That is programmable security functions should be not only scalable but also flexible. With the above design requirements in mind, in this dissertation, we demonstrate what potential challenges may be faced for different network security functions and what tools may be used to address those challenges. Especially, we hope that the micro-service

architecture used by the virtual NIDS could be an example for future comprehensive network security function development since according to our comparison, micro-service based virtual NIDS has better performance efficiency than the traditional NIDS.

## 7.2 Host-level Security Functions

In the host level, a flow-based model is employed to enable expressive security directives. Even though the current implementation of system flow is on the top of Linux operating systems, we consider that the design of system flow is general. This generality comes from our abstraction of system events from low-level system activities (i.e., system calls). Our abstraction bridges the semantic gap between different types and versions of operating systems to make system flow a uniform framework for different operating systems. The clear separation between the low-level system activities and the security directives, in addition, makes system flow easily portable to other operating systems as long as corresponding plugins are implemented. One of the meaningful future work is to extend system flow to support more operating systems, e.g, Windows and Mac OS.

The current development practice of host-level security functions on top of system flow leverages C programming language due to its generality. Inspired by abstractions in programmable network (e.g., Frenetic [61], Pyretic [113], Kinetic [86], and FRESCO [135]), one natural future direction of system flow is to provide a higher-level declarative programming interface, e.g., a domain-specific language (DSL), to hide tedious programming-language-related implementation details to a compiler and run-time system.

## 7.3 IoT Malware Defense

In this dissertation, we propose to use a weighted infection state machine for IoT malware infection detection because this method is sufficiently lightweight to run on IoT devices. As any detection approaches may face, our lightweight infection detector suffers some false alarms. After we carefully check with the false alarms, we find that those false alarms are caused by novel infection patterns that have not been included in our dataset. To remediate this, we can enable retaining our detection model using the new patterns collected during our large-scale online deployment. Since the classifier of our infection detector is implemented as a LKM, we can easily modify, recompile, and

reinstall it without rebooting the device. To further improve the detection accuracy of our infection detector, we may use deep learning approaches to model the infection process in the future. In addition, we can also augment our current dataset by making the samples in our dataset more balance. For example, we can use generative adversarial network (GAN) to augment our dataset with more infection samples that are not often seen in the wild.

# Chapter 8

# Conclusion

This dissertation devotes to exploring a framework to enable programmable security in modern software-defined infrastructures.

In the network level, we significantly redesigned two key security functions, the firewall and NIDS, such that they are more programmable. They now features not only scalable but also flexible. These improvements make the new firewall and NIDS more suitable to protect the networks in SDI. It is worth-noting that we employed the micro-service architecture when we redesign the NIDS. As a result, the micro-service architecture makes the new NIDS more efficient when deployed on a cloud environment.

In the host level, we proposed and presented the design a flow-based model, system flow, which enables expressive security directive. In addition, we have implemented the proposed model in Linux OS and conducted performance evaluation on it. The evaluations results imply that only a little performance overhead is introduced into the original OS.

We finally developed a cross-level IoT malware defense security function based upon ProSec. The developed IoT malware defense security function can achieve a high detection rate (more than 98%) and preventions rate (more than 99%) according to our comprehensive evaluation with large-scale virtual devices. The defense is quite responsive thanks to the programmability nature of ProSec, which is less than 6 seconds on average.

In summary, the contributions made by this dissertation are as follows.

- Developing a virtual firewall and evaluating its performance in SDI

- Developing a virtual NIDS and evaluating its performance in SDI
- Developing a system flow model and system flow rule table in Linux
- Implementing a malware infection detector based upon system flow model.
- Implementing a cross-level malware defense approach based upon the unified programming interface

Moreover, we expect the virtual firewalls and virtual NIDSes can provide suitable protections for SDN- and NFV-based networks, which are widely applied in SDI. We also envision the expressive security directions can make host security functions programmatically-extensible. Finally, we hope that our success in development of the cross-level IoT malware defense security function with ProSec can be an excellent use case example that can be referred by developers in the community to build more innovative security functions in the future.

# Bibliography


[1] Apparmor linux application security. https://wiki.ubuntu.com/AppArmor.

[2] CloudLab. http://www.cloudlab.us/.

[3] Header Space Library (Hassel). http://stanford.edu/~kazemian/hassel.tar.gz.

[4] IBM QRadar SIEM. https://www.ibm.com/us-en/marketplace/ibm-qradar-siem.

[5] LBNL/ICSI Enterprise Tracing Project. http://www.icir.org/enterprise-tracing/.

[6] Linux Audit. . https://linux.die.net/man/8/auditd.

[7] LMbench. http://lmbench.sourceforge.net/.

[8] LogRhythm. . https://logrhythm.com/.

[9] Project Floodlight. http://www.projectfloodlight.org/projects/.

[10] Selinux. https://github.com/SELinuxProject.

[11] Splunk. https://www.splunk.com/.

[12] VirusShare web site. https://virusshare.com/.

[13] Xen Toolstack. http://wiki.xen.org/wiki/XL.

[14] Firesheep. http://codebutler.com/firesheep, 2010.

[15] Bro Script for Detecting Cookie Hijacking. http://matthias.vallentin.net/blog/2010/10/taming-the-sheep-detecting-sidejacking-with-bro/, 2011.

[16] The CAIDA UCSD Anonymized Internet Traces 2016-0406. http://www.caida.org/data/passive/passive_2016_dataset.xml, 2016.

[17] Algo-logic systems. http://algo-logic.com/, 2017.

[18] Bro Script for Detecting DNS Tunneling. https://github.com/hhzzk/dns-tunnels, 2017.

[19] CodeSurfer. https://www.grammatech.com/products/codesurfer, 2017.

[20] Frama-c Software Analyzers. https://frama-c.com/, 2017.

[21] Nmap Security Scanner. https://nmap.org/, 2017.

[22] The Reigning King of IP Camera Botnets and its Challengers. http://blog.trendmicro.com/trendlabs-security-intelligence/reigning-king-ip-camera-botnets-challengers/, 2017.

[23] An implementation of an E-component of Network Intrusion Detection System. http://libnids.sourceforge.net/, 2018.

[24] Bro Scripting Language. https://www.bro.org/sphinx/scripting/, 2018.

[25] Open vswitch. https://www.openvswitch.org/, 2018.

[26] Scapy Project. http://www.secdev.org/projects/scapy/, 2018.

[27] The Bro Network Security Monitor. https://www.bro.org/, 2018.

[28] Port Scanner. https://en.wikipedia.org/wiki/Port_scanner, 2019.

[29] Snort Network Intrusion Detection & Prevention System. https://snort.org/, 2019.

[30] The Bro Network Security Monitor. https://www.zeek.org/, 2019.

[31] Muhamed Fauzi Bin Abbas and Thambipillai Srikanthan. *Low-Complexity Signature-Based Malware Detection for IoT Devices*, pages 181–189. Springer Singapore, Singapore, 2017.

[32] Ehab S Al-Shaer and Hazem H Hamed. Discovery of policy anomalies in distributed firewalls. In *Ieee Infocom 2004*, volume 4, pages 2605–2616. IEEE, 2004.

[33] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[34] Bashlex - Python parser for bash. https://github.com/idank/bashlex, 2020.

[35] A. BATES, D. J. TIAN, K. R. BUTLER, and MOYER. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Conference on Security Symposium (SEC)*, 2015.

[36] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. In *Computer Systems*, 1996.

[37] Edmund K Burke and Yuri Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017.

[38] P Busschbach. Network functions virtualisation-challenges and solutions. *Alcatel-Lucent Corp., France, Strategic White Paper*, 2013.

[39] Busybox man pages - user commands. https://busybox.net/downloads/BusyBox.html, 2020.

[40] Suresh Chari, Shai Halevi, and Wietse Venema. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *Network and Distributed System Security (NDSS) Symposium*, 2010.

[41] Huawei Home Routers in Botnet Recruitment. https://research.checkpoint.com/good-zero-day-skiddie/, 2017.

[42] C. Cowan, S. Beattie, C. Wright, and G. Kroah-hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *USENIX Security Symposium (SSYM)*, 2001.

[43] Cuckoo Sandbox Book. http://docs.cuckoosandbox.org/en/latest/, 2015.

[44] D-Link DCS-932L IP camera. https://linkstore.cl/adjuntos/991200-DCS-932L.PDF, 2020.

[45] Fan Dang, Zhenhua Li, Yunhao Liu, Ennan Zhai, Qi Alfred Chen, Tianyin Xu, Yan Chen, and Jingyu Yang. Understanding fileless attacks on linux-based iot devices with honeycloud. In *17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 482–493, 2019.

[46] Lorenzo De Carli, Robin Sommer, and Somesh Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[47] Drew Dean. Fixing Races for Fun and Profit: How to use access. In *USENIX Conference on Security Symposium (SEC)*, 2004.

[48] Hey Zollard, leave my Internet of Things alone! http://www.deependresearch.org/2013/12/hey-zollard-leave-my-internet-of-things.html, 2013.

[49] Rohan Doshi, Noah Apthorpe, and Nick Feamster. Machine learning ddos detection for consumer internet of things devices. *arXiv preprint arXiv:1804.04159*, 2018.

[50] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[51] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[52] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis . In *USENIX Annual Technical Conference (ATC)*, 2007.

[53] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones . In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[54] Md Nahid Hossain et al. SLEUTH: Real-time attack scenario reconstruction from COTS audit data . In *USENIX Conference on Security Symposium (SEC)*, 2017.

[55] P. Efstathopoulos et al. Labels and Event Processes in the Asbestos Operating System. In *ACM symposium on Operating systems principles (SOSP)*, 2005.

[56] Shiqing Ma et al. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows . In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[57] Ting-Fang Yen et al. Beehive: Large-scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks . In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[58] Xiaokui Shu et al. Threat Intelligence Computing . In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[59] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Security Symposium*, pages 817–832, 2015.

[60] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *NSDI*, volume 14, pages 533–546, 2014.

[61] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.

[62] Usha Devi Gandhi, Priyan Malarvizhi Kumar, R Varatharajan, Gunasekaran Manogaran, Revathi Sundarasekar, and Shreyas Kadu. Hiotpot: surveillance on iot devices against recent threats. *Wireless personal communications*, 103(2):1179–1194, 2018.

[63] A. Gember-Jacobson and A. Akella. Improving the safety, scalability, and efficiency of network function state transfers. In *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015.

[64] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 163–174. ACM, 2014.

[65] Github - The world's leading software development platform. https://github.com/, 2020.

[66] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, , and Eyal de Lara. The Taser Intrusion Recovery System . In *ACM symposium on Operating systems principles (SOSP)*, 2005.

[67] Jesse Gross, T Sridhar, P Garg, C Wright, I Ganga, P Agarwal, K Duda, D Dutt, and J Hudson. Geneve: Generic network virtualization encapsulation. *Internet Engineering Task Force, Internet Draft*, 2014.

[68] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation.

[69] R Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, volume 1, pages 5–7, 2012.

[70] Neil J. Gunther. UNIX Load Average Part 1: How It Works. https://www.helpsystems.com/resources/guides/unix-load-average-part-1-how-it-works, 2010.

[71] Michael Haag. Kaiten - Linux Backdoor. http://blog.michaelhaag.org/2013/12/kaiten-linux-backdoor.html, 2013.

[72] HaboMalHunter: Habo Linux Malware Analysis System. https://github.com/Tencent/HaboMalHunter, 2018.

[73] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and analysis of hajime, a peer-to-peer iot botnet. In *NDSS*, 2019.

[74] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[75] H. Hu, G.-J. Ahn, and K. Kulkarni. Detecting and Resolving Firewall Policy Anomalies. *IEEE Transactions on Dependable & Secure Computing*, 9(3):318–331, 2012.

[76] Ionut Indre and Camelia Lemnaru. Detection and prevention system against cyber attacks and botnet malware for information systems and internet of things. In *Intelligent Computer Communication and Processing (ICCP), 2016 IEEE 12th International Conference on*, pages 175–182. IEEE, 2016.

[77] Rhena Inocencio. BASHLITE Affects Devices Running on BusyBox. http://blog.trendmicro.com/trendlabs-security-intelligence/bashlite-affects-devices-running-on-busybox/, 2014.

[78] Marta Janus. Heads of the Hydra. Malware for Network Devices. https://securelist.com/heads-of-the-hydra-malware-for-network-devices/36396/, 2011.

[79] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[80] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking. In *USENIX Conference on Security Symposium (SEC)*, 2018.

[81] Xuxian Jiang, AAron Walters, Florian Buchholz, Dongyan Xu Yi-Min Wang, and Eugene H. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach . In *International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[82] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*, pages 97–112, 2017.

[83] G. Kandiraju, H. Franke, M. D. Williams, M. Steinder, and S. M. Black. Software defined infrastructures. *IBM J. Res. Dev.*, 58(2-3):2–2, March 2014.

[84] P. Kazemian, G. Varghese, and N. Mckeown. Header Space Analysis: Static Checking For Networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[85] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 15–27, 2013.

[86] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 59–72, 2015.

[87] Samuel T. King and Peter M. Chen. Backtracking Intrusions . In *ACM symposium on Operating systems principles (SOSP)*, 2003.

[88] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[89] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *ACM SIGOPS symposium on Operating systems principles*, 2007.

[90] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage Collecting Audit Log . In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[91] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[92] LightAidra source code. https://github.com/eurialo/lightaidra, 2016.

[93] LightSensor-daemon for OpenWrt. http://www.aboehler.at/hg/lightSensor-daemon, 2020.

[94] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu. Accelerating string matching using multi-threaded algorithm on gpu. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–5. IEEE, 2010.

[95] Linux man pages - user commands. https://linux.die.net/man/1/, 2020.

[96] LiSa Sandbox. https://github.com/danieluhricek/LiSa, 2017.

[97] A. X. Liu and M. G. Gouda. Complete Redundancy Removal for Packet Classifiers in TCAMs. *IEEE Transactions on Parallel & Distributed Systems*, 21(4):424–437, 2010.

[98] A. X. Liu, E. Torng, and Chad R. Meiners. Firewall Compressor: An Algorithm for Minimizing Firewall Policies. In *Proceedings of the 27th Conference on Computer Communications (INFOCOM'08)*, pages 176–180, 2008.

[99] Bing Liu. TheMoon - A P2P botnet targeting Home Routers. https://blog.fortinet.com/2016/10/20/themoon-a-p2p-botnet-targeting-home-routers, 2016.

[100] Michal Malik and Marc-Etienne M.Léveillé. Meet Remaiten - a Linux bot on steroids targeting routers and potentially other IoT devices. https://www.welivesecurity.com/2016/03/30/meet-remaiten-a-linux-bot-on-steroids-targeting-routers-and-potentially-other-iot-devices/, 2016.

[101] MMD-0037-2015 - A bad Shellshock & Linux/XOR.DDoS CNC "under the hood". http://blog.malwaremustdie.org/2015/07/mmd-0037-2015-bad-shellshock.html, 2015.

[102] MMD-0052-2016 - Overview of "SkidDDoS" ELF++ IRC Botnet. http://blog.malwaremustdie.org/2016/02/mmd-0052-2016-skidddos-elf-distribution.html, 2016.

[103] MMD-0059-2016 - Linux/IRCTelnet (new Aidra) - A DDoS botnet aims IoT w/ IPv6 ready. http://blog.malwaremustdie.org/2016/10/mmd-0059-2016-linuxirctelnet-new-ddos.html, 2016.

[104] MMD-0058-2016 - Linux/NyaDrop - a linux MIPS IoT bad news. http://blog.malwaremustdie.org/2016/10/mmd-0058-2016-elf-linuxnyadrop.html, 2017.

[105] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, pages 459–473, 2014.

[106] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[107] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Asaf Shabtai, Dominik Breitenbacher, and Yuval Elovici. N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22, 2018.

[108] Sadegh M. Milajerdi, Rigel Gjomemo, Birhanu Eshete†, R. Sekar, and V.N. Venkatakrishnan. HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows . In *IEEE Symposium on Security and Privacy*, 2019.

[109] Leaked mirai source code for research/ioc development purposes. https://github.com/jgamblin/Mirai-Source-Code, 2016.

[110] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[111] G. Misherghi, L. Yuan, Z. Su, Chen-Nee Chuah, and H. Chen. A general framework for benchmarking firewall optimization techniques. *IEEE Transactions on Network & Service Management*, 5(4):227–238, 2008.

[112] MJPG Streamer for OpenWrt. https://openwrt.org/packages/pkgdata/mjpg-streamer, 2020.

[113] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *USENIX NSDI*, 2013.

[114] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111, 2018.

[115] The Motion program. https://motion-project.github.io/index.html, 2020.

[116] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. SilverLine: Preventing Data Leaks from Compromised Web Applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[117] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A Holland, Peter Macko, Diana L MacLean, Daniel W Margo, Margo I Seltzer, and Robin Smogor. Layering in Provenance Systems. In *USENIX Annual technical conference (ATC)*, 2009.

[118] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on Android. In *USENIX Conference on Security Symposium (SEC)*, 2016.

[119] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Minh Hoang Dang, N Asokan, and Ahmad-Reza Sadeghi. Dïot: A federated self-learning anomaly detection system for iot. *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[120] Tj OConnor, William Enck, W. Michael Petullo, and Akash Verma. PivotWall: SDN-Based Information Flow Control . In *Symposium on SDN Research (SOSR)*, 2018.

[121] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[122] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. Iotpot: analysing the rise of iot compromises. In *9th USENIX Workshop on Offensive Technologies (WOOT'15)*, 2015.

[123] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: A yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, pages 289–300. ACM, 2006.

[124] Thomas Pasquier, Xueyuan Han, Thomas Moyer, Adam Bates, Olivier Hermant, David Eyers, Jean Bacon, and Margo Seltzer. Runtime Analysis of Whole-System Provenance. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[125] D. J. POHLY, S. MCLAUGHLIN, P. MCDANIEL, and K BUTLER. Hi-fi: collecting high-fidelity whole-system provenance. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[126] Phillip A. Porras, Martin W. Fong, and Alfonso Valdes. A Mission Impact Based Approach to INFOSEC Alarm Correlation. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[127] Paul Quinn and Uri Elzur. Network service header. *draft-quinnsfc-nsh-01*, 2014.

[128] Radare2: a portable reversing framework. https://github.com/idank/bashlex, 2020.

[129] Brickerbot results in pdos attack. https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/, 2017.

[130] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, volume 13, pages 227–240, 2013.

[131] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 41–52. ACM, 1995.

[132] Rodrigo Roman, Javier Lopez, and Masahiro Mambo. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, 78:680–698, 2018.

[133] Rsyslog. https://en.wikipedia.org/wiki/Rsyslog, 2020.

[134] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[135] Seung Won Shin, Phillip Porras, Vinod Yegneswara, Martin Fong, Guofei Gu, and Mabry Tyson. Fresco: Modular composable security services for software-defined networks. In *20th Annual Network & Distributed System Security Symposium*. NDSS, 2013.

[136] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance in e-Science . In *ACM SIGMOD Record*, 2005.

[137] S. SITARAMAN and S. VENKATESAN. Forensic analysis of file system intrusions using improved backtracking . In *IEEE International Workshop on Information Assurance (IWIA)*, 2005.

[138] Ben Smith, Rick Grehan, Tom Yager, and DC Niemi. Byte-unixbench: A unix benchmark suite. *Technical report*, 2011.

[139] Robin Sommer, Vern Paxson, and Nicholas Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurrency and Computation: Practice and Experience*, 21(10):1255–1279, 2009.

[140] John Sonchack, Jonathan M Smith, Adam J Aviv, and Eric Keller. Enabling practical software-defined networking security applications with ofx. In *NDSS*, volume 16, pages 1–15, 2016.

[141] V. Srinivasan, S. Suri, and G. Varghesea. Packet Classification Using Tuple Space Search. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, 1999.

[142] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. Looking inside the black-box: capturing data provenance using dynamic instrumentation . In *International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes (IPAW)*, 2014.

[143] Jiawei Su, Danilo Vasconcellos Vargas, Sanjiva Prasad, Daniele Sgandurra, Yaokai Feng, and Kouichi Sakurai. Lightweight classification of iot malware based on image recognition. *arXiv preprint arXiv:1802.03714*, 2018.

[144] Hao Sun, Xiaofeng Wang, Rajkumar Buyya, and Jinshu Su. Cloudeyes: Cloud-based malware detection with reversible sketch for resource-constrained internet of things (iot) devices. *Software: Practice and Experience*, 47(3):421–441, 2017.

[145] New Linux Malware Exploits CGI Vulnerability. http://blog.trendmicro.com/trendlabs-security-intelligence/new-linux-malware-exploits-cgi-vulnerability/, 2017.

[146] Persirai: New Internet of Things (IoT) Botnet Targets IP Cameras. http://blog.trendmicro.com/trendlabs-security-intelligence/persirai-new-internet-things-iot-botnet-targets-ip-cameras/, 2017.

[147] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[148] Tore Ulversoy. Software defined radio: Challenges and opportunities. *IEEE Communications Surveys & Tutorials*, 12(4):531–550, 2010.

[149] F. Valeur, C. Kruegel G. Vigna, and R. A. Kemmerer. Comprehensive approach to intrusion detection alert correlation . In *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2004.

[150] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Recent Advances in Intrusion Detection*, pages 107–126. Springer, 2007.

[151] Jornt van der Wiel, Vicente Diaz, Yury Namestnikov, and Konstantin Zykov. https://securelist.com/hajime-the-mysterious-evolving-botnet/78160/, 2017.

[152] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, 2008.

[153] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. {JIGSAW}: Protecting resource access by inferring programmer expectations. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 973–988, 2014.

[154] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding Name Resolution Vulnerabilities in Programs. In *USENIX Conference on Security Symposium (SEC)*, 2012.

[155] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process Firewalls: Protecting Processes During Resource Access. In *ACM European Conference on Computer Systems (EuroSys)*, 2013.

[156] Virustotal-free online virus, malware and url scanner. 2020.

[157] VMware. Introducing vmware validated designs for software-defined data center. https://www.vmware.com/pdf/vmware-validated-design-30-sddc-introduction.pdf.

[158] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[159] Intel white paper. The infrastructure for innovation: Software defined, cloud ready. https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2016/03/intel-whitepaper-160301.pdf, 2016.

[160] Microsoft white paper. Drive efficiency with a software-defined data center. https://www.microsoft.com/en-us/cloud-platform/software-defined-datacenter.

[161] Linux.Wifatch. https://gitlab.com/rav7teif/linux.wifatch, 2015.

[162] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance . In *ACM SIGCOMM Computer Communication Review*, 2014.

[163] Claud Xiao and Cong Zheng. New IoT/Linux Malware Targets DVRs, Forms Botnet. https://researchcenter.paloaltonetworks.com/2017/04/unit42-new-iotlinux-malware-targets-dvrs-forms-botnet/, 2017.

[164] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis . In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[165] Tianlong Yu, Seyed K Fayaz, Michael Collins, Vyas Sekar, and Srinivasan Seshan. Psi: Precise security instrumentation for enterprise networks. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSSÍ7)*, 2017.

[166] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[167] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 5. ACM, 2015.

[168] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[169] ZDNet.com. The rise of sddc and the future of enterprise it. https://www.zdnet.com/article/the-rise-of-sddc-and-the-future-of-enterprise-it/, 2016.

[170] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[171] Charles C Zhang, Marianne Winslett, and Carl A Gunter. On the safety and efficiency of firewall policy deployment. In *2007 IEEE Symposium on Security and Privacy (S&P'07)*, pages 33–50. IEEE, 2007.

[172] Nuyun Zhang, Hongda Li, Hongxin Hu, and Younghee Park. Towards effective virtualization of intrusion detection systems. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 47–50. ACM, 2017.

[173] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance . In *ACM symposium on Operating systems principles (SOSP)*, 2011.

[174] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale . In *ACM SIGMOD International Conference on Management of data (SIGMOD)*, 2010.

[175] Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks . In *Tech. Rep. EECS-2009-145*, 2009.