ABSTRACT

WANG, RUOWEN. Automatic Generation, Refinement and Analysis of Security Policies. (Under the direction of Dr. Peng Ning and Dr. William Enck.)

Security policies play an important role in ensuring the security of computer systems. In firewalls and intrusion detection systems (IDSes), security policies are the core components for identifying and blocking malicious network traffic. In access control systems, security policies are used to check and restrict access permissions to privileged resources. The security of computer systems highly depends on the quality of security policies. A security policy must be carefully developed, so that malicious attacks are blocked while benign operations are not affected. Unfortunately, developing such security policies is a manual and time-consuming task. It requires policy engineers to have both security expertise and domain knowledge of the target systems being protected. In addition, policy engineers need to keep refining security policies, since both attacks and system functionality keep upgrading.

In this dissertation, we focus on the goal of automating the development of security policy, in order to help reduce the manual workload for policy engineers and ensure the high quality of security policies. Specifically, we present three works from three aspects of the security policy development lifecycle: 1) automated policy generation, 2) automated policy refinement and 3) automated policy analysis.

In the first work, we present MetaSymploit, the first system of automatic attack script analysis and IDS policy generation. MetaSymploit focuses on attacks written in scripting languages. Such attack scripts can be quickly created and spread to exploit zero-day vulnerabilities. Unfortunately, manual analysis and IDS policy creation can hardly catch up the emerging speed of new attack scripts. MetaSymploit can automatically analyze newly created attack scripts and generate IDS signatures to defend against the attack scripts. We evaluated MetaSymploit using 45 real-world attack scripts from Metasploit, the most popular attack framework. MetaSymploit automatically generated Snort IDS rules that effectively detect all the attacks launched by these scripts. Furthermore, the results showed that MetaSymploit complements and improves existing Snort rules that are manually written by the official Snort team.

In the second work, we present EASEAndroid, the first SEAndroid analytic platform for automatic policy analysis and refinement. EASEAndroid focuses on mandatory access control (MAC) policies in security-enhanced Android (SEAndroid), a new security feature in Android systems. Developing SEAndroid policy is a challenging task, requiring many iterations of refinement. EASEAndroid models and automates the policy refinement innovatively using semi-supervised learning. EASEAndroid continually learns new access patterns based on audit logs collected from Android devices and producing policy refinement. We evaluated EASEAndroid on 1.3 million audit logs from real-world Android devices.
Samsung devices. EASEAndroid successfully learned 2,518 new access patterns and generates 331 new policy rules. In addition, EASEAndroid also discovered two new attacks in real world directly targeting SEAndroid MAC mechanism.

In the third work, we present SPOKE, an SEAndroid Policy Knowledge Engine with automatic domain knowledge collection for policy analysis. SPOKE focuses on the challenge that it is difficult to obtain domain knowledge of target system's functionality, causing policy engineers to develop unjustified and over-permissive policy rules, which increases the attack surface. To addresses this challenge, SPOKE first automatically extracts domain knowledge from rich-semantic functional tests. Second, SPOKE uses the knowledge for policy analysis with two outputs: 1) It reveals unjustified rules by matching rules with collected functionality knowledge. 2) It identifies over-permissive access patterns allowed by the unjustified rules. We evaluated SPOKE using 665 functional tests targeting a set of security functionalities developed by a major Android vendor. SPOKE successfully collected 12,491 access patterns for 28 functionalities as domain knowledge, and used the knowledge to reveal 320 unjustified policy rules and 210 over-permissive access patterns defined by those rules, including one related to the notorious `libstagefright` vulnerability. These findings have been confirmed by policy engineers.
Automatic Generation, Refinement and Analysis of Security Policies

by
Ruowen Wang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2016

APPROVED BY:

__________________________  ____________________________
Dr. Douglas Reeves          Dr. Munindar Singh

__________________________  ____________________________
Dr. Peng Ning               Dr. William Enck
Co-chair of Advisory Committee Co-chair of Advisory Committee
DEDICATION

To my parents.
ACKNOWLEDGEMENTS

I would like to thank my advisors Dr. Ning and Dr. Enck for their great helps. I also want to thank many colleagues in Samsung for helping me on my research projects. I also like to thank my committee members Dr. Ning, Dr. Enck, Dr. Reeves and Dr. Singh for helping me and attending my defense.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................................. vii

**LIST OF FIGURES** .............................................................. viii

**Chapter 1  Introduction** ....................................................... 1
  1.1 Thesis Statement ......................................................... 2
  1.2 Automated Policy Generation: MetaSymploit ........................................ 2
  1.3 Automated Policy Refinement: EASEAndroid ........................................ 2
  1.4 Automated Policy Analysis: SPOKE ........................................ 3

**Chapter 2  Background** .......................................................... 4
  2.1 Security Policy in Computer Systems ........................................... 4
  2.2 Related Work in Security Policy Development ..................................... 5
    2.2.1 Network-based Security Policy .............................................. 6
    2.2.2 Host-based Security Policy ................................................. 6
  2.3 New Challenges in Security Policy Development ................................... 7
    2.3.1 New Attacks ............................................................... 7
    2.3.2 New Security Mechanisms ................................................. 7
    2.3.3 New Techniques ........................................................... 8

**Chapter 3  MetaSymploit** ...................................................... 9
  3.1 Introduction ............................................................... 10
  3.2 Background ............................................................... 12
  3.3 MetaSymploit .............................................................. 15
    3.3.1 Problem Statement and Assumptions .................................... 15
    3.3.2 MetaSymploit Overview ................................................ 15
    3.3.3 Symbolic Execution Layer .............................................. 18
    3.3.4 Signature Generator ................................................... 20
  3.4 Implementation ............................................................ 23
    3.4.1 A Lightweight Symbolic Execution Engine for Ruby ..................... 24
    3.4.2 Adaptation for Metasploit .............................................. 26
  3.5 Evaluation ................................................................. 27
    3.5.1 Coverage Testing with Symbolic Execution Engine ....................... 28
    3.5.2 Effectiveness Validation using Real-world Attacks ....................... 30
    3.5.3 Comparison with Official Snort Rules ................................... 31
  3.6 Discussion ................................................................. 33
  3.7 Related Work ............................................................. 34
  3.8 Summary ................................................................. 35
  3.9 Acknowledgements ....................................................... 36

**Chapter 4  EASEAndroid** ..................................................... 37
  4.1 Introduction ............................................................. 38
  4.2 Background ............................................................. 40
4.2.1 SELinux and SEAndroid .............................................. 40
4.2.2 Semi-Supervised Learning ........................................ 41
4.3 Problem ................................................................. 42
4.4 EASEAndroid ............................................................ 43
  4.4.1 Nearest-Neighbors-based Classifier ............................ 45
  4.4.2 Pattern-to-Rule Distance Measurer ............................ 47
  4.4.3 Co-Occurrence Learner ....................................... 49
  4.4.4 Learning Balancer & Combiner ................................ 50
  4.4.5 Policy Refinement Generator ................................ 51
4.5 Evaluation .............................................................. 52
  4.5.1 Environment Setup ............................................. 52
  4.5.2 Audit Log & Existing Knowledge .............................. 53
  4.5.3 Coverage & Precision of the Classification by EASEAndroid .. 53
  4.5.4 Case Study of Refinement Generation & Comparison with Human Policy ... 56
  4.5.5 Case Study of Classified Malicious Access Patterns .......... 57
4.6 Discussion .............................................................. 59
4.7 Related Work ........................................................... 60
4.8 Summary ............................................................... 60

Chapter 5  SPOKE ............................................................. 64
  5.1 Introduction .......................................................... 65
  5.2 Background and Definitions ...................................... 67
    5.2.1 SEAndroid Basics ........................................... 67
    5.2.2 SEAndroid Definitions ..................................... 68
    5.2.3 Android Functional Testing .................................. 69
    5.2.4 Definitions introduced by SPOKE ......................... 70
  5.3 Problem Statement and Assumptions ............................. 72
  5.4 SPOKE ................................................................. 72
     5.4.1 Automatic Knowledge Collection ........................... 74
     5.4.2 Policy Rule Justification w.r.t. K .......................... 78
     5.4.3 Attack Surface Analysis of Policy Rules ................ 80
  5.5 Evaluation ............................................................ 82
     5.5.1 Data Set and Research Questions .......................... 82
     5.5.2 Knowledge Base Construction from MDM Functional Tests .... 83
     5.5.3 Justifying Policy Rules w.r.t. MDM Domain Knowledge .... 85
     5.5.4 Analyzing attack surface of Policy Rules ................ 87
  5.6 Discussion ........................................................... 89
  5.7 Related Work ......................................................... 91
  5.8 Summary ............................................................. 91

Chapter 6  Future Work and Conclusion ................................... 93
  6.1 Future Work ........................................................ 93
    6.1.1 Big Data Analytics for Security Policy Development .......... 93
    6.1.2 Artificial Intelligence for Security Policy Development ........ 94
    6.1.3 New Attacks Against Security Policy Development ........... 94
6.2 Conclusion ......................................................................................... 95

BIBLIOGRAPHY ...................................................................................... 97

APPENDICES .......................................................................................... 110
  Appendix A  MetaSymploit ................................................................. 111
    A.1 Example of Rule Comparison ................................................... 111
  Appendix B  EASEAndroid ................................................................. 114
    B.1 Audit Log Collection ............................................................... 114
# LIST OF TABLES

Table 3.1  The distribution of different situations in the symbolic execution of the 548 Metasploit attack scripts. .................................................. 28  
Table 3.2  The list of three Metasploit attack scripts which evade the detection from 3 Official Snort Rules ................................................................. 32  
Table 4.1  The coverage and precision of EASEAndroid with different threshold settings after comparison with ground truth. The first three columns summarize the overall classification coverage over the 3530 patterns. The following four columns give more details about the percentages of each set of true/false-classified benign and malicious patterns. Row 4 is the threshold setting used in Figure 4.3(b). ................................................................. 54  
Table 5.1  Examples of background & dummy test access patterns .................................................. 84  
Table 5.2  Summary and reasoning of policy rule justification result .................................................. 86
Figure 3.1 MetaSymploit consists of two major parts drawn in grey. (The arrows show the workflow of an attack script analysis.) .............................................. 16
Figure 3.2 Pattern comparison between 53 MetaSymploit generated rules and 50 official Snort rules for 22 Metasploit attack scripts. ............................................. 31
Figure 4.1 EASEAndroid consists of four learning components and a policy generator to iteratively learn new patterns and refine policy ......................... 44
Figure 4.2 A decision tree example based on rules related to subject domain as untrusted_app. The black nodes are from existing rules and the blue nodes are semantic siblings. ................................................................. 48
Figure 4.3 The comparison between naive matching and EASEAndroid on analyzing the same set of access patterns ......................................................... 62
Figure 4.4 The distribution of malicious access patterns classified by EASEAndroid. ... 63
Figure 5.1 SPOKE consists of three components from collecting test-driven tracing logs to generating final report by the analytic engine. ......................... 73
Figure 5.2 Cross-layer correlation between low-level access patterns and high-level functionality traces during a functional test test_addFirewallRule() ... 76
Figure 5.3 A bipartite graph illustrating two over-permissive access patterns (red) by a policy rule allowing a vulnerable keyboard app to read and write critical system files. .......................................................... 81
Figure 5.4 The summary of access patterns, functionality trace and justified policy rules in the knowledge base using MDM functional tests ......................... 83
Figure 5.5 The average logging rate and running time of 28 MDM test categories in SPOKE knowledge extraction platform ................................. 85
Figure 5.6 The representative bipartite result of attack surface analysis generated by SPOKE. Subjects are red nodes in left. Objects are blue nodes in right. Justified access patterns are in grey. Over-permissive access patterns in red allow unjustified subjects to access critical anonymized security files, which have been confirmed and revoked by policy engineers. ......................... 88
Security policies play an important role in ensuring the security of computer systems. In firewalls and intrusion detection systems (IDSes), security policies (e.g., firewall/IDS rules) are the core components for identifying and blocking malicious network traffic and host activities. In access control systems, security policies (e.g., access control list/rules) are used to check and restrict access permissions to privileged resources.

The security of computer systems highly depends on the quality of security policies. A security policy must be carefully developed, so that malicious attacks can be blocked while benign operations are not affected. Developing such security policies is usually a manual and time-consuming task for policy engineers, who must have both security expertise of various attacks and domain knowledge of the target systems being protected. Furthermore, since new attacks keep emerging and system functionality keeps upgrading, policy engineers need to keep refining security policies, in order to meet the new security requirements in a timely manner (e.g., defending against new attacks, protecting new functionality).

In this dissertation, we focus on the goal of automating the development of security policy, to help reduce the time cost and manual workload for policy engineers and ensure the high quality of security policies. Specifically, we present three works from three aspects of the security policy development lifecycle: 1) automated policy generation, 2) automated policy refinement and 3)
1.1 Thesis Statement

Security policy development can be automated using 1) symbolic analysis of attack programs for policy generation, 2) machine learning of access control patterns for policy refinement and 3) test-driven domain knowledge collection for policy analysis, to help reduce manual workload of human policy engineers and ensure high policy quality.

1.2 Automated Policy Generation: MetaSymploit

In the first work, we present MetaSymploit (published in USENIX Security ’13)[Wan13], the first system of fast attack script analysis and automatic IDS policy generation. MetaSymploit focuses on attacks written in scripting languages (e.g., Python, Ruby). Such attack scripts can be quickly created and widely distributed to exploit zero-day vulnerabilities. We design a new program analysis technique called “security-enhanced symbolic execution” to automatically analyze the newly created attack script and generate specific IDS signatures to defend against all possible attacks launched by the script from day one of its distribution. We implement a prototype of MetaSymploit targeting Metasploit, the most popular industrial attack framework. In the experiments on 45 real attack scripts, MetaSymploit automatically generates Snort IDS rules that effectively detect the attacks launched by the 45 scripts. Furthermore, the results show that MetaSymploit substantially complements and improves existing Snort rules that are manually written by the official Snort team.

1.3 Automated Policy Refinement: EASEAndroid

In the second work, we present EASEAndroid (published in USENIX Security ’15)[Wan15], the first SEAndroid analytic platform for automatic policy analysis and refinement. EASEAndroid focuses on mandatory access control (MAC) policies used in mobile devices running security-enhanced Android (SEAndroid), which is a new security feature introduced by Google since Android 4.3. Developing SEAndroid MAC policy is a challenging task, requiring many iterations of refinement for commercial deployment. To address this challenge, EASEAndroid models and automates the policy refinement process innovatively using semi-supervised learning, a knowledge base construction technique. Specifically, given an existing policy and a small set of known access patterns, EASEAndroid continually expands the knowledge base by learning new access patterns based on audit logs collected from Android devices and producing suggestions for policy refinement. We evaluate
EASEAndroid on 1.3 million audit logs from real-world Samsung mobile devices. EASEAndroid successfully learns 2,518 new access patterns and generates 331 new policy rules. In addition, EASEAndroid also discovers eight categories of malicious access patterns, two of which are new attacks directly targeting SEAndroid MAC mechanism.

1.4 Automated Policy Analysis: SPOKE

In the third work, we present SPOKE, an SEAndroid Policy Knowledge Engine with automatic domain knowledge collection for policy analysis. SPOKE focuses on the challenge that it is difficult for policy engineers in practice to obtain detailed and updated domain knowledge of target system’s functionality. This causes policy engineers to develop unjustified and over-permissive policy rules that are not only inconsistent with the target system’s functionality, but also increase the attack surface of the target system. To addresses this challenge, SPOKE first automatically extracts domain knowledge from rich-semantic functional tests. Second, SPOKE uses the knowledge for policy analysis. Our analysis yield two high-impact outputs: 1) It reveals unjustified rules by matching rules with collected functionality knowledge. 2) It identifies over-permissive access patterns allowed by the unjustified rules. We evaluated SPOKE using 665 functional tests targeting a set of security functionalities developed by a major Android vendor. SPOKE successfully collected 12,491 access patterns for 28 functionalities as domain knowledge, and used the knowledge to reveal 320 potentially unnecessary policy rules and 210 over-permissive access patterns defined by those rules, including one related to the notorious libstagefright vulnerability. These findings have been confirmed by policy engineers.
Security policy is a definition or a specification aimed at securing a system or an organization. It defines the constraints on the behavior of both entities and adversaries, and is typically enforced by a security mechanism in the system[PT02; Bis04]. For example, physical security such as door lock is a traditional security mechanism and a security policy defines who can hold the key to unlock the door and who can enter through the door (e.g., only the house owner can have the key; guests can enter the house with the house owner’s invitation).

2.1 Security Policy in Computer Systems

In computer security, security policy plays an important role in ensuring the security of computer and network systems. Formally, if a computer system is considered as a finite state machine, a security policy is defined as a statement that partitions the states of the computer system into a set of authorized (secure) states and a set of unauthorized (insecure) states. When the security policy is enforced, its goal is to ensure that the computer system cannot enter an unauthorized state from an authorized state[Bis04].

Security policies are widely used by various computer security systems and mechanisms in different forms. In network firewalls and intrusion detection systems (IDSes), firewall and IDS rules
are used as security policies to detect and defend against malicious network traffic. Such policies typically use network-related attributes such as IP addresses, port numbers, packet payloads, etc, to determine whether a network packet should be accepted or blocked. For instance, Snort\textsuperscript{[Snoa]} is a popular open-source IDS, whose rules are actively updated by the official Snort team and other contributors. The Snort rules can be specified using IP address, port number, payload string and also regular expression to match against network packets. The rules also define what action to take on matched packets, such as pass, alert, reject and drop\textsuperscript{[Snob]}. For example, the following is a Snort rule that raises an alert in the log with the defined message, when external IP addresses connect to host addresses with port 80.

\begin{verbatim}
alert tcp $EXTERNAL_NET any -> $HOST_NET 80 (msg: "sample alert from external net to host net on port 80")
\end{verbatim}

In operating systems, anti-malware software uses malware patterns to define signatures as security policies to detect and prevent malware. Access control mechanisms such as Discretionary Access Control (DAC) and Mandatory Access Control (MAC) confine capabilities of different privileged entities and restrict access to sensitive/privileged resources, by enforcing access control rules as security policies\textsuperscript{[SS94; VTJ11]}. For example, Linux inherits Unix's DAC to assign permission bits to system subjects such as processes and system objects such as files, while allowing the owners of the subjects (or objects) to set the permission bits discretionarily. Modern Linux systems further enhance the security model using MAC such as SELinux\textsuperscript{[Seld]}. For example, the following SELinux policy rule allows Apache server process to read and write HTTP-server-related files.

\begin{verbatim}
allow apache_t httpd_sys_content_rw_t:file {read write}
\end{verbatim}

IT administrators can enforce strict policy to control the accesses to critical and sensitive resources.

\section{Related Work in Security Policy Development}

The security of computer systems highly depends on the quality of security policies. A well-developed security policy can block malicious attacks and protect normal operations from being affected, while a poor security policy may either be too permissive (allowing malicious attacks) or too strict (blocking benign operations)\textsuperscript{[PT02]}. Developing a good security policy is a non-trivial task. It requires policy engineers to have both the security expertise of defense against attacks and domain knowledge of normal functionality of the target system being protected. Furthermore, security policy development in practice is not an
one-time effort. It requires continuous refinements to cover emerging new attacks and updated new functionality. This development process is typically manual, time-consuming and also error-prone if policy engineers are inexperienced or have to manage a considerably large policy. Previously, quantitative studies such as [Woo04] have shown that firewall rules in real-world enterprise systems are often mis-configured and violate well-known security principles.

Over years, a plethora of research efforts have been made in the field of security policy development and analysis. Theoretically, security models such as Bell-LaPadula model[BL73], Biba model[Bib77], Clark-Wilson model[CW87] were proposed as principles (e.g., no read-up, no write-down) of security policy development. After various specific security mechanisms have been implemented and deployed, more practical analysis techniques have been proposed to analyze security policies in different forms. In general, we categorize them into two categories: (1) Network-based Security Policy and (2) Host-based Security Policy.

2.2.1 Network-based Security Policy

To help human policy engineers and IT administrators better understand security policies, multiple policy management tools also provided visualization and interaction for analyzing complex firewall policy rule set. [May00] proposed a firewall analysis tool to allow network administrators to easily discover and test a firewall policy, based on parsing network topology and configuration files. [ASH03; ASH04] proposed a firewall policy advisor to simplify the management of firewall rules by providing automatic discovery of firewall policy anomalies and anomaly-free policy editing. [Gol06] applied data mining techniques to analyze and manage firewall policy rules, including policy rule generalization, decaying rule identification and anomaly detection. [MK05] focused on analyzing the rule chains in iptables, the firewall system in Linux kernel. [Wuu07] built an intrusion pattern miner for Snort IDS rules using labeled malicious network traffic. [KO04] proposed a visualization system to monitor Snort IDS logs with statistical analysis to identify falsely-developed firewall policy rules.

2.2.2 Host-based Security Policy

Multiple access control languages and mechanisms have been implemented in operating systems, such as Access Control List (ACL)[Wic90], Extensible Access Control Markup Language (XACML)[Mos05], Security Enhanced Linux (SELinux)[McC04] and AppArmor[Bau06]. Several work proposed verification and management for XACML, a type of security policies written in XML format[God03; Lor03; HB08]. A series of research efforts have been made to analyze SELinux policy, including formal verification[Zan04; Hic10; Ala08; Sas06], integrity measurement of policy-enforced information
2.3 New Challenges in Security Policy Development

With continuous evolution of security challenges and analysis techniques, we believe that the security policy development will also evolve and new techniques could be leveraged to address new challenges in this development process. In this dissertation, we specifically focus on developing and analyzing security policies for new types of attacks and new security mechanisms, by applying new techniques in program analysis and machine learning to the security field.

2.3.1 New Attacks

With the increasing prevalence and complexity of various (potentially vulnerable) computer and network systems, new attacks keep emerging in the same scale. Unlike previous ad-hoc attacks, modern attacks can be quickly developed, systematically managed and widely distributed with the help of well-designed attack frameworks and toolkits, such as Blackhole[Sop13], Metasploit[Met]. This poses a new challenge for the defense side to catch up the speed of new attacks, since it is usually manual and time-consuming to analyze vulnerabilities and attacks in order to develop new security policies. It is necessary to automate the security policy development targeting new attacks. (See Chapter-3).

2.3.2 New Security Mechanisms

As mobile devices such as smartphones have become an indispensable part of people's daily life, new security mechanisms are developed to protect the security and privacy of mobile devices. For example, the most popular mobile operating system, Android is equipped with multiple security mechanisms including permission model, Linux sandboxing and Security-Enhanced Android (SE-Android) MAC[Anda]. Particularly, SEAndroid is a new MAC mechanism introduced into Android since 4.3, ported from SELinux[Sea]. Due to Android's new architecture, SEAndroid security policy has to be written from scratch. With continuous new features and attacks in Android ecosystem, it is non-trivial to refine the policy to adapt to the updates. (See Chapter-4).
2.3.3 New Techniques

In parallel to the advances in computer security, techniques in other fields such as program analysis [Nie99], software testing in software engineering and big data analytics [ZE11] in machine learning have evolved dramatically as well. We envision new possibilities of leveraging new techniques in these fields to help better address the security policy development. Meanwhile, we are also facing new challenges on finding the semantic connections between different fields and adapting various techniques in the context of security policies. (See symbolic execution in Chapter-3, semi-supervised learning in Chapter-4 and test-driven domain knowledge collection in Chapter-5).

In the following chapters, we will discuss in more details on how we address the above new challenges.
A script-based attack framework is a new type of cyber-attack tool written in scripting languages. It carries various attack scripts targeting vulnerabilities across different systems. It also supports fast development of new attack scripts that can even exploit zero-day vulnerabilities. Such mechanisms pose a big challenge to the defense side since traditional malware analysis cannot catch up with the emerging speed of new attack scripts. In this chapter, we propose MetaSymploit, the first system of fast attack script analysis and automatic signature generation for a network Intrusion Detection System (IDS). As soon as a new attack script is developed and distributed, MetaSymploit uses security-enhanced symbolic execution to quickly analyze the script and automatically generate specific IDS signatures to defend against all possible attacks launched by this new script from Day One. We implement a prototype of MetaSymploit targeting Metasploit, the most popular penetration framework. In the experiments on 45 real attack scripts, MetaSymploit automatically generates Snort IDS rules as signatures that effectively detect the attacks launched by the 45 scripts. Furthermore, the results show that MetaSymploit substantially complements and improves existing Snort rules that are manually written by the official Snort team.
3.1 Introduction

Over the years, with rapid evolution of attacking techniques, script-based attack frameworks have emerged and become a new threat[Exp; Sop13; Met; Sec]. A script-based attack framework is an attack-launching platform written in scripting languages, such as Ruby and Python. Such framework carries various attack scripts, each of which exploits one or more vulnerabilities of a specific application across multiple versions. With the high productivity of using scripting languages, attackers can easily develop new attack scripts to exploit new vulnerabilities.

To launch an attack, an attacker runs an attack script on the framework remotely. By probing a vulnerable target over the network, the attack script dynamically composes an attack payload, and sends the payload to the target to exploit the vulnerability. The attack framework also provides many built-in components with APIs of various attack functionalities to support rapid development of new attack scripts. Once a zero-day vulnerability is found, a new attack script can be quickly developed and distributed in hacking communities, where other attackers even script kiddies can directly download the new script to launch attacks exploiting the zero-day vulnerability.

A well-known example of the script-based attack frameworks is Metasploit[Met], the most popular Ruby-based penetration framework. It has more than 700 attack scripts targeting various vulnerable applications on different operating systems (OSes). It also provides built-in components for creating new attack scripts. Metasploit was originally developed for penetration testing using proof-of-concept scripts. But with years of improvements, it has become a full-fledged attack framework. Unfortunately, as an open source project, Metasploit can be easily obtained and used by attackers for illegal purposes. For example, it was reported that the well-known worm “Conficker” used a payload generated by Metasploit to spread[Con]. A Metasploit attack script was immediately distributed after a zero-day vulnerability was found in Java 7[Rag12]. A four-year empirical study shows real malicious network traffic related to Metasploit on a worldwide scale. Moreover, the study shows that many Metasploit attack scripts are used by attackers almost immediately after the scripts are distributed in hacking communities[RSD].

When a new attack script is distributed and captured by security vendors, the traditional approach to defend against it is to first set up a controlled environment with a vulnerable application installed. Then security analysts repeatedly run the script to exploit the environment over a monitored network, collecting a large number of attack payload samples, and finally extract common patterns from the samples to generate IDS signatures.

However, with the attack framework, new attack scripts can be quickly developed and distributed to exploit the latest vulnerabilities. This poses a great challenge that the traditional approach can hardly catch up with the release speed of new attacks, due to the time-consuming process of
setting up test environments and analyzing attack payload samples. In our evaluation (Section 5), we observe that even the latest Snort IDS rules written by security analysts cannot detect many Metasploit-based attacks.

In this chapter, we propose MetaSymploit, the first system of fast attack script analysis and automatic IDS signature generation. As soon as a new attack script is distributed, MetaSymploit quickly analyzes the attack script and automatically generates IDS signatures of its attack payloads, thereby providing defense against new attacks launched by this script from Day One. Particularly, MetaSymploit gives the first aid to zero-day vulnerabilities whose security patches are not available while the attack scripts that exploit them are already distributed.

Specifically, MetaSymploit leverages symbolic execution while enhancing it with several security features designed for attack script analysis and signature generation. By treating environment-dependent values as symbolic values, MetaSymploit symbolically executes attack scripts without interacting with actual environments or vulnerable applications, thus substantially reducing the time and cost of the analysis. With path exploration of symbolic execution, MetaSymploit also explores different execution paths in an attack script, exposing different attack behaviors and payloads that the script produces under different attack conditions.

To generate signatures of attack payloads, instead of analyzing large volumes of payload samples, MetaSymploit keeps track of the payload composing process in the attack script during symbolic execution. MetaSymploit uses symbolic values to represent variant contents in a payload (e.g., random paddings), in order to distinguish constant contents (e.g., vulnerability-trigger bytes) from variant ones. When the script sends a composed payload to launch an attack, MetaSymploit captures the payload’s entire contents, extracts constant contents as patterns and generates a signature specific to this payload.

In a case study, we implement a security-enhanced symbolic execution engine for Ruby, develop MetaSymploit as a practical tool targeting Metasploit, and generate Snort rules as IDS signatures. Particularly, instead of heavily modifying the script interpreters, we design a lightweight symbolic execution engine running on unmodified interpreters. This lightweight design can keep pace with the continuous upgrades of the language syntax and interpreter (e.g., Ruby 1.8/1.9/2.0). Therefore, our design supports analyzing attack scripts written in different versions of the scripting language.

We evaluate MetaSymploit using real-world attack scripts. We assess our automatically generated Snort rules by launching attacks using 45 real-world Metasploit attack scripts from exploit-db.com, including one that exploits a zero-day vulnerability in Java 7. Our rules successfully detect the attack payloads launched by the 45 scripts. Furthermore, we also compare our rules with the official Snort rule set written by security analysts, and have three findings: (1) the official rule set is incomplete and 23 of the 45 attack scripts are not covered by the official rule set; (2) for the scripts covered by the
3.2 BACKGROUND

official rules, our rules share similar but more specific patterns with the official ones; (3) our studies also expose 3 deficient official rules that fail to detect Metasploit attacks. Therefore, MetaSymploit is a helpful complement to improve the completeness and accuracy of existing IDS signatures to defend against attack scripts.

In summary, we make three major contributions:

1. We point out the security issues of script-based attacks, and propose a scalable approach called MetaSymploit that uses security-enhanced symbolic execution to automatically analyze attack scripts and generate IDS signatures for defense.

2. We implement a security-enhanced symbolic execution engine for Ruby and develop a practical tool for the popular Metasploit attack framework. Our tool can generate Snort rules to defend against newly distributed Metasploit attack scripts from Day One.

3. We demonstrate the effectiveness of MetaSymploit using recent Metasploit attack scripts in real-world attack environments, and also show that MetaSymploit can complement and improve existing manually-written IDS signatures.

3.2 Background

We first give the background of how an attack script works. Generally, when an attack script runs on top of an attack framework, the script performs four major steps to launch an attack. (1) The script probes the version and runtime environment of the vulnerable target over the network. (2) Based on the probing result and the script’s own hard-coded knowledge base, the script identifies the specific vulnerability existing in this target. The knowledge base is usually a list containing the information (e.g., vulnerable return addresses) of all targets that this script can attack. (3) Then the script dynamically composes an attack payload customized for this target. (4) Finally, the script sends the payload to the target to exploit the vulnerability.

Depending on the attack strategy and vulnerability type, different scripts may have different attack behaviors when performing these steps. For example, a brute-force attack may keep composing and sending payloads with guessed values until the target is compromised, while a stealthy attack may carefully clean up the trace in the target’s log after sending the payload.

Among these steps, composing and sending an attack payload are the key steps of launching an attack. An attack payload is typically a string of bytes composed with four elements: (a) special and fixed bytes that can exploit a specific vulnerability; (b) an arbitrary shellcode that attackers choose to execute after the vulnerability is exploited. The shellcode content is usually variant, especially
when obfuscated; (c) random or special paddings (e.g., NOP 0x90) that make the payload more robust; (d) other format bytes required by network protocols.

With the help of the rich libraries of scripting languages and the built-in components provided by the attack framework, an attack script can call APIs of related libraries or components to help it perform each step, especially composing an attack payload.

As an example, Listing 3.1 shows a Ruby code snippet extracted from a real Metasploit attack script exploiting a vulnerable application called Arkeia. In the example, the script defines two methods. `exploit` is the main method that performs the major steps to launch the attack. `prep_ark5` is one of the payload composing methods. When the script runs on Metasploit, it first connects to the target over the network (Line 2), and then probes the target’s version (Line 4). Here both `connect` and `probe_ver` are API methods of a built-in network protocol component. Based on the version, it calls the corresponding method to start composing the attack payload specific to the target (Lines 5-9).

When `prep_ark5` is called, the payload is first assigned by the shellcode component, which
returns a configured shellcode (Line 18). Note that the shellcode can be freely chosen and obfuscated. The shellcode component offers several different shellcodes for different purposes. Then the payload is appended («) with several contents (Lines 19-23). rand_alpha generates random alphabet padding to not only extend the payload to the required size of the network protocol, but also introduce more randomness for evasion. The concrete bytes represent some assembly code that will jump to the shellcode (e.g., "\xeb\xf9" and "\xe9" are two JMP instructions). pack("V") converts the integer to bytes as the offset of one JMP. get_target_ret is another attack framework API that queries the script’s knowledge base (omitted here due to space limit, please refer to [Typ]) to retrieve the exploitable return address based on the target version, which can hijack the control flow\(^1\) (Line 22). After the payload is composed, the script first sends a preamble packet to the target, followed by the attack payload packet to exploit the vulnerability (Lines 11-13).

Popular attack frameworks provide plenty of built-in components covering various network protocols, OSes, and offering different shellcodes and NOP paddings, which enable attackers to quickly develop new attack scripts to exploit different targets. Furthermore, advanced attackers can create even sophisticated attack scripts, which have multiple execution paths performing different attack behaviors and payloads. Some of them may be triggered only under certain attack conditions.

Therefore, the traditional approach that requires both controlled environments and vulnerable applications is not scalable for analyzing attack scripts. Since different attack scripts target different applications and OSes, it is costly and time-consuming to obtain every application (let alone the expensive commercial ones) and set up environments for every OS. It is even harder to create different attack conditions to expose different attack behaviors and payloads in sophisticated attack scripts.

\(^1\)In [Typ], the exploitable return address actually points to a POP/POP/RET instruction sequence, which is a typical SEH-based attack to hijack control flow in Windows.
3.3 MetaSymploit

In this section, we first state the problem and assumptions we focus on, and then give an overview of MetaSymploit, followed by the detailed techniques in its two core parts.

3.3.1 Problem Statement and Assumptions

Problem Statement. We focus on the problem caused by script-based attack frameworks and their attack scripts: how to provide an automated mechanism that can analyze and defend against newly distributed attack scripts. Particularly, the mechanism should be time-efficient in order to address the security issues caused by two major features of attack scripts: a large number of scripts with wide-ranging targets, and fast development and distribution of new scripts that can be directly used to exploit zero-day vulnerabilities.

Assumptions. We assume that both script-based attack frameworks and attack scripts are available from either public or underground hacking communities. As soon as a new attack script is distributed, it can be immediately captured and analyzed. We also assume that the scripting languages used by attack frameworks are general-purpose object-oriented scripting languages, such as Ruby and Python. In reality, sectools.org lists 11 most popular attack tools[Sec] in the public community. 8 of them are Ruby/Python-based attack frameworks. Most of them are actively maintained with frequent updates of new attack scripts.

3.3.2 MetaSymploit Overview

Given an attack script, the goal of MetaSymploit is to quickly analyze fine-grained attack behaviors that the script can perform, and automatically generate specific IDS signatures for every attack payload that the script can compose, providing a fast and effective defense against attacks launched by this script. To achieve this goal, MetaSymploit leverages symbolic execution and enhances it with a number of security features designed for attack scripts analysis and signature generation.

Symbolic execution\(^2\) is a program analysis technique that executes programs with symbolic rather than concrete values. When executing branches related to symbolic values, it maintains a path constraint set and forks to explore different execution paths. By using symbolic execution, MetaSymploit has three advantages to achieve fast analysis and defense against attack scripts: (1) analyzing scripts without requiring actual environments or vulnerable targets, (2) exploring different execution paths to expose different attack behaviors, (3) using symbolic values to represent variant contents in attack payloads to ease the extraction of constant patterns.

\(^2\)For more background of symbolic execution, please refer to [Kin76]
Figure 3.1 MetaSymploit consists of two major parts drawn in grey. (The arrows show the workflow of an attack script analysis.)

Figure 3.1 shows the architecture of MetaSymploit, which consists of two major parts, the symbolic execution layer (SymExeLayer) and the signature generator (SigGen). Given an attack framework, SymExeLayer is built upon the framework. It reuses the framework's execution facility while extending the framework interface to support symbolic execution of attack scripts. When a script is symbolically executed, SymExeLayer captures all attack behaviors and payloads that the script can perform and compose. After the symbolic execution is done, SigGen takes the captured results as inputs. It extracts constant patterns by parsing the contents of the attack payloads. It also analyzes the attack behaviors to derive the semantic contexts that describe the extracted patterns. Finally, SigGen combines the patterns and the contexts to generate IDS signatures for this attack script.

More specifically, three key techniques are developed to realize the functionalities of SymExeLayer and SigGen, respectively. As shown in Figure 3.1, SymExeLayer consists of (1) Symbolic API Extension. It extends the APIs of both the attack framework and the scripting language to support symbolic values and operations. Notably, it extends the APIs related to environments/targets and
variant payload contents to return symbolic values. (2) **Behavioral API & Attack Constraint Logging.** It records critical API calls that represent attack behaviors. It also logs path constraints of symbolic values related to environments and targets. Both logs will be used for deriving pattern context (described later). (3) **Output API Hooking.** It hooks various output APIs that are used to send attack payloads, in order to capture complete payload contents for extracting constant patterns.

SigGen consists of (1) **Constant Pattern Extracting.** By parsing the payload contents, it extracts constant patterns that can represent the payload. Constant patterns include fixed contents, fixed lengths of contents, and fixed offsets of the contents in the format. (2) **Pattern Refining and Consolidating.** It refines patterns by distinguishing critical patterns from common benign bytes and trivial patterns. It also avoids generating duplicated signatures by examining repeated patterns. (3) **Pattern Context Deriving.** In order to describe what the extracted pattern represents, it analyzes the logs of behaviors and constraints to derive the semantic context of the pattern.

To illustrate the workflow of MetaSymploit, we revisit the script in Listing 3.1. First, SymExeLayer takes the script as input and symbolically executes it. The script calls a number of symbolic-extended APIs, including `probe_ver`, `shellcode` and `rand_alpha`. Instead of returning a concrete number, `probe_ver` assigns `version` a symbolic integer representing the target version. `shellcode` and `rand_alpha` return symbolic strings to represent all possible shellcodes and random paddings, respectively. Meanwhile, `probe_ver` indicates the probing behavior. SymExeLayer logs it as one attack behavior. SymExeLayer also logs the path constraint `version==5` since it indicates that the Line 6 branch is taken only under the attack condition that the target version is 5. In contrast, when symbolic execution forks to explore Line 8, SymExeLayer logs the negated constraint `version! =5`.

When executing `prep_ar5`, SymExeLayer logs `shellcode`, `rand_alpha`, and `get_target_ret`, since these APIs indicate a typical attack behavior of composing a stack overflow payload. Note that because `get_target_ret` is a call with a concrete argument, SymExeLayer uses the underlying framework to execute it normally to get the concrete return address value. On the other hand, SymExeLayer symbolically extends the `«` API to support appending symbolic strings. Finally, when the composed payload is sent, the hooked output API `sock.put` captures the complete payload contents.

SigGen then analyzes the payload contents and the behavior & constraint logs to generate signatures. Listing 3.2 shows one Snort rule generated by SigGen. The `content` is the byte pattern extracted from the constant bytes in the payload composed in Lines 20-22. The first 8 bytes are two JMP instructions and the last 4 bytes are the return address. The `pcre` is a regular expression matching the entire payload packet, including constant bytes and random paddings. `content` provides general fast matching, while `pcre` provides more precise matching. The `msg` shows the
pattern context. The target version is derived from the `version==5` constraint. The behavior and the meaning of the patterns are derived from the logged behavioral API calls. The `msg` gives more insights that guide security analysts to use the signature to protect vulnerable application of specific version.

### 3.3.3 Symbolic Execution Layer

This section explains more details about the three techniques of SymExeLayer that extend the attack framework to perform symbolic execution and attack logging.

#### 3.3.3.1 Symbolic API Extension

The key point of performing symbolic execution on attack scripts is to treat all variant values involved in the attack launching process as symbolic values, so that all possible attack variations can be covered. Since attack scripts use APIs to operate variant values, we extend the variant-related APIs of both the scripting language and the attack framework with symbolic support.

The variant-related APIs can be further divided into two categories: direct and indirect. Direct-variant-related APIs always return variant values. There are two major types in this category, (1) the APIs probing external environments/targets, (2) the APIs generating random payload contents. In both cases, we replace the original APIs with our symbolic-extended ones, which directly return symbolic values when called. As a result, the first type of APIs skips probing the actual environment/target, such as `probe_ver` in the example. Such skipping makes MetaSymploit scalable and efficient, since there is no need to prepare different environments or applications when analyzing different scripts. For the second type, as the payload content is a string of bytes, the APIs use symbolic values to represent any variant bytes, such as `shellcode` and `rand_alpha`. Hence, we can clearly distinguish concrete contents from symbolic contents in one payload. In addition, every symbolic value is assigned with a label showing what it represents based on its related API, such as `sym_ver`, `sym_shellcode`, and `sym_rand_alpha`. Note that SymExeLayer uses these labels to keep the semantics of the values, rather than relying on variable names, which can be freely decided by attackers.

Indirect-variant-related APIs return variant values only when their arguments are variant values. Such case typically happens in the operations of some primitive classes such as String, Integer, and some payload composing operations. In SymExeLayer, we extend such APIs by adding the logic of handling symbolic arguments. If the arguments are concrete, the APIs execute the original logic and return concrete values as normal. If the arguments are symbolic, the APIs switch to the symbolic handling logic, which propagates the symbolic argument in accord with the API functionality, and
returns a symbolic expression. In Listing 3.1, for a concrete string argument, the symbolic-extended « appends it as normal. For a symbolic argument, it holds both the original string and the new appended symbolic one in order and returns them as one symbolic string expression.

3.3.3.2 Behavioral API & Attack Constraint Logging

Since symbolic execution is a general program analysis technique, in order to provide additional security analysis of attack scripts, for every execution path, we keep a log recording both critical API calls that reflect attack behaviors and path constraints that represent the attack condition when exploring each execution path.

Behavioral API Logging. As mentioned in Section 2, attack scripts use APIs provided by the language library and the attack framework to launch attacks. In the analysis, it is critical to capture the API calls that perform the detailed attack behaviors during the launching process. There are two major types of behavioral APIs, network protocol APIs and payload-related APIs. By logging the first type, we are able to capture all the interactions between the attack script and the target. By logging the second type, we know exactly how a payload is composed and keep track of its detailed format and contents.

In practice, given an attack framework, we build a knowledge base collecting the APIs from the libraries and components that provide network protocols and payload-related operations. During execution, SymExeLayer identifies behavioral APIs and logs them while keeping the API call sequence in the execution path. Note that we also log the arguments and return values of the APIs, especially for payload-related APIs, whose return values may be a part of the payload contents.

Attack Constraint Logging. In symbolic execution, path constraints are the set of branch conditions involving symbolic values in one execution path. When encountering a new symbolic branch condition, symbolic execution consults a constraint solver to decide which branch(es) is feasible to take, and adds the new branch constraint into the path constraint set. If both branches are feasible to take, the execution path forks into two paths to explore both branches[Kin76].

In attack scripts, we focus on the constraints related to environments and targets. We regard these constraints as attack constraints because different symbolic conditions that they represent typically indicate different attack conditions reflecting the probing results of environments or targets, therefore leading to different execution paths that compose different payloads in consequence. In the example, `version==5 ? prep_ark5 : prep_ark4`.

Recall that the APIs that probe external environments and targets are symbolic-extended. The symbolic return values of these APIs carry the labels showing what external source they represent. When executing a symbolic branch condition, we check if any symbolic value with external-source
3.3. METASYMPLOIT

label is involved. If so, we log the corresponding constraint. In the example, when `version==5` is executed, we find that `sym_ver` is an external source, and thus log the constraint.

In summary, this behavior & constraint logging provides a fine-grained analysis report that saves the time-consuming work for security analysts. More importantly, the behaviors and constraints logged in each execution path can be further parsed to derive the semantic context for the extracted patterns (discussed in Section 3.4.3).

### 3.3.3.3 Output API Hooking

After an attack script finishes composing an attack payload, the script sends the payload as a network packet to the target to exploit the vulnerability. This payload sending step is the exact point of launching an attack. In order to capture the complete content of the attack payload for pattern extraction, we hook the output APIs that are used by attack scripts for sending payload.

Starting from the network layer to the application layer in the OSI model, we keep a list of the output APIs and their corresponding network protocols from both the scripting language’s own network library and the built-in components of the attack framework.

We symbolically extend the output APIs by overriding their functionality from sending real network packets to dumping the entire packets locally. By doing so, the entire network flow sent from the attack script can be dumped throughout the execution. To keep the semantic context of each dumped packet, we associate them with the behavior & constraint log of that execution path, so that later the payload packets can be identified and the extracted patterns can be correlated with the context derived from the log. In the example script, the hooked `sock.put` dumps two packets. With the associated log, we identify the payload packet for pattern extraction.

Note that as a part of the network protocol APIs, the output APIs are also behavioral APIs that need to be logged. In addition, we also include the corresponding network protocols in the log. Later during signature generation, the log gives a clear view of which network protocol is used, and therefore SigGen can apply the correct packet format when parsing the packet contents.

### 3.3.4 Signature Generator

Given the dumped payload packets and the logs as inputs, SigGen includes three techniques to generate signatures.

#### 3.3.4.1 Constant Pattern Extracting

In order to generate a signature that can detect a payload packet, it is necessary to extract a set of constant patterns that always stay the same across different variations of the payload. Specifically,
there are three constant patterns that can be extracted: fixed-content pattern, fixed-length pattern and fixed-offset pattern. For ease of explanation, we first present the formal form of a dumped symbolic attack payload.

Recall that an attack payload is a string of bytes containing both concrete contents (e.g., fixed vulnerable return address) and variant contents (e.g., arbitrary shellcode, random padding). When a payload is being composed during the symbolic execution of the attack script, we use symbolic strings to represent variant contents and use extended APIs to perform symbolic string operations, while keeping concrete values and operations as normal. Thus the dumped payload packet is a big symbolic string composed of a sequence of substrings, where each substring is either a concrete byte string or a symbolic string by itself. Formally, $S_{sym} = (s_1s_2\ldots s_i\ldots s_n)$, where $s_i \in \{S_{con}\} \cup \{S_{sym}\}$. In addition, we also embed $<sym\_label, length>$ in $S_{sym}$ to keep the semantics and the possible length of the string, where the length is either a concrete or symbolic integer. As an example, Listing 3.3 shows the contents of the payload when being composed in Lines 18-23 of the example script.

The final dumped payload is the same as the one in Line 23.

Fixed-content pattern. This pattern has two types, either a simple byte string or a regular expression (regex). When parsing the payload, for each concrete substring, we extract it as a byte string pattern, such as the 12-byte string in the payload of Line 23. For each symbolic substring, if it can be matched by a regex, we extract the regex as a fixed pattern. If no regex is found, we move on to the next substring. In practice, we keep a mapping between regex-matchable symbolic labels and the regexes. Currently, we mainly focus on using regexes on payload paddings to achieve precise matching. For instance, we map the symbolic label `sym\_rand\_alpha` to a regex pattern `[a-zA-Z]`.

Fixed-length pattern. In some cases, although the contents may vary, their lengths stay the same. Such case typically happens when using padding to meet the size requirement. To achieve precise
matching, SymExeLayer keeps track of the payload length during the composition. When parsing the payload, we identify the symbolic substrings with fixed lengths and extract them as patterns. When executing the example script in SymExeLayer, we keep updating the payload length. Later when parsing \texttt{<sym\_rand\_alpha, 2917>} in the dumped payload, we produce a length-quantified regex \texttt{[a-zA-Z]{2917}} as shown in Listing 3.2.

**Fixed-offset pattern.** Due to the format of some network protocols, some payloads can be located only after certain offsets of the packets. For instance, some FTP-based attack packets have regular FTP commands, followed with overlong paths as payloads to launch overflow attacks. In such cases, since the network protocol of the output API is logged, by applying the packet format of the protocol, we extract the offset of the payload, which is a pattern for precise matching of the payload location.

### 3.3.4.2 Pattern Refining and Consolidating

As MetaSymploit automatically generates signatures in a large scale, there are two requirements for the quality of the signatures. First, we should avoid generating signatures only having patterns of common benign bytes or patterns of trivial bytes/regexes, which may otherwise cause false positive. Second, we should avoid generating duplicated signatures with the same pattern set, which may cause useless redundancy and confuse the IDS.

**First requirement.** When a payload is finally sent through the output API, common benign bytes are introduced by network protocols as concrete substrings in the payload packet, including default protocol bytes (e.g., “Content-Type:text/html”) and delimiter bytes (e.g., “\r\n”). To identify them, for each protocol, we keep a list of benign bytes. Based on the packet format, we examine the concrete substrings to search for the occurrences of benign bytes. If found, we strip the benign part and focus on the rest bytes for pattern extraction.

In addition, it is also important to avoid generating signatures only using trivial patterns such as too short byte string or too general regex patterns. Thus, we set a threshold of minimum byte string length (e.g., \texttt{>= 10}) and a list of critical regexes (e.g., NOP regex \texttt{[\x90]*}). Given a set of extracted patterns, we generate signatures only if we can find at least one pattern whose length is above the threshold or whose regex is critical. Note that both the threshold and the critical regex list are adjustable. Security analysts can also define different thresholds and lists for different network protocols.

**Second Requirement.** Recall that SymExeLayer explores different execution paths in an attack script and dumps payloads in each path. Sometimes, two paths may differ only in a branch that is irrelevant to the payload content, thus finally composing the same payloads. Furthermore, two attack scripts may also share the same patterns. To consolidate the same patterns from different payloads into
one signature, we keep a key-value hash map where each key is a pattern set and each value is a set of different payloads with the same pattern set. When a new payload is parsed, if its pattern set already exists in the hash map, we add this new payload, particularly its behavior & constraint log into the corresponding value set. The payloads and the logs in one set are analyzed together to generate only one signature.

3.3.4.3 Pattern Context Deriving

Apart from pattern extraction, it is equally important to provide the context of the patterns. The pattern context shows the insight into the attack script, such as what attack behavior and attack payload the patterns represent. It also gives security analysts the guidance on how to use the patterns, such as which target version and what OS environment the patterns can be used to protect.

Therefore, we analyze the behavior & constraint log to derive the pattern context. Since attack behaviors are captured as behavioral APIs in the log, we derive the context by translating the behavioral APIs into human-readable phrases. Some APIs have straightforward names, which can be simply translated into the description phrase (or even directly used), such as `probe_ver => Version Probing`. Others may not be intuitive. Particularly, certain behavior cannot be shown from a single API but a series of API calls. In such case, we group these API calls together as one behavioral pattern. When such pattern is found in the log, we translate it into the matched behavior name, such as `shellcode + get_target_ret => Stack Overflow`.

Sometimes, sophisticated attack scripts may have unprecedented behaviors whose APIs do not match any patterns. In such cases, we keep the derivable context while highlighting underived behavioral APIs in the log to help security analysts discover new attack behaviors. In fact, we use this technique in our prototype to collect patterns.

In regard to attack constraints, since the involved symbolic values represent attack conditions of each execution path, we retrieve the external source names in the symbolic labels and bind them with the conditions derived from the constraints (e.g., `Target Version: 5`).

Finally, when both the extracted pattern set and the derived context are ready, SigGen combines two together and generates a signature, which can be used to detect the payloads associated with this specific pattern set.

3.4 Implementation

We implement a prototype of MetaSymployt as a practical analysis tool targeting the Ruby-based attack framework Metasploit. Given a Metasploit attack script, our tool quickly analyzes it and auto-
matically generates Snort rules as signatures that can defend against this specific script. Particularly, we developed a lightweight Ruby symbolic execution engine designed for attack script analysis. Powered by the engine, we build SymExeLayer on top of the launching platform of Metasploit. In this section, we first describe how the engine is designed and then explain how to adapt the engine for Metasploit.

### 3.4.1 A Lightweight Symbolic Execution Engine for Ruby

Traditionally, developing a symbolic execution engine requires heavy modification of the interpreter, which causes great engineering effort since Ruby has multiple active versions and interpreters (e.g., 1.8/1.9/2.0). However, we discover a new way to design a lightweight engine without modifying the interpreter. The engine is developed purely in Ruby (9.3K SLOC) as a loadable package compatible with multiple versions of Ruby. Thus it supports analyzing attack scripts written in different versions. Specifically, our engine has two modules: (1) a symbolic library that introduces rich symbolic support into Ruby; (2) a symbolic execution tracer that performs symbolic execution based on the actual script execution.

#### 3.4.1.1 Library of Symbolic Support

The symbolic library realizes the functionality of *Symbolic API Extension*. The library introduces symbolic classes to hold symbolic values (e.g., SymbolicString, SymbolicInteger). To be transparent to attack scripts, we develop the same APIs in the symbolic classes as their concrete counterparts. On the other hand, we also extend indirect-variant-related APIs in the concrete classes to support handling symbolic arguments, so that concrete and symbolic objects can operate with each other.

Notably, SymbolicString class plays the key role in representing attack payloads. To hold the contents, SymbolicString has an internal ordered array, where each item is either a concrete substring, or a symbolic substring with the `<sym_label, length>` embedded. When a SymbolicString API is called, it first checks whether the original concrete operation is still applicable to the concrete substrings. If so, the API uses the original logic in String to operate the concrete substrings. Otherwise, the API treats the contents as symbolic substrings, and processes the internal string array as symbolic expressions. When a symbolic-extended String API is called with symbolic arguments, it handles concrete and symbolic substrings in the same way as above and returns a SymbolicString object.

Later when SymExeLayer is integrated with Metasploit, we further include the symbolic-extended APIs of Metasploit into the symbolic library.
3.4. IMPLEMENTATION

3.4.1.2 Symbolic Execution Tracer

The symbolic execution tracer transforms normal script execution into symbolic execution. It also realizes the functionality of Behavior & Constraint logging. To this end, we develop three techniques based on three advanced language features in Ruby (and Python³).

(1) Fine-grained execution tracing. This technique traces the symbolic execution line-by-line in an attack script. It keeps track of every method call. It also explores different paths when executing branches. We develop it by enhancing a language feature called Debug tracing function with Control Flow Graph (CFG).

    **Debug tracing function** is a step-by-step execution tracing facility used for debugging such as Ruby's `set_trace_func` (Python's `sys.settrace`). It captures three major events, line, call, return. The line event shows the number of the current executing line. The call/return event shows the name of the method being called/returned. Every time an event happens, Debug tracing function suspends the execution and calls a registered callback function for further event analysis.

    We develop our callback function using the CFG of the attack script. Since the CFG holds both the source code and the control flows, it offers rich semantics for analyzing the execution details when parsing every event. When a line event happens, we locate the current line’s source code in the CFG. Then we retrieve all call sites in the current line, which will be matched with the following call/return events happening in this line. Particularly, this tracing mechanism can log behavioral API calls when they are found in the call sites.

    Our callback function also handles branches to explore different paths. When the line event reaches a symbolic branch, we evaluate the branch source code and consult a constraint solver for both true and false branch constraints. If a solution exists, we concretize the symbolic branch condition to guide the interpreter to the desired branch (explained next). If both branches can be satisfied, we fork the script execution process into two processes to trace both branches. Otherwise, if no solution is returned, we terminate the execution process. Particularly, if attack constraints are found, the callback function would perform constraint logging.

(2) Runtime symbolic variable manipulation. This technique leverages the Runtime context binding language feature to manipulate the runtime values of symbolic variables. In particular, it inspects the values of attack payloads during composing. It also concretizes symbolic branch conditions to guide branch execution.

    **Runtime context binding** can inspect and modify the runtime states of the script, such as Ruby’s `Binding` and Python’s `inspect`. It provides a context object that binds the runtime scope of the

---

³The techniques can also build an engine to analyze Python-based attack scripts, since Ruby and Python share many language features.
current traced code. The callback function can use this object to access all variables and methods in the scope of the traced code.

The first use of context is to inspect the runtime value of an attack payload when it is being composed. When a variable is detected to be assigned by payload composing APIs, the callback uses context to keep track of its value. The callback then logs the inspected values together with the payload composing APIs in the behavior log.

The second use of context is to guide symbolic branch execution. Since the interpreter cannot move forward with a symbolic condition, when the constraint solver returns a solution, for each symbolic variable in the condition, we use context to temporarily replace the symbolic value with the solved concrete value to guide the interpreter to the desired branch. Later when the line event shows that the branch is taken, we recover them back to their symbolic form. Recall the version==5 in Listing 3.1. Since version is symbolic value, we temporarily replace its value with 5 to explore one branch, and uses a non-5 value for the other branch.

3.4.2 Adaptation for Metasploit

To analyze Metasploit attack scripts, we adapt the engine and the six techniques in both SymExeLayer and SigGen to work with the APIs provided by Metasploit and its built-in components.

The current prototype is based on Metasploit version 4.4 (released in Aug 2012). We select the top 10 most popular built-in components in Metasploit: Tcp, Udp, Ftp, Http, Imap, Exe, Seh, Omelet, Egghunter, Brute. The first 5 are popular network protocol components. The next 4 are used to attack Windows systems. Exe can generate exe file payloads. Seh can create SEH-
based attacks. Both Omelet and Egghunter can compose staged payloads. The last Brute can create bruteforce attacks. These components cover 548 real attack scripts carried in Metasploit. By examining the APIs provided by the launching platform and these components of Metasploit, we perform three steps to adapt the engine for SymExeLayer and SigGen.

First, in the symbolic library, we apply symbolic API extension to the environment-related APIs such as tcp.get, ftp.login, http.read_response, and variant-payload-content-related APIs such as rand_text, make_nops, gen_shellcode. The library also replaces the output APIs such as ftp.send_cmd, http.send_request with our local-dumping APIs. When the script calls these APIs during symbolic execution, SymExeLayer redirects the calls to the symbolic-extended APIs.

Second, to equip the symbolic execution tracer with behavior & constraint logging ability, we build a knowledge base collecting behavioral APIs such as http.fingerprint, gen_egghunter and keep a mapping between APIs and their behavior meaning for pattern context deriving. We also keep a list of symbolic labels for identifying attack constraints.

Third, based on the standards of the protocols and the implementation of the built-in components, we add the packet formats and common benign bytes of the five network protocols into the knowledge base. For instance, we develop specific parsers to parse payloads embedded in HTTP headers and FTP commands.

Note that both the API extension and the knowledge base are one-time system configuration. Since Metasploit components and their APIs are relatively stable for compatibility with various attack scripts, once they are collected and supported by MetaSymploit, newly distributed attack scripts that rely on these components can be directly supported and automatically analyzed.

3.5 Evaluation

We conduct our evaluation on an Intel Core i7 Quad 2.4GHz, 8GB memory, Ubuntu 12.10 machine. We run MetaSymploit based on Metasploit 4.4, using the official Ruby 1.9.3 interpreter. We evaluate our approach from three perspectives: (1) the percentage of real-world attack scripts that can be analyzed by MetaSymploit’s symbolic execution; (2) the effectiveness of our automatically generated signatures to defend against real-world attacks; (3) the difference between our automatically generated rules and official Snort rules.

4The listed API names are abbreviated due to space limits. Note that Metasploit uses payload to represent shellcode. We use shellcode as a more general term to avoid confusion with attack payloads.
3.5. EVALUATION

Table 3.1 The distribution of different situations in the symbolic execution of the 548 Metasploit attack scripts.

<table>
<thead>
<tr>
<th>Category</th>
<th>Num</th>
<th>Percentage</th>
<th>Require Manual Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatically Executed</td>
<td>509</td>
<td>92.88%</td>
<td>No</td>
</tr>
<tr>
<td>Symbolic Loop</td>
<td>9</td>
<td>1.64%</td>
<td>Avg 10 LOC/per script</td>
</tr>
<tr>
<td>Non-Symbolic-Extended API Call</td>
<td>12</td>
<td>2.19%</td>
<td>Avg 3 LOC/per script</td>
</tr>
<tr>
<td>Obfuscation &amp; Encryption</td>
<td>13</td>
<td>2.37%</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Multi-threading</td>
<td>3</td>
<td>0.55%</td>
<td>Not Supported</td>
</tr>
<tr>
<td>Bug in Scripts</td>
<td>2</td>
<td>0.37%</td>
<td>2 LOC in each script</td>
</tr>
<tr>
<td><strong>Total Coverage</strong></td>
<td></td>
<td>Auto 92.88%</td>
<td>All 96.90%</td>
</tr>
</tbody>
</table>

3.5.1 Coverage Testing with Symbolic Execution Engine

We first evaluate whether MetaSymploit can symbolically execute various attack scripts. We use MetaSymploit to analyze all 548 real attack scripts created with the top 10 popular Metasploit components. As the result shown in Table 1, 509 scripts (92.88%) are automatically executed by MetaSymploit in the symbolic mode without any manual modification of the scripts. Different attack conditions in the scripts are explored. The attack payloads are captured and Snort rules are generated.

In terms of analysis cost, since MetaSymploit reuses the launching platform of Metasploit on the official Ruby interpreter, the symbolic execution has almost the same speed as that Metasploit executes attack scripts normally (less than one minute on average). In fact, since the environment-related APIs are symbolic-extended, the time for real network communication is saved. Furthermore, signatures are generated in less than 10 seconds.

Among the remaining 39 scripts that MetaSymploit cannot automatically deal with, we encounter five main situations that deserve more discussion.

**Loop with Symbolic Condition.** We find that 9 scripts have conditional loops whose symbolic conditions cannot be solved by constraint solvers, which may cause infinite looping. As a common problem in classical symbolic execution, some previous approaches proposed using random concrete values to replace symbolic conditions to execute loops\([GK]\). However, in our case, doing so may affect the precision of the payload contents. Other approaches such as LESE\([Saxb]\) specifically handle loops, which we plan to explore in future work.

Currently, after manual analysis, we find that there are two cases of using the loops: byte-by-byte modifying a symbolic string whose length is a symbolic integer, and performing repeated attack steps in a brute-force attack. In the first case, since the string length is not concrete, the looping rounds cannot be decided. However, we find no matter how many rounds are, the looping result
is still a symbolic string. Therefore, we replace the loop code that operates the symbolic string with a new symbolic string to represent the looping result (10 LOC per script on average), while propagating the symbolic label and logging the loop information for further investigation.

In the second case, the Brute component provides an API that checks whether the target is compromised or not. It is typically used as a while loop condition. The loop keeps attacking the target until the API returns that the target is compromised. Since in our case the API returns a symbolic value as the target status, to avoid infinite looping, we set a counter with an upper bound in the extended version of this API, to control the looping rounds. If there are payloads and logs captured inside the loop, the differences between each round are analyzed to identify the constant patterns.

**Non-Symbolic-Extended API Call.** Due to the time limitations, other than the top 10 components, we have not symbolically extended other APIs in Metasploit. We detect 12 scripts that call the non-extended APIs related to assembly translating and encoding the payloads. Since very few APIs are involved, we decide to modify each of them individually at this time, and extend the entire components in future work. To handle these API calls, since SymbolicString supports payload content processing, when applicable to the concrete substrings, we allow the APIs to operate on the concrete parts, while preventing them from using the symbolic substrings, which may otherwise cause runtime errors. When the API operates on a pure symbolic string with no concrete substrings, we replace the API calls by creating new symbolic strings to represent the results of the API calls (3 LOC per script on average).

**Obfuscation & Encryption.** There are 13 cases with complicated obfuscation and encryption on the payload, where payload content processing is not feasible. Since the output of these operations is completely random, there is no constant pattern that can be extracted from the obfuscated or encrypted payload. Defending against obfuscation and encryption is an open question, which is beyond the scope of signature-based defense.

**Multi-threading.** Handling multi-threading is an advanced topic in symbolic execution. Existing research[SA] explored the possibility by extending symbolic execution to handle multi-threaded programs. Currently, due to only 3 cases related to this situation, we plan to address this issue in future work.

**Bug in Scripts.** Interestingly, during the testing, we also discover 2 scripts with bugs that hang the execution when the script is generating a specific assembly code that jumps to the shellcode. From this result, we see that our approach is also useful for the purpose of finding bugs in attack scripts.

In summary, the percentage of scripts that are automatically handled is 92.88%. If the manually modified scripts are included, the percentage reaches 96.90%. 


3.5. EVALUATION

3.5.2 Effectiveness Validation using Real-world Attacks

To evaluate whether the automatically generated Snort rules can effectively detect real attacks, we use Metasploit attack scripts to attack 45 real-world vulnerable applications. These applications are acquired from exploit-db.com, a popular hacking website collecting attack scripts and free vulnerable applications. In all, there are 45 free vulnerable applications available in the website, with 45 corresponding Metasploit scripts. They include Java 7, Adobe Flash Player 10, Apache servers 2.0, Firefox 3.6, RealPlayer 11, multiple FTP servers such as Dream FTP, ProFTPD, VLC player 1.1, IRC servers and some less popular web-based programs.

We first use MetaSymploit to analyze the 45 attack scripts and automatically generate Snort rules. Then we set up two virtual machines, with one running Metasploit to simulate the attacker and the other running the vulnerable application as the vulnerable target. For each script, we choose two different shellcodes to launch two real attacks. To expose the entire attack flow, we allow the attack to compromise the target, and use Snort IDS 2.9.2 with our generated rules to detect attack payloads. Note that due to the limited available versions of the applications, we focus on the rules of the attack payloads that target the application versions that we are able to obtain.

The initial results show that except the HTTP-based ones, all attack payload packets with both two types of shellcodes are correctly detected. Recall that our rules are based on the constant patterns of the payload, variant parts such as shellcodes do not affect the detection. But for Apache server attacks and Firefox attacks, our rules fail to catch the attack packets because the order of each HTTP header field is different from the one in our rules. Since the order of the HTTP header fields is not enforced by RFC definition, the extracted patterns from the HTTP header cannot be simply put into the signature in sequence. Therefore, we further improve our HTTP parser to handle each header field separately, to enable order-insensitive pattern matching. In the second round of testing, the HTTP-based attacks are also correctly detected.

Another interesting case is the Java 7 attack. In late Aug 2012, two days after a zero-day vulnerability in Java 7 was disclosed (CVE 2012-4681), a Metasploit attack script was distributed targeting this vulnerability [Rag12]. At that time, we immediately used MetaSymploit to analyze this attack script and automatically generate a Snort rule based on the malicious jar payload composed by this script, and tested it in our environment. Our rule successfully detected the jar payload. Admittedly, there might be other ways different from the distributed Metasploit script to exploit the vulnerability. Nevertheless, our rule provides the first aid to the vulnerability without available security patch, to defend against attackers who directly use this widely-distributed script to launch attacks.

Apart from the effectiveness evaluation, we also use our rules generated from the 45 attack scripts to monitor normal network traffic, to investigate whether our rules would raise false positives...
3.5. EVALUATION

3.5.3 Comparison with Official Snort Rules

To further assess the quality of the generated rules, we compare the MetaSymploit rules (MRs) of the 45 attack scripts with the recent Official Snort rules (ORs), released in Nov 2012\(^5\). We use CVE number carried in both attack scripts and ORs to match each other. The result is surprising that only 22 attack scripts have corresponding ORs. The rest 23 are not even covered by ORs. This reveals a serious issue that existing defense is still quite insufficient compared to the fast spreading of public attack resources.

For the 22 officially covered scripts, there are 53 MRs and 50 ORs. In MetaSymploit, one script may have multiple rules detecting different payloads for different target versions. Whereas in the official rule set, one vulnerability may also have multiple rules detecting different ways that exploit it. By comparing the patterns in both rule sets, we summarize the result in Figure 3.2. We find that 44 MRs share patterns with 21 ORs. Specifically, 6 MRs and 6 ORs share the same content byte

\[^5\text{snortrules-snapshot-2922.tar.gz on www.snort.org/snort-rules/}\]
Table 3.2 The list of three Metasploit attack scripts which evade the detection from 3 Official Snort Rules

<table>
<thead>
<tr>
<th>Metasploit Script Name</th>
<th>CVE</th>
<th>Failure Reason of Official Snort Rules Missing Metasploit Payloads</th>
<th>Official Rule SID</th>
</tr>
</thead>
<tbody>
<tr>
<td>badblue_ext_overflow.rb</td>
<td>2005-0595</td>
<td>The <code>http_uri</code> flag restricts the pattern searching in one header field, thus missing the Metasploit payload located in the following fields.</td>
<td>3816</td>
</tr>
<tr>
<td>sascam_get.rb</td>
<td>2008-6898</td>
<td>The <code>flow</code> pattern is set to check packets sent to the client while our pattern context shows the Metasploit payload is sent to the server.</td>
<td>16715</td>
</tr>
<tr>
<td>mozilla_reduceright.rb</td>
<td>2011-2371</td>
<td>The <code>content</code> byte pattern is wrong since it includes two variant bytes, which are randomly generated in the Metasploit payload.</td>
<td>19713</td>
</tr>
</tbody>
</table>

patterns. 4 MRs and 4 ORs share the same pcre regex patterns. Notably, 35 MRs have specific content that are matched with 11 ORs’ general pcre. This is because the pcre regexes are generalized by security analysts based on large volumes of samples, while the content bytes (usually including vulnerable return addresses) are generated based on every attack payload of the scripts. An example is shown in Appendix A. Although in this case, the MR set is a subset of the OR one, we argue that as our goal is to defend against specific attack scripts, MRs give more insight of the attack payloads with more precise matching. Meanwhile, there are 5 MRs and 26 ORs with no pattern shared. This is because some vulnerabilities can be exploited in different ways, and the ORs have more patterns defined by analysts, while Metasploit scripts usually choose one way to exploit one vulnerability. Nevertheless, we still find that 2 scripts have 5 MRs whose patterns are not seen in ORs, which complement the OR set.

Besides, we also load the 50 ORs into Snort to test whether they can detect attacks launched by the 22 attack scripts. Interestingly, the result shows that only 17 scripts’ attack payloads are detected, while no alert is raised for the other 5 scripts. 2 scripts\(^6\) are missed due to the lack of OR patterns as we mentioned above. The other 3 scripts, which have 3 MRs, are supposed to be detected by 3 corresponding ORs. After comparing these rules, we find the 3 ORs have some deficiencies that cause this inconsistent detection results. We list the detailed information of the 3 scripts and the deficiencies of the 3 ORs in Table 2. Note that some deficiencies are actually caused by inaccurate use of Snort rule flags such as the `http_uri`, `flow`. We find them by comparing these flags with the pattern context (e.g., Behaviors) in our rules. We have reported these discoveries to the official Snort team.

In sum, these results show that even the official Snort rules written by security analysts are

---

\(^6\) adobe_flash_sps.rb, mozilla_mchannel.rb
incomplete and tend to be error-prone. MetaSymploit serves as a useful tool to complement and augment the existing IDS signatures by improving the completeness and the accuracy.

3.6 Discussion

**Scenarios of using MetaSymploit signatures.** As shown in the comparison (35 MRs vs 11 ORs), due to different pattern extracting mechanisms, ORs have less rules with more general patterns, while MRs have more rules with more specific patterns. It is possible that as the number of attack scripts is increasing, more and more signatures will be generated. If all signatures are loaded into the IDS, this may slow down the matching speed.

However, we argue that unlike ORs are used for general detection, MetaSymploit signatures should be used in two typical scenarios, which do not require loading all MRs in an IDS. First, as the goal of MetaSymploit is to provide quick defense against newly distributed attack scripts, the typical way of using our signatures is to give first aid to the vulnerable application without available patches to prevent attackers especially script kiddies using the new scripts to launch attacks (e.g., the Java 7 case). When the vulnerability is patched or the application is upgraded, our signatures can be removed from the IDS. Second, as the pattern contexts are embedded with the signatures, security analysts only need to deploy the signatures whose contexts are related to the protected environment or the protected target version, to avoid loading irrelevant signatures which may slow down the matching speed of the IDS.

**Limitations.** MetaSymploit inherits the limitations of classical symbolic execution. As we mentioned in Section 5.1, our current prototype requires manual analysis on handling complex symbolic loops. Recent approaches propose to use bounded iteration\cite{God}, search-guiding heuristics\cite{Xie} and loop summary\cite{Saxb, GL} to address the loop issue. In MetaSymploit, different loop cases of attack scripts may require different techniques. For example, bounded iteration can be applied to handle loops of bruteforce attacks. Loop summaries can summarize the post-loop effect on symbolic payload contents. Search-guiding heuristics can help target payload-related loops to avoid getting stuck in irrelevant loops.

Apart from loops, path explosion is a more general issue related to performance and scalability. Too many paths in an attack script may prolong the analysis and delay the defense. In addition, it is possible that different paths in a script finally lead to the same attack payload output. Exploring these paths incurs extra efforts of pruning redundant payloads. Several techniques such as equivalent state tracking\cite{Boo}, state merging\cite{Kuz} and path partitioning\cite{Qi} have been proposed to mitigate the path explosion issue. We plan to incorporate these techniques into MetaSymploit to avoid exploring paths that would compose redundant payload contents.
The limitations of constraint solvers may also affect the effectiveness of path exploration. Currently, we use Gecode/R[Gec] for solving integer/boolean constraints and HAMPI[Kie] for solving string constraints. In case when encountering complicated constraints (e.g., a non-linear constraint), the solvers cannot decide which branch to take. For the sake of completeness, we conservatively explore both branches, while marking the path constraints as uncertain in the log, which require more investigation by security analysts. Due to this fact, we regard our prototype as an assistant tool to reduce the workload of analysts, so that they only need to focus on complicated ones when facing large numbers of new attack scripts.

We envision possible attacks directly against MetaSymploit’s defense mechanism. As MetaSymploit rules stick to the patterns in the distributed attack scripts, it is possible that experienced attackers may modify the distributed one to create new script variants without releasing them, which may evade the detection of MetaSymploit rules. Besides, experienced attackers may also try to exploit the limitation of symbolic execution when developing new scripts, such as introducing complex loops, non-linear constraints or even obfuscating the script code. However, both cases are non-trivial. They require advanced attack developing techniques, which are usually time-consuming and slow down the speed of developing and launching new attacks. In other words, with MetaSymploit, we raise the bar of the skill level and the time cost for developing and launching new attacks.

3.7 Related Work

Signature Generation. There has been a lot of work on automatic signature generation for malware defense. From the perspective of attacks, Autograph KK04, Polygraph New and Hamsa Li automatically generate worm signatures by extracting invariant contents from the network traffic of worms. Particularly, these approaches are based on the observation that even polymorphic worms have invariant contents that can be used as signature patterns. In MetaSymploit, we have the same observation when analyzing constant and variant payload contents composed by attack scripts. On the other hand, these approaches require collecting large amounts of malicious network traffic to identify invariant contents. However, this process is usually time-consuming and cannot provide quick defense against new attacks. In contrast, MetaSymploit does not need to collect any network traffic but only attack scripts, thus largely reducing the time of performing analysis and providing defense.

From the perspective of vulnerabilities, Vigilante Cosb, ShieldGen Cui and Bouncer Cosa analyze vulnerable applications and their execution traces to generate signatures to block exploit inputs that can trigger the vulnerability. Brumley et al. [Brua; Brub] also provide the formal definition
of vulnerability-based signatures and propose constraint-solving-based techniques to generate such signatures. Elcano\cite{Elcano} and MACE\cite{MACE} further use protocol-level concolic exploration to generate vulnerability-based signatures. Notably, program analysis techniques such as symbolic execution play an important role in these approaches as well as in MetaSymploit. But unlike these approaches, MetaSymploit only analyzes attack scripts without requiring the presence of vulnerable applications, thus avoiding the cost of obtaining various vulnerable applications or preparing various testing environments.

**Symbolic Execution.** Symbolic execution has been actively applied for security purposes\cite{Sch}. BitBlaze\cite{Son} is a binary analysis platform based on symbolic execution. SAGE\cite{God} uses dynamic symbolic execution to detect vulnerabilities in x86 binaries. EXE\cite{Cad} and AEG\cite{Avg} generate malicious inputs and exploits by symbolically executing vulnerable applications. Moser et al.\cite{Mos} explore multiple execution paths for malware analysis. Since our analysis target, attack script is quite different from host-based binary level malware, the techniques proposed in these approaches such as memory inspection, system call analysis are not adaptable in our case.

Symbolic execution for scripting languages is still at early stage, due to the diversity of different kinds of scripting languages and various purposes of applications. Most work focuses on the web-based scripting languages, such as JavaScript\cite{Saxa}, PHP\cite{Art}, and Ruby on Rails\cite{CF} web frameworks. Since these approaches are specifically designed for testing web applications (e.g., finding XSS and SQL Injection vulnerability), they are not applicable for analyzing general attack scripts and attack frameworks that target various vulnerable applications on different OS environments.

In particular, little work has been done for the symbolic execution of general-purpose scripting languages, such as Ruby and Python. PyStick\cite{Nm12} is an automated testing tool with input generation and invariant detection for Python. It is different from our purpose of using symbolic execution for security analysis. Bruni et al.\cite{Bru11} propose a library-based approach to develop symbolic execution. However, it uses only the dynamic dispatching feature, which limits symbolic execution only in primitive types. This limited functionality is insufficient for practical use.

### 3.8 Summary

Script-based attack frameworks have become an increasing threat to computer security. In this chapter, we have presented MetaSymploit, the first system of automatic attack script analysis and IDS signature generation. MetaSymploit leverages security-enhanced symbolic execution to analyze attack scripts. We have implemented a prototype targeting the popular attack framework Metasploit. The results have shown the effectiveness of MetaSymploit in real-world attacks, and also the practical
use in improving current IDS signatures.

3.9 Acknowledgements

We would like to thank the conference reviewers and our shepherd Dr. Prateek Saxena for their feedback in finalizing this work. This work is supported by the U.S. Army Research Office (ARO) under a MURI grant W911NF-09-1-0525, and also supported in part by an NSA Science of Security Lablet grant at North Carolina State University, NSF grants CCF-0845272, CCF-0915400, CNS-0958235, CNS-1160603.
Mandatory protection systems such as SELinux and SEAndroid harden operating system integrity. Unfortunately, policy development is error prone and requires lengthy refinement using audit logs from deployed systems. While prior work has studied SELinux policy in detail, SEAndroid is relatively new and has received little attention. SEAndroid policy engineering differs significantly from SELinux: Android fundamentally differs from traditional Linux; the same policy is used on millions of devices for which new audit logs are continually available; and audit logs contain a mix of benign and malicious accesses. In this chapter, we propose EASEAndroid, the first SEAndroid analytic platform for automatic policy analysis and refinement. Our key insight is that the policy refinement process can be modeled and automated using semi-supervised learning. Given an existing policy and a small set of known access patterns, EASEAndroid continually expands the knowledge base as new audit logs become available, producing suggestions for policy refinement. We evaluate EASEAndroid on 1.3 million audit logs from real-world devices. EASEAndroid successfully learns 2,518 new access patterns and generates 331 new policy rules. During this process, EASEAndroid discovers eight categories of attack access patterns in real devices, two of which are new attacks directly against the SEAndroid MAC mechanism.
4.1 Introduction

Operating system integrity relies on the correctness of 1) trusted computing base (TCB) code and 2) access control policy protecting the TCB code and OS resources. It is generally impractical to verify the correctness of OS code in commodity systems. Therefore, mandatory access control (MAC) policy is often used as a fallback when the security of the software inevitably fails [Los98].

SELinux [Seld] is the most notable MAC policy framework widely used in practice. Security Enhanced Android [SC13] (known simply as SEAndroid) is a recent port of SELinux to the Android platform. However, while Android is based on a Linux kernel, the runtime environment is vastly different than existing Linux distributions for commodity PCs. This difference resulted in a complete redesign of the MAC policy rules, with several new object classes (e.g., for Android’s binder IPC).

As with SELinux, SEAndroid policy development is a challenging task, requiring many iterations of refinement to be ready for commercial deployment. For example, Google introduced a very permissive SEAndroid policy into Android version 4.3 and did not enable enforcement. Version 4.4 enabled enforcement, but the policy was still very permissive, containing only a few system daemons. Finally, Android version 5.0 provides a much more robust (but not perfect) version of the policy. Additionally, major smartphone vendors need to customize Google’s base SEAndroid policy for their devices to add additional protections against known attacks.

SEAndroid policy refinement is currently a very manual process that typically involves analyzing audit logs to identify proposed changes. There are two general approaches to SEAndroid policy refinement. The first approach is to develop a least privilege [SS75] policy (also known as a “strict” policy in SELinux terminology) and monitor audit logs for access patterns that should be allowed. The second approach is to begin with a more permissive policy and refine the policy to prevent (or contain) privilege escalation attacks using suspicious activity in audit logs. Each approach has disadvantages. If the policy is too strict, it will hurt the usability of deployed real-world devices. If the policy is too permissive, it will allow attacks. As a result, smartphone vendors use a combination of these two approaches.

The goal of our research is to significantly reduce the manual effort required to refine SEAndroid policy using audit logs. Audit log analysis is challenging for several reasons. First, audit logs are collected from millions of real-world devices, and purely manual analysis is impractical. Second, the platform (OS, hardware) is rapidly evolving, requiring frequent revisions to the security policy. Third, the audit logs contain both benign access patterns and malicious access patterns. Since audit logs contain malicious access patterns, existing SELinux tools such as audit2allow are error prone. Fourth and last, the functionality of both benign applications and malicious exploits is continually changing, requiring frequent reassessment of the deny/allow boundary.
In this chapter, we present Elastic Analytics for SEAndroid (EASEAndroid) as the first large-scale audit log and policy analytic platform for automatic policy analysis and refinement of SEAndroid-style MAC policy. Our key insight is that the policy refinement process can be modeled and automated using semi-supervised learning [Cha06], a popular knowledge-base construction technique [Car10b; Don14]. We apply EASEAndroid to a database of 1.3 million audit logs from a major smartphone vendor. The audit logs are from real-world devices running an Android 4.3 over the entire year of 2014.¹ EASEAndroid correctly discovers 336 new benign access patterns and automatically translates these access patterns into 51 policy rules. The generated rules are consistent with rules manually added by policy analysts. EASEAndroid also finds 2,182 new privilege-escalation access patterns, and discovers two new types of attacks in the wild directly targeting SEAndroid MAC mechanism itself.

This work makes the following contributions:

• **We propose EASEAndroid, a semi-supervised learning approach for refining MAC policy at large scale.** Our approach scales to millions of audit logs that contain a mix of benign and malicious access patterns. While we focus on SEAndroid, the approach is more broadly applicable to type enforcement (TE) MAC policy.

• **We implement and deploy EASEAndroid in a production environment of a major smartphone vendor.** The implementation generates policy refinements, and discovers new Android attacks, while significantly reducing the workload of policy analysts.

• **We evaluate EASEAndroid on 1.3 million audit logs from real-world devices.** Using this dataset, EASEAndroid successfully generates 331 policy rules as a refinement. It also learns 2,182 new malicious access patterns, including two attacks directly targeting SEAndroid. With the help of EASEAndroid, this is the first large-scale study on real-world malicious access patterns in Android devices.

The remainder of this chapter proceeds as follows. Section 4.2 provides background on SEAndroid and semi-supervised machine learning. Section 4.3 defines the problem addressed in this work. Section 4.4 describes the EASEAndroid design. Section 4.5 evaluates EASEAndroid against a large database of real-world audit log denials. Section 4.6 discusses limitations. Section 4.7 overviews related work. Section 4.8 concludes.

¹See Appendix A for more details about our audit log collection.
4.2 Background

4.2.1 SELinux and SEAndroid

SEAndroid is a port of SELinux’s type enforcement (TE) MAC policy to the Android platform [Sea]. As such, SEAndroid enforces mandatory policy on system-level operations between subjects and objects (e.g., system calls) [Seld]. In general, processes are regarded as subjects, whereas files, sockets, etc. are objects in different classes. A security context label is assigned to subjects (or objects) that share the same semantics. Traditionally, the subject label is called a domain, and the object label is called a type (nomenclature from DTE [Bad95]). A policy rule defines which domain of subjects can operate which class and type of objects with a set of permissions, such as open, read, write [LS01]. For example,

```
allow app app_data_file:file {open read}
```

allows processes with the app domain to open and read file class objects assigned the app_data_file type. In addition to allow rules, SELinux provides neverallow rules to define policy invariants for malicious accesses that should never be allowed. These rules are enforced at policy compile-time and are necessary due to the complexity of the SELinux policy language.

SEAndroid extends SELinux’s policy semantics to support Android-specific functionality, including mediation of Binder IPC and assigning security contexts based on application digital signatures. The goal of SEAndroid is to reduce the attack surface and limit the damage if any flaw or vulnerability is exploited causing privilege escalation [SC13]. This goal is accomplished by confining the capabilities of different privileged Android applications and system daemons.

The Android platform is vastly different than traditional Linux distributions, therefore the SEAndroid policy rules were created from scratch. While the regularity of Android’s UNIX-level interactions results in a policy that is significantly less complex than the example SELinux policy for PCs, the SEAndroid policy is still nontrivial and error prone. It requires careful understanding of subtle interactions between different privileged processes. In practice, policy development requires continual manual refinement based on audit logs.

An audit log captures security labels and system calls of the operations that are not explicitly allowed by a rule. As shown in Listing 4.1, a denied operation generally has three entries with epoch timestamps. Log entries with type=1400 record the denied permission (e.g., entrypoint), the security labels of the subject (source), called scontext, and the object (target), called tcontext, as well as the object’s class, called tclass (e.g., file). Log entries with type=1300 record the system call and the subject’s process information, including the executable file path. Log entries with type=1302 record the object information (e.g., the file name).
4.2. BACKGROUND

<table>
<thead>
<tr>
<th>type=1400 msg=audit(1399587808.122:14):</th>
</tr>
</thead>
<tbody>
<tr>
<td>avc: denied { entrypoint } pid=285 comm=&quot;init&quot;</td>
</tr>
<tr>
<td>scontext= u:r:init:s0</td>
</tr>
<tr>
<td>tcontext= u:object_r:system_file:s0 tclass=file</td>
</tr>
<tr>
<td>type=1300 msg=audit(1399587808.122:14):</td>
</tr>
<tr>
<td>syscall=11(execve) success=no exit=-13</td>
</tr>
<tr>
<td>items=1 ppid=1 pid=285 uid=0 gid=0</td>
</tr>
<tr>
<td>comm=&quot;init&quot; exe=&quot;/init&quot; subj=u:r:init:s0</td>
</tr>
<tr>
<td>type=1302 msg=audit(1399587808.122:14):</td>
</tr>
<tr>
<td>item=0 name=&quot;/system/etc/install-recovery.sh&quot;</td>
</tr>
<tr>
<td>inode=3799 dev=b3:10 mode=0100755</td>
</tr>
<tr>
<td>ouid=0 ogid=0 obj=u:object_r:system_file:s0</td>
</tr>
</tbody>
</table>

Listing 4.1 A denied access event example recorded at the epoch time 1399587808.122 in an audit log. It consists of three entries: labels & permission (1400), syscall & process info (1300), object info (1302).

Traditionally, analysts develop and refine a policy by manually analyzing audit logs. Sometimes, the security labels in type=1400 are used directly to create a new allow rules using a tool called audit2allow. However, blindly using this tool may increase attack surface. In some cases, the existing labels are too coarse-grained or semantically inappropriate. Therefore, policy refinement usually consists of the creation and modification of both security labels and policy rules. Once the policy is refined by analysts, it is pushed to users’ devices through a secure over-the-air (OTA) channel, similar to antivirus signature updates.

4.2.2 Semi-Supervised Learning

Semi-supervised learning is a type of machine learning that trains on both labeled\(^2\) data (used by supervised learning) and unlabeled data (used by unsupervised learning) [Cha06]. It is typically used when labeled data is insufficient and expensive to collect, and a large set of unlabeled data is available. By correlating the features in unlabeled data with labeled data, a semi-supervised learner infers the labels of the unlabeled instances with strong correlation. This labeling increases the size of labeled data set, which can be used to further re-train and improve the learning accuracy [Zhu08]. This iterative training process is commonly referred to as bootstrapping. Semi-supervised learning is popular for information extraction and knowledge base construction. Examples include NELL [Car10a; Car10b], Google Knowledge Vault [Don14].

We hypothesize that the process of developing and refining SEAndroid policy is analogous to semi-supervised learning. Human analysts encode their knowledge about various access patterns

\(^2\)Note that labeled/unlabeled data are different from security labels.
4.3 Problem

Refining SEAndroid policy is more challenging than refining SELinux policy. Existing SELinux tools such as `audit2allow` are severely limited in their ability to help policy analysts. This task has the following challenges.

**C-1:** Consumer devices produce millions of audit log entries. Policy analysts cannot practically analyze audit manually. A solution must automate or semi-automate the audit log analysis.

**C-2:** Real-world audit logs contain a mixture of benign and malicious accesses. Classifying log entries as benign or malicious is a central design challenge. It is often difficult to classify an access in isolation. Instead, the analysis must look at the broad context of the access, as well as the contexts of related known accesses.

**C-3:** Target functionality is not static. The set of benign and malicious applications continues to evolve as new software and malware is developed. This makes correlations between subjects difficult to identify. For example, benign software may access new resources. In contrast, new malware may adopt privilege escalation code from known malware.

We now define two terms to clarify the discussion in the remainder of this work.

**Definition 4.1 (Access Event).** An access event is the access control event that causes the three audit log entries described in Section 4.2. These log entries may result from a policy denial, or an `auditallow` policy rule, which allows but logs the access.

Note that this definition does not include allowed accesses that are not contained in the audit log.

Since audit logs are collected for millions of devices, the logs contain many duplicate access events. For the purposes of audit log analysis, it is useful to abstract the salient details of access events into an access pattern.
Definition 4.2 (Access Pattern). An access pattern is a 6-tuple \((\text{sbj}, \text{sbj\_label}, \text{perm}, \text{tclass}, \text{obj}, \text{obj\_label})\). Many access events may map to the same access pattern.

Here \(\text{sbj}\) refers to a concrete subject such as an Android application or system binary. Binaries carried inside an application are generalized as the application. \(\text{obj}\) refers to concrete objects such as file paths and socket names. In some cases, we group over-specific files that share filesystem semantics as one \(\text{obj}\) (e.g., /sdcard). The values of \(\text{sbj}\) and \(\text{obj}\) are derived from the \text{comm}, \text{exe}, \text{pid}, \text{name} values in the \text{type}=1300, 1302 log entries. \(\text{perm}\) and \(\text{tclass}\) are the same as the permission and the object's class in the \text{type}=1400 log entries and policy rules. \(\text{sbj\_label}\) and \(\text{obj\_label}\) are derived from the \text{scontext} and \text{tcontext} values in \text{type}=1400 log entries. For example, the access pattern for the access event in Listing 4.1 is \(("/init", "init", "entrypoint", "file", "/system/etc/install-recovery.sh", "system\_file")\).

Problem Statement: Given 1) a large dataset of new access patterns from audit logs, 2) a small set of known access patterns (e.g., known attacks), and 3) an SEAndroid policy, we seek to a) separate new benign access patterns from new malicious access patterns in the dataset, and b) suggest new rules and refined labels for the policy.

Threat Model and Assumptions: We assume that an audit log is collected from an Android device with a policy loaded in either enforcing or permissive mode. We assume the integrity of audit log contents. We therefore assume that the Linux kernel and audit subsystem is not compromised. However, even if the SEAndroid policy properly confines other system daemons, those daemons may be compromised by the adversary.

4.4 EASEAndroid

Elastic Analytics for SEAndroid (EASEAndroid) is a large-scale audit log and policy analytic platform for automated policy refinement. The novelty of EASEAndroid is that it models the policy refining process as a semi-supervised learning of new access patterns. At a high level, EASEAndroid starts with an initial knowledge base containing existing policy rules and a small set of (potentially manually) identified access patterns. It expands the knowledge base by correlating, classifying, and incorporating new access patterns captured by audit logs. Based on the new knowledge, EASEAndroid suggests policy changes (new rules and new domain and type labels in the context of SEAndroid). As more audit logs become available, EASEAndroid continuously expands the knowledge base and refines the policy.

Figure 5.1 shows the architecture of EASEAndroid. The architecture uses three machine learning algorithms that consider different perspectives of the knowledge base and input audit logs. The
4.4. EASEANDROID

CHAPTER 4. EASEANDROID

Figure 4.1 EASEAndroid consists of four learning components and a policy generator to iteratively learn new patterns and refine policy.

output of these algorithms is fed into a combiner that combines and appends the new knowledge into the knowledge base. This learning process is iterated multiple times until no more new knowledge can be learned from the current audit log input. Finally, the policy generator suggests refinements.

Each machine learning algorithm analyzes a different perspective of the data. The goal of each algorithm is to find semantic correlations between unknown new access patterns and existing knowledge base, in order to classify each new access pattern as benign or malicious.

1. The nearest-neighbors-based (NN) classifier classifies new access patterns based on their relations to known access patterns in the knowledge base. It finds new access patterns that are related to known subjects/objects (e.g., known subjects are updated and perform new access patterns). By treating these known subjects/objects as neighbors of the new access patterns, it classifies the new access patterns based on the majority of their known neighbors.

2. The pattern-to-rule distance measurer calculates the distance between new access patterns and existing policy rules. If a new access pattern is closest to an allow rule, it is classified as benign. If it is closest to a neverallow rule, it is classified as malicious. If the access pattern is not close to either type of rule, it remains unclassified. The pattern-to-rule distance measurer also exposes potentially incomplete rules in existing policy for refinement.

3. The co-occurrence learner considers correlations across access patterns using statistical relations between new and known access patterns that frequently occur together in audit logs. Our intuition is that a benign functionality and malicious attacks often involve a series of
access patterns that are captured together in an audit log. If known and new access patterns occur together, we can use the known ones to infer the classification of the new ones.

Each learner is configured with its own threshold to classify new access patterns independently. However, it is non-trivial to define proper thresholds because too relaxed thresholds could cause false classification, lowering the learning precision, while too strict thresholds could leave potential access pattern candidates unclassified, lowering the learning coverage.

The learning balancer & combiner manages the threshold of each learner, balances the precision and coverage, and combines the classification results from the three learners. It has two modes: (1) an automated mode that uses strict thresholds in each learner to achieve high precision with the cost of less coverage; and (2) a semi-automated mode that relaxes each learner’s threshold to achieve high coverage and relies on a majority vote from the three learners to increase the precision. In practice, the result of semi-automated mode requires analysts verification to control error rate.

Finally, the policy generator takes newly classified access patterns as input from the combiner, to suggest policy refinements in the form of new rules and new security labels. It uses a clustering algorithm to group similar subjects and objects together. The clustering algorithm follows the principle of least privilege by inferring fine-grained labels that can cover and only cover the clustered concrete subjects and objects. The resulting refined policy should be confirmed by policy analysts. The refined policy should also be merged into the knowledge base to analyze new logs after the refined policy is deployed.

The remainder of this section describes each stage of the EASEAndroid architecture in detail.

4.4.1 Nearest-Neighbors-based Classifier

Nearest-neighbors-based (NN) learning is a common technique for classifying an unlabeled instance based on its nearest labeled neighbors within a defined distance [Yia93]. Our intuition of using NN for access pattern classification is two-fold. First, known subjects often perform previously unseen access patterns in audit logs. This scenario often occurs when Android applications and system binaries are updated with new capabilities. Second, some known access patterns are also performed by new subjects. This scenario occurs when certain operations become popular and are copied by other new applications. In practice, some exploit kits and repackaged applications [ZJ12] have been found to share the same set of known malicious access patterns.

These two scenarios cause known subjects and patterns to be semantically connected with new subjects and patterns. EASEAndroid leverages this connectivity as the distance metric to design the NN classifier. When multiple known subjects (or patterns) connect to the same new pattern (or subject), the NN classifier can infer whether the new pattern (or subject) is benign or malicious, based on
Algorithm 1 NN-based Classification of Access Patterns

\[ AP_k \leftarrow \{(s_k, p_k, t_k, o_k) | s_k \in S_k, (p_k, t_k, o_k) \in P_k\} \]
\[ AP_u \leftarrow \{(s_u, p_u, t_u, o_u) | s_u \in S_u, (p_u, t_u, o_u) \in P_u\} \]
\[ AP_c \leftarrow \emptyset \]

**procedure** NN_CLASSIFIER\((AP_k, AP_u, AP_c)\)

- **for each** \((s, p, t, o) \in AP_u\) **do**
  - **if** \(s \in S_k \cap S_u\) and \((p, t, o) \in P_u - P_k\) **then**
    - \(S_{tmp} \leftarrow \text{findAllSbjs}((p, t, o), AP_u)\)
    - **if** IsMajorityKnown\((S_{tmp}, S_k)\) **then**
      - \(AP_c \leftarrow AP_c \cup \text{Classify}((s, p, t, o))\)
    - **end if**
  - **else if** \(s \in S_u - S_k\) and \((p, t, o) \in P_k \cap P_u\) **then**
    - \(P_{tmp} \leftarrow \text{findAllPatterns}(s, AP_u)\)
    - **if** IsMajorityKnown\((P_{tmp}, P_k)\) **then**
      - \(AP_c \leftarrow AP_c \cup \text{Classify}((s, p, t, o))\)
    - **end if**
  - **end if**
**end for**
**end procedure**
**return** \(AP_c\)

The majority of the connected known neighbors. Note that here the observation is with respect to concrete subjects and objects in access patterns. Hence, only a 4-tuple \((s_b j, p e r m, t c l a s s, o b j)\) out of the original 6-tuple is required. For completeness, our implementation still includes \(s_b j_{-}l a b e l\) and \(o b j_{-}l a b e l\) in the dataset, but they are not used in this learner.

Algorithm 1 shows the procedure of the NN classifier. \(AP_k\) collects known 4-tuple access patterns, either benign or malicious. In practice, our \(AP_k\) is a small set containing a few well-confirmed subjects and patterns, used as the initial seed. \(AP_u\) collects all unknown new access patterns from audit logs. To clearly describe the above two cases, we further divide the 4-tuple into \(S\) for all subjects, and \(P\) for the triples \((p e r m, t c l a s s, o b j)\) as partial patterns shared by multiple subjects. \(AP_c\) is the result set of newly classified access patterns.

For each 4-tuple in \(AP_u\), we check if it is a known subject with a new triple (partial pattern), or a new subject with a known triple. In the first case, besides the subject in this 4-tuple, \(\text{findAllSbjs}\) collects all subjects \(S_{tmp}\) that perform (connect) the same new triple in \(AP_u\), including both known and new subjects. Then \(\text{IsMajorityKnown}\) checks if the majority of \(S_{tmp}\) is a set of known subjects from \(S_k\) with the same benign or malicious flag. If so, the new access pattern is classified as benign or malicious accordingly. The second case is done in the same way but using known triples.
4.4. EASEANDROID CHAPTER 4. EASEANDROID

to classify new subjects.

The function $IsMajorityKnown$ uses two empirically defined thresholds $(m, \sigma)$. $m$ determines the minimum required neighbors and $\sigma$ is a percentage for how many known neighbors in $S_{tmp}$ or $P_{tmp}$ are required as a majority. Table 4.1 in the evaluation studies the effects of different threshold values.

From the perspective of machine learning, our NN-based classifier is a type of radius-based near neighbors learning [Ben75], a variant of the common k-nearest-neighbors (kNN). The difference is that kNN is based on the top $k$ neighbors while we find all neighbors within a radius as nearest neighbors (connectivity is the radius in our case).

Note that, it is possible that some access patterns are rarely connected with known ones. Besides, an access pattern could be evenly connected to both known benign and malicious ones. Both cases cause $IsMajorityKnown$ to return false. In this case, the NN classifier leaves the access patterns as unclassified and relies on the following learners to complement the learning process.

4.4.2 Pattern-to-Rule Distance Measurer

EASEAndroid’s second data perspective is the closeness of access patterns to policy rules. Since audit logs record denied accesses that cannot match with an allow rule, it is useful to know how far/close the denied access pattern is from an existing rule. In particular, because policy rules are developed incrementally, they may only cover a subset of permissions or access patterns and miss similar access patterns belonging to the same operation.

A common case of this is an imprecise list of permissions in an allow rule. For example, writing a file not only requires write permission, but also append and sometimes create (in case the file does not exist). Some malicious operations can also be performed using semantically equivalent, but different access patterns.

The pattern-to-rule distance measurer quantifies the difference between access patterns and existing rules. The purpose of this measurer is two-fold. First, pattern-to-rule distance indicates how likely a new access pattern is to be benign or malicious. Second, if an access pattern is very close to policy rule, the policy refinement generator (Section 4.4.5) can update the rule rather than creating new rules from scratch.

The distance measurer uses a metric based on the 1) subject label, i.e., domain, 2) object label, i.e., type, 3) tclass, e.g., file, and 4) permission, e.g., write. Note that all four of these elements are in both the SEAndroid policy allow rules, as well as the 6-tuple representing an access pattern in the audit log. Intuitively, an access pattern is very close to a rule if it shares the same labels and tclasses only with slightly different permissions (e.g., write vs append). The distance increases a
little, but is still close, if a pattern and a rule operate on different but similar tclasses (e.g., file vs dir).

EASEAndroid systematically measures distance using decision trees based on existing policy rules. The distance is defined by the matching depth for a specific access pattern. Decision trees are built as follows.

**Step 1:** For every subject label, find all related rules and follow their semantic order to build a tree skeleton starting from the subject as the root, followed by object labels, tclasses and permissions as nodes in each layer.

**Step 2:** Extend each node with its semantically similar siblings.

Figure 4.2 shows an example decision tree. The black nodes indicate the tree skeleton, which uses rules such as `allow untrusted_app app_data_file:file {open}`, where `app_data_file` is the object node in the second layer and `file` is the tclass node in the third layer and so on. Then each node is extended with its semantic siblings, such as `sdcard_file` in the same group of low sensitive data as app data.

Given an access pattern and a decision tree, the distance measurer walks the decision tree and tries to match the access pattern's subject label, object label, tclass, and permission with each layer. The matching depth indicates how close a pattern is to existing rules. For the example in Figure 4.2, access pattern \( ap_i = \{\text{untrusted_app}, \text{sdcard_file}, \text{dir}, \text{read}\} \) matches the fourth layer. The
distance is computed as follows:

\[ \text{Dist}(ap_i) = \text{TotalLayerDepth} - \text{MatchedDepth}(ap_i) \]

If we define TotalLayerDepth = 4, then Dist(ap_i) = 0, indicating the access pattern is very close to the rule. We create trees for both allow and neverallow rules to compute the distances from both sides.

In practice, the effectiveness of the distance metric depends on the correctness of semantic siblings. Fortunately, the SEAndroid policy development frequently uses semantic groups. A list of permissions, tclasses, and object types are already grouped together in policy source code using macros and attribute [Sele]. These groups form a ground truth for semantic siblings.

Additionally, recall that some existing subject and object labels are coarse-grained (e.g., labels assigned to various objects using wildcard in policy source code). If a pattern matches with a rule with a coarse-grained label, the distance measurer marks the distance as low confidence and relies on the learning balancer & combiner for additional verification (Section 4.4.4).

Finally, note that this technique can be further extended to measure access pattern to access pattern distance. Since the pattern-to-rule distance helps to infer both new patterns as well as identify incomplete rules for refinement, our design considers policy rules and leave the distance between access patterns for future work.

4.4.3 Co-Occurrence Learner

When analyzing a large number of audit logs, some access patterns frequently occur together in many logs. This is because some high-level benign functionality or some popular multi-step attacks consist of a series of access patterns within a time period (typically minutes). The statistics of co-occurrence is a valuable means of correlating access patterns that have different subjects or objects, but share the same group semantics. When a group contains both known and new access patterns, the known access patterns can be used to infer the semantics of the new access patterns. In fact, co-occurrence is popular in natural language processing and knowledge extraction. For example, it is used for finding words that are frequently used together in a specialized domain [BL07].

The co-occurrence of access pattern can be represented using a \( n \times n \) matrix for all \( n \) unique access patterns from the audit logs, as shown below. Each row stores one access pattern \( ap_i \)'s co-occurrence percentage with every other access pattern, denoted in each column. The value \( c_{ij} \) is the percentage of the number of times that \( ap_i \) co-occurs with \( ap_j \) out of the total number of \( ap_i \)'s
4.4. EASEANDROID CHAPTER 4. EASEANDROID

occurrences throughout the logs.

\[ CO_{AP} = \begin{bmatrix}
    a_{p_i} & a_{p_j} & \ldots \\
    1 & c_{i j} & \ldots \\
    c_{j i} & 1 & \ldots \\
    \vdots & \vdots & \ddots & 1
\end{bmatrix}, \]

where \( c_{i j} = \frac{\text{CoOccurNum}(a_{p_i}, a_{p_j})}{\text{TotalOccurNum}(a_{p_i})} \)

When counting the number of co-occurrences, it is important to avoid noise and duplicates. In practice, we use a time frame of 10 minutes to determine whether two access patterns are part of a co-occurrence set. Recall from Section 4.2 that each access pattern has an epoch timestamp. Additionally, when counting the occurrence at the granularity of logs, repeated pairs of co-occurred access patterns in one log are counted only once.

To use this co-occurrence matrix, the learner focuses on the rows with new access patterns. For each \( a_{p_i} \) row, the learner sorts columns and selects the set of known access patterns in columns whose percentage is above a threshold. A majority vote of this known access pattern set determines the classification of the new \( a_{p_i} \) (benign or malicious). On the other hand, the known access pattern rows may also have some highly co-occurred new access pattern columns. However, one known access pattern is usually not enough to classify a new access pattern.

Note that the matrix is not symmetric. \( c_{i j} \) can be different from \( c_{j i} \) due to different total occurrence counts. For instance, some popular known malicious access patterns (e.g., remount/system) can co-occur with multiple less popular new access patterns, because multi-step attacks often use different steppingstones to achieve the final privilege escalation goal.

4.4.4 Learning Balancer & Combiner

Each learner is configured with its own threshold to classify new access patterns independently. However, it is non-trivial to define proper thresholds due to two reasons. On the one hand, if a threshold is too relaxed, it could cause false classification, lowering the learning precision and might further propagate the error to the next iteration of semi-supervised learning. On the other hand, if a threshold is too strict, it could miss potential access pattern candidates and leaves them as unclassified, lowering the learning coverage.

We design the learning balancer & combiner to manage the threshold setting of each learner, balance the precision and coverage, and combine the classification results from the three learners (also called “multi-view learning”[Car10a]). The final combined classification result is added to
the knowledge base and sent to the policy refinement generator. Specifically, we propose two quantifiable methods of achieve the balancing:

**Automated Mode:** Since each learner specializes in one dimension, each learner with a strict threshold can directly contribute its classified access patterns with high precision. For example, we can set a minimum of 10 required known neighbors with a 90% bar for \( IsMajorityKnown \) in NN classifier; \( Dist(ap_i) = 0 \) with fine-grained rules in pattern-to-rule distance measurer; and \( c_{ij} > 0.9 \) with known access pattern set \( \geq 10 \) in co-occurrence learner. The high precision of strict thresholds enables EASEAndroid to be used in an automated mode over multiple iterations of semi-supervised learning. However, with such strict thresholds, some access pattern candidates can be left as unclassified.

**Semi-Automated Mode:** This mode relaxes the thresholds to get more access pattern candidates. It uses a majority vote to choose the candidates shared by at least two learners with the same classification result. However, the third learner must not have conflicting result.

Note that relaxed thresholds can increase the possibility of error propagation. However, if the analysis can tolerate a semi-automated configuration, relaxed thresholds can be used. Here a human analyst can investigate low-confident candidates and input external knowledge for better learning in future.

### 4.4.5 Policy Refinement Generator

Finally, the policy refinement generator translates newly classified access patterns\(^3\) into the final policy form. A key part of the generator is to assign the concrete subjects (\( sbjs \)) and objects (\( objs \)) in the access pattern with appropriate security labels before generating policy rules.

According to the Android Open Source Project, Google provides a baseline definition of security labels for common subjects (e.g., system apps and binaries) and basic objects (e.g., basic files/\( dirs \) in Android file system structure). However, Google recommends that manufacturers replace the generic default labels with fine-grained labels to decrease the attack surface [Sea].

Recall that both the access pattern 6-tuple and the incomplete rules identified by the distance measurer include subject labels and object labels from the existing policy. While some of the labels are coarse-grained, they serve as a baseline to derive fine-grained labels. Specifically, the policy refinement generator takes all access patterns as input and cluster them into groups where each group shares the same 4-tuple (\( sbj\_label, perm, tclass, obj\_label \)). Each group is further clustered by \( sbjs \) and \( objs \) to create subgroups that share the detailed semantics used to derive fine-grained labels.

\(^3\)In practice, the learning process can iterate multiple times with current audit logs, the generator caches all classified access patterns.
Our current generator prototype groups subjects and derives fine-grained subject labels for built-in, vendor, and untrusted applications and binaries separately. The generator also groups file-like objects (e.g., file, dir, blk_file), which comprise the majority of classes. Group is performed using a longest common prefix search on file paths. This optimization helps to derive more fine-grained labels than provided by the general Android filesystem structure. Finally, the generator produces rules in the form of (new_sbj_label, perm, tclass, new_obj_label) as a policy refinement. Note that if access patterns are matched with incomplete rules by the distance measurer, new rules also merge with existing rules’ permissions.

The generator handles benign and malicious patterns separately and generates allow and neverallow rules, respectively. Note that, it is possible that newly generated rules may conflict with existing rules due to incomplete or tightened access control. In such cases, policy analysts manually resolve conflicts (e.g., using auditallow to verify). Nevertheless, EASEAndroid exposes these conflicts with evidence collected through learning, therefore easing the policy refining process.

### 4.5 Evaluation

We implement a prototype of EASEAndroid and evaluate the learning capability and the security effectiveness of EASEAndroid from three perspectives:

1. We evaluate the coverage and precision of the classification result of EASEAndroid, and how they are affected by different threshold settings (Section 4.5.3).
2. We conduct a case study of the policy refinement generated by EASEAndroid, also comparing the generated rules with human-written rules (Section 4.5.4).
3. We further conduct a study on the new malicious access patterns classified by EASEAndroid and discuss several interesting new findings of attacks in the wild (Section 4.5.5).

#### 4.5.1 Environment Setup

We build a prototype of EASEAndroid on an 8-node Hadoop cluster with each node having 8-core Xeon 2GHz, 32 GB memory. We deploy the prototype as an alpha version in a production environment. We use Cloudera Impala as the distributed SQL layer, with 10K SLOC Java as the learning layer. Parallelism is heavily employed for fast analytics. A data set of 1.3 million audit logs used in the following experiments are analyzed by EASEAndroid within 3 hours on average.
4.5.2 Audit Log & Existing Knowledge

Audit Logs & Existing Policy We make use of 1.3 million audit logs over the entire 2014 from real-world devices running Android 4.3. All devices are loaded with an early version of vendor's SEAndroid policy (the policy remained unchanged) in enforcing mode. The policy contains 5094 allow rules and 59 neverallow rules developed by policy analysts. This policy is loaded as existing knowledge into EASEAndroid's knowledge base, used by the pattern-to-rule distance measurer.

The audit logs contain a total of over 14 million denied access events. After eliminating duplicate entries, we identify approximately 145K unique access events and further generalize them into 3530 access patterns. For example, third-party app process ids under /proc/ are generalized as /proc/app_pid in access patterns.

The subjects in the audit logs consist of 113 system (built-in) binaries, 1182 external binaries (e.g., installed by adb), and 626 Android apps, which are captured because they perform system-level operations that do not go through Android framework/Dalvik VM (normal app operations are already allowed by the policy).

Initial Known Malicious Access Patterns In the initial knowledge base, we also prepare a small set of known malicious access patterns as the initial seed to kick off learning. The set contains 9 confirmed exploit kits with their 17 malicious access patterns (e.g., psneuter CVE-2011-1149, Motochopper CVE-2013-2596, vroot CVE-2013-6282 and several exploit apps). Note that we do not have known benign access patterns initially as we rely on the allow rules in the existing policy.

Ground Truth To analyze the classification result of benign and malicious access patterns, we use a later version of human-written policy (6337 allow rules, 94 neverallow rules) as the ground truth. We also consult with experienced policy analysts about the result.

4.5.3 Coverage & Precision of the Classification by EASEAndroid

4.5.3.1 Coverage compared with naive matching

To illustrate the effect of EASEAndroid's learning coverage, we design a naive matching tool as a baseline to compare with EASEAndroid learning when both analyze the same set of new access patterns from the audit logs, as shown in Figure 4.3. The naive matching tool is a dumb access pattern matching tool with no learning capability. It only uses the known subjects and access patterns in the initial knowledge base and can only match new access patterns related to them (existing policy is no use here). In contrast, EASEAndroid starts from the initial knowledge base and keeps expanding the knowledge base.

Some devices are found being rooted and may switch to permissive mode. See Section 4.5.5
Table 4.1 The coverage and precision of EASEAndroid with different threshold settings after comparison with ground truth. The first three columns summarize the overall classification coverage over the 3530 patterns. The following four columns give more details about the percentages of each set of true/false-classified benign and malicious patterns. Row 4 is the threshold setting used in Figure 4.3(b).

<table>
<thead>
<tr>
<th>Threshold Setting</th>
<th>Classified Malicious (TP+FP)</th>
<th>Classified Benign (TN+FN)</th>
<th>Remain Unclassified</th>
<th>True Malicious (TP)</th>
<th>False Malicious (FP)</th>
<th>True Benign (TN)</th>
<th>False Benign (FN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma = 55%, Dist \leq 2, c_{ij} &gt; 0.55$</td>
<td>77.2%</td>
<td>14.0%</td>
<td>8.8%</td>
<td>62.96%</td>
<td>37.04%</td>
<td>58.65%</td>
<td>41.35%</td>
</tr>
<tr>
<td>$\sigma = 65%, Dist \leq 1, c_{ij} &gt; 0.65$</td>
<td>70.0%</td>
<td>11.8%</td>
<td>18.2%</td>
<td>88.73%</td>
<td>11.27%</td>
<td>71.35%</td>
<td>28.65%</td>
</tr>
<tr>
<td>$\sigma = 75%, Dist \leq 1, c_{ij} &gt; 0.75$</td>
<td>65.7%</td>
<td>10.9%</td>
<td>23.4%</td>
<td>91.35%</td>
<td>8.65%</td>
<td>88.92%</td>
<td>11.08%</td>
</tr>
<tr>
<td>$\sigma = 85%, Dist = 0, c_{ij} &gt; 0.85$</td>
<td>63.9%</td>
<td>10.5%</td>
<td>25.7%</td>
<td>96.81%</td>
<td>3.19%</td>
<td>90.81%</td>
<td>9.19%</td>
</tr>
<tr>
<td>$\sigma = 95%, Dist = 0, c_{ij} &gt; 0.95$</td>
<td>53.1%</td>
<td>9.2%</td>
<td>37.7%</td>
<td>97.27%</td>
<td>2.73%</td>
<td>100.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

As the audit logs are continuously collected over the year, we setup 6 analyses at a rate of every two months. Each analysis takes as input the accumulated audit logs from Jan 2014 to the current month (e.g., “Feb” is 2-month logs, “Apr” is 4-month logs, “Dec” is the entire year’s logs). It is a typical scenario of semi-supervised learning with incremental input data. It also follows the nature that new benign/malicious patterns are gradually accumulated in audit logs over time.

As shown in Figure 4.3, EASEAndroid dramatically outperforms the naive matching in each analysis. As the total number of access pattern keeps increasing, EASEAndroid’s coverage reaches about 74% in the last Dec. EASEAndroid also discovers that the majority of denied access patterns in real world are malicious and they keep emerging while benign access patterns gradually get stable. In contrast, the coverage of naive matching remains around 7%, because it can only match access patterns related to the initial known ones, such as the 9 exploit kits, which are updated with a small set of new access patterns over time. But it still leaves the majority unclassified.

Specifically, all three learners of EASEAndroid contribute to the high classification coverage. In the first Feb analysis, EASEAndroid first matches 118 malicious access patterns, same as naive matching. Then it performs multiple learning iterations with the current audit logs in both automated and semi-automated mode. In summary, in automated mode, the NN classifier finds 282 patterns using threshold \((m, \sigma) = (10, 85\%)\) in IsMajorityKnown. The pattern-to-rule distance measurer finds 95 patterns using \(Dist(ap_i) = 0\) with existing neverallow rules. The co-occurrence learner finds 110 patterns with \(c_{ij} > 0.85\). In semi-automated mode, we relax the thresholds to \((10, 75\%), Dist(ap_i) \leq 1, c_{ij} > 0.75\), respectively and further find 143 patterns based on the majority vote of the three learners.

As for benign access patterns in Feb, since the initial knowledge lacks benign patterns, the pattern-to-rule distance measurer classifies the first 23 benign patterns using \(Dist(ap_i) = 0\) with existing allow rules and add them to the knowledge base. Then the three learners contribute the
4.5. EVALUATION

Thanks to the strict thresholds, the automated mode classifies access patterns with no false result. But in the semi-automated mode, we do find 34 false-benign (False-Negative\(^5\)) access patterns and the 72 false-malicious (False-Positive) ones in the final Dec, mainly due to two reasons. First, a small set of access patterns are shared by both privileged benign system binaries and malicious apps with similar occurrences (e.g., both access `/proc/stat`), which make EASEAndroid hard to distinguish with relaxed thresholds. Second, some mis-classified patterns in early analyses affect the learning precision in later ones. In fact, there are only 4 false-malicious patterns in Feb. But then the NN classifier uses them to mistakenly find more false-malicious ones in the following analyses. Nevertheless, this limitation of the semi-automated mode is expected. Therefore in practice, it requires policy analysts to verify the result to avoid error propagation. Analysts can also input extra constraints and knowledge about privileged system binaries to help EASEAndroid increase the precision.

There are still 906 access pattern unclassified in the final analysis that EASEAndroid are uncertain due to low occurrence (less than five days throughout the year). After manual analysis, we find that some access patterns are likely malicious and might be some isolated attack attempts in the wild. But the statistics is too low to reach the threshold. In such cases, we have to wait for more similar access patterns coming in future audit logs.

### 4.5.3.2 Coverage & precision with different threshold settings

The thresholds for the three learners play an important role on the coverage and the precision of EASEAndroid. In practice, it is important to find a balance between the coverage and the precision. In this section, we further investigate the detailed coverage and precision difference by choosing 5 different threshold settings from very relaxed to very strict as shown in Table 1. The listed threshold settings are for the automated mode. The semi-automated mode are relaxed by reducing 10% on \(I_{Majority\ Known}\) (minimum neighbors unchanged) and \(c_{ij}\), and increasing 1 in \(Dist(a_{pi})\). The first three columns show the overall percentages (adds to 100%) summarizing the classified malicious and benign and unclassified over the total 3530 access patterns. The following four columns provide the more detailed TP/FP and TN/FN percentage of classified benign and malicious patterns.

We can see that the thresholds in Row 1 is largely relaxed. Although it has the highest coverage, both FP (37.04\%) and FN (41.35\%) are too high, making it practically useless. In contrast, Row 5’s thresholds achieve 100% correctness on classifying benign patterns. But it also leaves 37.7\% patterns

\(^5\)We treat malicious as positive, benign as negative
4.5. EVALUATION

unclassified. The middle 3 rows are more balanced. Row 3 and 4 are candidates for practical use. In practice, analysts can also use multiple thresholds respectively, such as with Row 4 and 5 together, and only need to investigate the diff of their learning results since we have high confidence with the result of Row 5.

Admittedly, each individual threshold for each learner may have different effect on the final classification result. To analyze more detailed threshold difference, or find an optimal vector of thresholds, a cross-validation can be performed with multiple real-world audit log sets[Koh95].

4.5.4 Case Study of Refinement Generation & Comparison with Human Policy

In the last Dec analysis in Figure 4.3, the policy refinement generator finally generates 51 new allow rules from the 336 benign access patterns, and 280 new neverallow rules from the 2182 malicious access patterns, by extending identified incomplete rules and creating new fine-grained security labels to replace existing coarse-grained ones. In this section, we use the following example as a case study to illustrate the generated refinement.

EASEAndroid classified as benign 9 access patterns that read some time-zone data files under /data/misc/zoneinfo. These access patterns are found in multiple Android framework-related binaries (subjects) in /system/bin, including surfaceflinger, dhcpcd, pppd and a vendor-specific daemon. In the 6-tuples, the time-zone data files carry system_data_file, which is the default label for all files under /data. Naively generating a rule with this label (using audit2allow) would over-grant the subjects with permissions to access all files under /data.

EASEAndroid instead finds that these files all share the same /data/misc/zoneinfo file path prefix, and thus derives a new label zoneinfo_file specifically for them. It also creates a new attribute access_zoneinfo_domain to group the above subject domains. Finally, EASEAndroid generates a new rule:

```
allow access_zoneinfo_domain zoneinfo_file:file {open read}
```

This rule only covers the 9 patterns observed by EASEAndroid, thus preventing unnecessary accesses being granted, following the least privilege principle.

The rules generated by EASEAndroid for the 336 benign access patterns are compared with human-written rules in the later policy version. All access patterns allowed by EASEAndroid rules are also permitted by human-written rules. EASEAndroid in general creates a larger set of more-specific rules (human-written rules frequently use policy macros [Sele]). It may be desirable to aggregate these more specific rules for better human-readability; this remains for future work.
4.5.5 Case Study of Classified Malicious Access Patterns

For the one-year dataset of audit logs processed by EASEAndroid, 2182 access patterns are classified as malicious. The reader is reminded that the starting point of analysis is 17 access patterns derived (manually) from 9 confirmed exploit kits. The access patterns newly classified as malicious by EASEAndroid capture malicious behavior much more precisely than has previously been possible. To the best of our knowledge, this is the first large-scale study of system-level malicious access patterns from real-world Android devices.

The subjects in these malicious access patterns are mostly untrusted third-party shell binaries and apps. For the purpose of understanding and discussion, they are categorized based on the permissions [Selc] (shown in braces) and the objects they accessed, which were mainly privileged files. Figure 4.4 shows the resulting 8 categories of malicious access patterns, each discussed below. Two of them (modify /sys/fs/selinux and transition to privileged domains) are new attacks in Android which directly target the SEAndroid MAC mechanism itself.

1. Exploit /dev nodes
The most common malicious access patterns are the ones that exploit various vulnerabilities in device nodes under /dev. For instance, EASEAndroid found 62 different shell binaries and exploit apps trying to directly read and write /dev/graphics/* (exploiting a previously-known framebuffer vulnerability). After identifying these subjects, EASEAndroid further discovered that they bundled various exploits targeting several other device nodes as well (including vendor-specific nodes). Some of these subjects were found to successfully gain root privileges. However, note that a good SEAndroid policy is still able to provide protection even on a rooted device (e.g., even init has limited permissions), as long as the Linux kernel is not compromised.6

2. Request file-related privileged capabilities
The second most frequent category of malicious access patterns are that subjects try to use privileged capabilities to modify the file mode bits and ownership of various files. This is a classic privilege escalation attack step; external binary files pushed to the device (e.g., in /data/local/tmp) may be given unintended capabilities, and important data files (e.g., in /data/system) can be made writable for attacks to proceed.

3. Modify /system partition
This is also a common step in exploits that the /system partition is modified with new binaries added such as su,busybox. Normally, the /system partition is mounted as read-only. But some subjects were able to remount the partition as writable. However, they were still captured by the

---

6Since certain subjects gain root, they may be able to rollback to the permissive mode. The audit logs might just record the malicious access patterns but not actually block them.
audit logs because their domain labels were not allowed to write system_file under /system.

4. Access /sys filesystem

/sys is a virtual filesystem that exports kernel-level information to userspace, normally used by privileged system daemons. EASEAndroid found that untrusted subjects also try to directly access /sys, particularly /sys/fs/selinux, which contains the policy content and runtime state. Untrusted subjects may try to modify the policy content, either to switch to the permissive mode, or to get more permissions. We believe this is a new type of attack directly against SEAndroid MAC mechanism. Although this new attack is expected to emerge, it is still surprising to discover that the new attack has already become popular in the wild.

5. Request process-related privileged capabilities

EASEAndroid also found that some untrusted subjects ask for privileged process capabilities, such as killing other processes, or sys_admin managing a list of functionalities [Selc]. The most common example is to use sys_ptrace to ptrace another process. This capability is attempted by several third-party management/monitor apps, and game hacking apps (to modify other game apps’ score/rewards).

6. Transition to privileged domains

Another new type of attack found by EASEAndroid is that untrusted apps try (some succeed) to transform their subject domains from untrusted_app to domains with higher privileges, including init, init_shell, system_app, and vendor daemon domains. Interestingly, this case is found due to the conflicts reported by the majority-vote in the semi-automated mode. An access pattern classified as malicious by both the NN classifier and the co-occurrence learner, is classified as benign by the pattern-to-rule distance measurer, because it is close to an allow rule, indicating the subject carries a wrong domain label.

7. Access /proc filesystem

Like /sys, /proc is also frequently accessed, especially by third-party management/monitor apps. Although reading /proc/app_pid/* might not be directly damaging, the information can be leveraged as a side-channel to compose attacks [Che14]. Besides, EASEAndroid also showed that certain apps try to write /proc/sys/kernel/kptr_restrict to gain access to the kernel symbol table, a common step in kernel exploits.

8. Connect to Unix sockets of privileged daemons

Unix domain socket is a more complicated case in SEAndroid. By design, some Unix sockets in system daemons such as adbd, debuggerd can be connected by apps, while others are reserved only for privileged daemons. EASEAndroid is able to distinguish these two cases, mainly by the

---

7Policy analysts suggest that it is also possible that some vendor daemons have zero-day vulnerabilities that are exploited to run attacks.
co-occurrence learner. It found one new benign pattern between two vendor daemons and several malicious ones that untrusted apps try to directly connect to Unix sockets of highly privileged daemons, such as init.

In summary, with EASEAndroid’s learning, we find a group of interesting malicious access patterns and new attacks in Android. EASEAndroid also generates 280 fine-grained neverallow rules. 52 rules are found in the later policy. But others still require deeper investigation, since the knowledge learned by EASEAndroid is not sufficient to understand the attack mechanisms behind these malicious access patterns.

4.6 Discussion

Blurred line between benign and malicious
In practice, the line between benign and malicious might be blurred and subjective. It depends on specific security requirements and use cases to determine whether an access pattern is really benign or malicious. For example, individual users may like rooting their own devices and using the game hacking apps mentioned above, while game developers treat them as malicious because they bypass the in-app purchase. Nevertheless, EASEAndroid’s learning provides more detailed evidence of various patterns’ semantics for policy analysts to make the final decision.

Information missed by audit logs EASEAndroid relies on audit logs to learn new patterns and derive policy refinements. However, two types of information could be missed or not available in audit logs, which can cause EASEAndroid to miss important knowledge. First, by default, audit logs only capture system-level operations that are denied by the policy currently loaded in a device. If the policy is too permissive or has too coarse-grained allow rules, malicious access patterns could be mistakenly allowed and missed by audit logs. To mitigate this issue, analysts should use auditallow to mark coarse-grained/uncertain rules so that audit logs can still capture the operations allowed by these rules.

Second, framework-level operations are not available in audit logs, because they are controlled by Android permission model. But these upper-level operations contain valuable semantics (e.g., attack mechanisms). Without them, it is difficult to explain and distinguish certain benign/malicious access patterns in audit logs. Since Android 5.0, logcat is involved in SEAndroid auditing. In future, EASEAndroid can integrate logs from logcat to have more semantics in the knowledge base.

Countermeasure against EASEAndroid Similar to tampering virus sampling in AntiVirus programs, attackers can disable or compromise the audit log mechanism (logging and uploading) to avoid malicious access patterns being learned. Currently, we rely on Linux kernel protection[Aza14] to ensure the integrity of audit log mechanism. And we argue that enabling audit log is a recommended
security service for the majority users who want policy refinements (or mandatory for enterprise users). If a malicious pattern is widely spread and affects a large number of normal users, audit logs can catch it for EASEAndroid to analyze.

It is also potentially possible that attackers manipulate the co-occurrence rate by intentionally forcing the benign and malicious patterns to co-occur in one log, such as triggering the benign pattern first and then launching the attack. Such data poisoning attack may fool EASEAndroid's learning, which requires extra constraints or more logs from different devices to dilute the poisoned logs[Big11].

4.7 Related Work

Though SEAndroid is fairly new, SELinux has been developed and researched for years, including SELinux policy analysis and verification[Zan04; Hic10; Ala08; Sas06], policy visualization[Xu08], policy conflict resolving[Jae04], policy simplifying[Nak05; Nak09], policy comparison[Che09], policy information-flow integrity measurement[Gan06; Jae03; Jae06; Sha06], etc. Also, the above research work usually assumes a relatively complete SELinux policy that has already been well developed. And the analysis usually focuses on stable desktop Linux system or only a few specific application programs (e.g., sshd, httpd). Due to the architecture difference, SEAndroid faces different challenges from SELinux, because current SEAndroid policy is far from complete. It is still under active development and continuous refinement (Section 4.1).

In terms of SELinux policy generation, Polgen proposed by MITRE is a tool that guides policy analysts to develop policies based on system call traces[Sni06]. However, it does not have machine learning capability and only focuses on system call traces from a single application program, which is not scalable. Madison proposed by Redhat is an extension of audit2allow that can generate policy similar to the reference policy style[Mac07]. However, like audit2allow, it cannot create new security labels to cover new access patterns.

There is very little SELinux research related to machine learning. Marouf et al. proposed a similar approach to Polgen that analyzes system call traces to simplify SELinux policy[Mar10]. Markowsky et al. proposed an IDS system that uses SELinux denials as input to an SVM classifier to detect attacks [Mar12]. But there was no policy analysis or refinement.

4.8 Summary

Developing SEAndroid policies is a non-trivial task. In this chapter, we have proposed EASEAndroid, the first SEAndroid audit log analytic platform for automatic policy analysis and refinement. EASE-
Android innovatively applies semi-supervised learning to MAC policy development. It has been evaluated with 1.3 million audit logs from real-world devices. It successfully discovered over 2500 new benign and malicious access patterns, generated 331 policy rules, and found 2 new attacks in the wild directly targeting SEAndroid MAC mechanism.
4.8 SUMMARY

CHAPTER 4. EASEANDROID

Figure 4.3 The comparison between naive matching and EASEAndroid on analyzing the same set of access patterns.
4.8. SUMMARY

CHAPTER 4. EASEANDROID

Distribution of Classified Malicious Access Patterns

1. {read,write} files on /dev/graphics, /dev/block, /dev/exynos-mem, /dev/mem, /dev/android_adb, /dev/s3c-mfc
2. {dac_override,chown,fsetid} capability on /data/data, /data/local, /data/misc, /data/system, /sdcard
3. {create,write,unlink} files on /system/app, /system/bin, /system/xbin, /system/etc
4. {read,write} files on /sys/block, /sys/devices, /sys/fs, /sys/kernel
5. {kill,sys_admin,sys_trace,sys_chroot,setuid,setgid} capability
6. {transition,dyntransition} process
7. {read,write} files on /proc/sys, /proc/app_pid/cwd|environ|exe|mem|moun
ts
8. {connectto} unix domain sockets of privileged daemons directly

Figure 4.4 The distribution of malicious access patterns classified by EASEAndroid.
SEAndroid is a mandatory access control (MAC) framework that can confine faulty applications on Android. Nevertheless, the effectiveness of SEAndroid enforcement depends on the employed policy. The growing complexity of Android makes it difficult for policy engineers to have complete domain knowledge on every system functionality. As a result, policy engineers sometimes craft over-permissive and ineffective policy rules, which unfortunately increased the attack surface of the Android system and have allowed multiple real-world privilege escalation attacks.

We propose SPOKE, an SEAndroid Policy Knowledge Engine, that automatically extracts domain knowledge from rich-semantic functional tests and further uses the knowledge for characterizing the attack surface of SEAndroid policy rules. Our attack surface analysis is achieved by two steps: 1) It reveals policy rules that cannot be justified by the collected domain knowledge. 2) It identifies potentially over-permissive access patterns allowed by those unjustified rules as the attack surface. We evaluate SPOKE using 665 functional tests targeting a set of security functionalities developed by a major Android vendor. SPOKE successfully collected 12,491 access patterns for 28 functionalities as domain knowledge, and used the knowledge to reveal 320 unjustified policy rules and 210 over-permissive access patterns defined by those rules, including one related to the notorious libstagefright vulnerability. These findings have been confirmed by policy engineers.
5.1 Introduction

Security-Enhanced Android (SEAndroid), also known as SELinux in Android, is a framework to enforce a Mandatory Access Control (MAC) policy on native access operations in an Android system [Sea]. SEAndroid is capable of limiting the impact of attacks by confining malicious and compromised applications. However, the protection is only as good as the SEAndroid policy. Ideally, SEAndroid strives to define a least-privilege [SS75] policy for subjects. However, in reality, policy engineers define the policy in a more conservative manner, meaning the policy is defined to allow access patterns that could be unnecessary to the functionality required by the Android system. Unfortunately, allowing unnecessary access patterns increases the attack surface of the SEAndroid policy and the target Android system.

This conservative policy development is the result of multiple factors: 1) policy engineers have incomplete domain knowledge of knowing what exact access patterns are required by the system; and 2) the consequences of breaking functionality or adversely impacting user experience are significant. Designing a good SEAndroid policy requires domain knowledge of the entire functionality of the Android system, which typically incorporates components that continuously grow in their complexity. As a result, it becomes increasingly difficult for particular individuals, or teams, to possess the domain knowledge required for developing an effective policy. Policy engineers, who are responsible for writing and maintaining the policy, cannot keep the domain knowledge on every system functionality. At the same time, functionality developers usually lack SEAndroid expertise to write policy rules on their own. In other words, there is a knowledge gap between policy engineers and functionality developers.

The impact of this knowledge gap on the policy attack surface was recently demonstrated by two real-world attacks on Android. Both resulted from policy rules that were defined to allow unnecessary access patterns. In the first example, a pre-installed keyboard app in a popular Android device was mistakenly over-granted unnecessary access permission, which caused privilege escalation (CVE-2015-4640, CVE-2015-4641). In the second example, a vulnerability in a system daemon was successfully exploited via an access pattern that was mistakenly allowed by an outdated unnecessary policy rule (CVE-2015-3825). These two examples show that policy engineers lack proper knowledge about the required access patterns of system functionality. This result is also confirmed by a recent study [Res15] that showed multiple Android devices have vendor-modified policies that are less strict and have wider attack surface than Google's baseline policy.

In this chapter, we propose SPOKE, an SEAndroid Policy Knowledge Engine that identifies potentially unnecessary attack surface in SEAndroid policy by bridging the knowledge gap between policy engineers and functionality developers. To achieve this goal, SPOKE provides three capabilities.
First, SPOKE automatically extracts a knowledge base of functionally required access patterns from semantically rich functional tests. Second, SPOKE uses the knowledge base to identify potentially unnecessary access patterns allowed by corresponding SEAndroid policy rules. Finally, SPOKE analyzes the attack surface of the SEAndroid policy by constructing a bipartite graph depicting the access patterns between subjects and objects. It further aids policy engineers to triage potentially unnecessary access patterns and corresponding policy rules by highlighting areas of high risk.

We implemented a prototype of SPOKE for a major Android vendor, and evaluated it by taking inputs of 665 functional tests targeting vendor’s enterprise Mobile Device Management (MDM) framework. The MDM framework is a representative case for SPOKE as it provides enterprise security functionalities that require strict access control protection by SEAndroid policy. SPOKE successfully collected 12,491 low-level access patterns correlated with 1,492 high-level functionality traces of core MDM classes (android.app.enterprise.*) as the domain knowledge. With this knowledge base, SPOKE first identified functional needs (or partial functional needs) for 1,036 rules out of total 1,356 MDM-related SEAndroid policy rules. In the remaining 320 potentially unnecessary policy rules, SPOKE further identified 210 over-permissive access patterns, including an access pattern related to the libstagefright vulnerability [Lib]. Policy engineers have confirmed SPOKE’s findings and revised the policy.

In summary, this work makes three contributions:

1. We propose SPOKE, a novel knowledge collection and analytics engine that bridges the knowledge gap between policy engineers and functionality developers.

2. We implement SPOKE by first building a knowledge extraction platform that automatically collects domain knowledge from rich semantic functional tests, and second, creating an analytic engine to identify potentially unnecessary policy rules, which can aid attack surface analysis of a policy.

3. We evaluate SPOKE using 665 functional tests targeting security functionalities developed by a major Android vendor. SPOKE successfully collects 12,491 access patterns and 1,492 functionality trace as the domain knowledge. SPOKE further uses this knowledge to identify 210 over-permissive access patterns. SPOKE’s findings help policy engineers identify and fix the risky policy rules.

We note that SPOKE’s performance is directly related to the quality of functional tests used to define the knowledge base. However, a perfect set of functional tests is not required to benefit from SPOKE. Functional tests can be imperfect in two ways: 1) they do not include all functionally required access patterns; and 2) they include unnecessary access patterns. In practice, we believe
that in the case of functional tests, missing required access patterns is much more common than including unnecessary access patterns (and we have techniques for removing common unnecessary access patterns). For functional tests with a large number of missing required access patterns, SPOKE will still function, but produce many more potentially unnecessary rules. In practice, we found that triaging the rules using a bipartite graph and highlighting high risk areas produces effective results. Section 5.3 states our assumption of functional tests in more detail.

5.2 Background and Definitions

5.2.1 SEAndroid Basics

As mentioned in Section 4.2.1, SEAndroid is a port of SELinux [Sma01] to Android with extensions to support Android-specific features, such as Binder IPC [SC13]. The goal of SEAndroid is to reduce attack surface and contain damage if any flaw or vulnerability is exploited for privilege escalation, via MAC enforcement in Android native layer. For more details of the basic background of SEAndroid, please refer to Section 4.2.1. Here, we clarify the concepts of SEAndroid that are closely related to this work, followed with formal definitions.

An SEAndroid policy has two parts. The first part is a mapping that assigns security labels to concrete subjects (or objects) sharing the same semantics. Traditionally, a subject label is called a domain. An object label is called a type. The second part is a set of rules that define which domain of subjects can access which class and type of objects with a set of permissions [LS01]. For example,

```
allow app app_data_file:file {read write}
```

allows processes with app domain to read and write file objects with app_data_file type. Since SEAndroid policy is a whitelist-based policy, allow rules are the major rules used for runtime enforcement. In the rest of this work, we refer to allow rules as major policy rules. Apart from allow rules, to avoid malicious accesses being mistakenly allowed, neverallow rules encode malicious accesses to check allow rules at compile time. During runtime, if no allow rule can match an access event, the access event will be denied and logged [Sela] (Section 5.4.1.2 introduces a new way of logging access events).

An access event usually has two entries\(^1\) as shown in Listing 5.1. The first entry (type=1400) records the access operation between specific subject (by comm) and file object (by path), with their security labels untrusted_app, app_data_file and permission write. The second entry (type=1300) captures more details of the related system call and the subject information such as uid,gid.

\(^1\)Previously, there was an object entry, which is merged into 1400.
5.2. BACKGROUND AND DEFINITIONS

Listing 5.1 A simplified access event example recorded at the epoch time 1445635785.573 in an audit log. It consists of two entries: subject & object with labels and permission (1400), syscall info (1300).

Traditionally, policy engineers develop and refine policy rules by manually analyzing the access events. Only a few basic tools (e.g., setools[Set]) previously in SELinux can be used for SEAndroid. Such tools can only perform manual and syntactic analysis with no capability of semantic analysis. Particularly, a tool called audit2allow can directly transform security labels in type=1400 entry of an access event into an allow rule. However, it could cause default/coarse-grained security labels to be used inappropriately, which is one security issue mentioned in [Res15].

5.2.2 SEAndroid Definitions

To clarify the concepts of SEAndroid, we present the following definitions. We first introduce the definition of access pattern, which is one of the key concepts used in this work.

Definition 5.1 (Access Pattern). An access pattern is a 4-tuple $a = (s, o, c, p)$. It denotes a concrete subject $s$ accesses a concrete object $o$ of class $c$ with permission $p$.

An access pattern can be either extracted from corresponding items in a raw access event, or defined in the set of allowed accesses by a policy rule. In the first case, $s$ is a fine-grained concrete value extracted from scontext, comm(command), exe(executable) and pid in an access event. Similarly, $o$ is extracted from tcontext, name and path. $c$ is from tclass and $p$ is the permission. For example, the access pattern of Listing 5.1 is (‘contacts_app’, ‘/data/data/contacts_app/contacts.db’, ‘file’, ‘write’). The second case is explained as following.

Definition 5.2 (SEAndroid Policy). An SEAndroid policy is $P = (L_s, L_o, M, S, O, R)$, where $L_s, L_o$ are the set of security labels of subjects and objects, $M : L_s \cup L_o \rightarrow S \cup O$ is a mapping that assigns security labels to concrete subjects $S$ and objects $O$, $R = \{r\}$ is the set of policy allow rules.
In practice, we parse the compiled SEAndroid policy and store each element in \( \mathbb{P} \) as a database table. Concrete subjects and objects are collected from devices that are either in a clean state (e.g., after factory-reset) or running functional tests (test-only temporary subjects/objects are excluded).

Given a policy rule \( r \in R \), "allow \( l_s l_o : c_r P_r \)”, we further define it as the following.

**Definition 5.3 (Policy Rule).** An SEAndroid policy rule is a tuple \( r = (l_s, l_o, S_r, O_r, c_r, P_r, A_r) \), where subject label \( l_s \in L_s \), object label \( l_o \in L_o \). \( S_r = M(l_s) \) and \( O_r = M(l_o) \) are the sets of concrete subjects and objects mapped with the labels, respectively. \( c_r \) is the class of the objects, \( P_r \) is the permission set granted to the subjects when accessing the objects, \( A_r = \{ a = (s, o, c_r, p) \mid s \in S_r, o \in O_r, p \in P_r \} \) is the set of all access patterns defined by this rule, i.e., \( A_r = S_r \times O_r \times \{ c_r \} \times P_r \).

Here, we extend the policy rule \( r \) with the set of concrete access patterns \( A_r \) that this rule defines to allow. Note that, the access patterns collected from runtime access events (e.g., required by the functionality under test) could be inconsistent with the access patterns defined by the policy rules, due to the knowledge gap, which SPOKE is designed to address.

### 5.2.3 Android Functional Testing

A functional test examines whether a specific functional component meets the design requirement, by feeding an input and checking the expected output. In Android testing, functional tests are developed using Android testing framework, which is an integral part in the official Android development environment. It provides libraries such as AndroidJUnitRunner, UI Automator [Andb].

Since Android functional tests are developed based on the standard JUnit structure and automated UI simulation, they are well organized and closely associated with design requirements and end user operations. This makes functional tests self-explanatory and inherently carry rich semantics of the functionality under test. Examples of such functional tests are checking specific API functions (Unit test), clicking/typing on UI widgets (UI test), and setting up an enterprise email account (Integration test).

We hypothesize that Android functional tests can enable a systematic way to synchronize domain knowledge between developers and policy engineers, and provide the knowledge foundation for the attack surface analysis of SEAndroid policy. Policy engineers can run developers’ functional tests to extract functionality trace and access patterns, store them into a knowledge base, and use the knowledge base to justify and analyze attack surface of policy rules. Section 5.2.4 defines the new concepts introduced in this work.

**Impact of Functional Test Coverage**

An important factor of using functional tests is the test coverage, which is an active industrial and research field in software engineering. Note that, test coverage is orthogonal to our work. One
of our contributions is to leverage the outcomes of industrial practice and research efforts in the field of test coverage, to enhance the security analysis of SEAndroid policy.

Specifically, in the software industry, multiple coverage-measuring tools [Emm; Jac] are developed to ensure the high test coverage. As test-driven development (TDD) [Bec03] is a popular software engineering practice, many Android testing tools and frameworks are actively used in the industry (e.g., Testdroid [Tes], AWS device farm [Aws]).

Increasing test coverage is also a research topic in software testing [DO91; BY98; McM04; Vis04]. Various techniques have been developed for automated testing and test input generation of mobile applications. For example, Dynodroid [Mac13] is an automated test input generation system for Android apps. Swifthand [Cho13] is a guided GUI testing system for Android apps based on machine learning. Symbolic and concolic executions are also used to generate event sequence for automated testing of Android apps [Jen13].

In Section 5.1, we mentioned two possible imperfections of functional tests in practice. We note that even using functional tests with low coverage, SPOKE can still function. As test coverage increases, the value of SPOKE linearly increases by collecting more domain knowledge and analyzing more policy rules. In other words, we design SPOKE as a dynamic system, which can be continuously running. Every time a new functional test is developed, SPOKE can extract new domain knowledge, expand the knowledge base, and analyze the attack surface of the policy incrementally and repeatedly.

5.2.4 Definitions introduced by SPOKE

Here, we use the following definitions to clarify the concepts introduced in this work.

**Definition 5.4 (Functionality Trace).** A functionality trace is a set of descriptive items that can describe the execution semantics of the functionality. In functional tests, the following concrete and semantic items can be automatically collected as descriptive items:

- Metadata of a functional test, such as `test_class`, `test_case`, `@annotation`
- Key function calls/control flow within the execution of a functionality, such as API calls

Intuitively, a descriptive item shows one aspect of a functionality. By monitoring the runtime execution of the functional test, we obtain concrete and specific items that describe how the functionality works from multiple aspects and granularities. An example is `test_addFirewallRule, {Firewall.addRule, Firewall.setIptablesOption}`, where the former item shows a user-level functional operation under test, and the latter two items provide code-level details of
5.2. BACKGROUND AND DEFINITIONS

the functionality. In SPOKE, such descriptive items are correlated with low-level access patterns, providing a full picture of a functionality.

A knowledge base stores correlated functionality trace and access patterns in a unified form. Formally, we define this as the following.

**Definition 5.5 (Knowledge Base).** A knowledge base is a set of pairs \( K = \{(a, f)\} \), where ‘\( a \)’ denotes an access pattern from kernel layer (Section 5.4.1.2) and ‘\( f \)’ denotes a functionality trace from Dalvik layer (Section 5.4.1.3). A pair of \((a, f)\) shows there is a correlation between \( a \) and \( f \).

Given a pair of \((a_k, f_k)\), we use the predicate \( \text{Correlated}(a_k, f_k) \) to denote whether they are correlated. If \( \exists (a_k, f_k) \in K \), then \( \text{Correlated}(a_k, f_k) = \text{True} \). In practice, \( K = \{(a, f)\} \) is stored in a database table. Given a functionality trace \( f_k \), we can query the database to find all correlated access patterns \( A_k = \{a_k | \text{Correlated}(a_k, f_k) = \text{True}\} \).

A potentially unnecessary access pattern is one that is not found in the knowledge base and thus cannot be justified by the knowledge base. If none of the access patterns in the \( A_r \) of a rule \( r \) can be justified by the knowledge base, then the entire rule is considered to be a potentially unnecessary rule. To be more precise and formal, we define “potentially unnecessary” as “unjustified w.r.t. the knowledge base”, which is shown in the following.

**Definition 5.6 (Policy Rule Justification w.r.t. \( K \)).** A policy rule \( r \) is said to be justified with respect to a knowledge base \( K \) if for all access patterns \( a_r \) in \( A_r \), there is a corresponding access pattern \( a_k \) in \( K \) (i.e., \( a_r = a_k \)). A rule is said to be partially justified (or unjustified) if only a subset (or none) of the access patterns in \( A_r \) have corresponding entries in the knowledge base.

For example, by running a functional test \texttt{test_addFirewallRule()}, policy engineers can understand and correlate functionality trace:

```
android.app.enterprise.Firewall.addRule()
```

with access pattern:

```
('system_server', '/system/bin/iptables',
 'file', 'execute')
```

and thus justifies a policy rule \texttt{allow system_server iptables_exec:file \{execute\}} that allows the access pattern.

Among the access patterns that cannot be justified w.r.t. \( K \), we further focus on the over-permissive access pattern which is defined as following.

**Definition 5.7 (Over-permissive Access Pattern).** An over-permissive access pattern is an access pattern \((s, o, c, p)\) defined by a policy rule that can potentially allow attackers to use or exploit
the subject ‘s’ to maliciously access the object ‘o’ with permission ‘p’, in order to compromise the confidentiality or integrity of ‘o’.

5.3 Problem Statement and Assumptions

Problem Statement: In this paper, we seek to reduce the attack surface of an SEAndroid policy by identifying unnecessary access patterns allowed by the policy. Because there is a knowledge gap between policy engineers and functionality developers, causing the necessary access patterns to be unclear, we use functional tests as a proxy to extract the necessary access patterns for policy analysis.

More formally, let $A_N$ be the set of access patterns determined to be necessary from the functional tests. Let $A_P$ be the access patterns allowed by a policy $P$. In this work, we assume that $A_N \subseteq A_P$, otherwise the functional tests would not pass. Our goal then becomes to identify the unnecessary access patterns $A_U = A_P \setminus A_N$. However, since functional tests may not be perfect in practice, we frequently refer to these access patterns as potentially unnecessary access patterns.

Assumptions: We assume that functionality developers use functional tests to verify the design and execution logic of the functionality, which is consistent with the industrial practice. We therefore assume that functionally required access patterns can be extracted by running functional tests. Note that, as mentioned in Section 5.2.3, test coverage is orthogonal to SPOKE.

We also assume that tests are executed on real devices because some functionalities require hardware features such as ARM TrustZone. Before running each test, devices are in the same clean state as being ready for normal user operation. We also assume that target functionalities are correctly implemented and already passed the tests successfully. Functionalities should involve multi-layer operations in Android, which are visible by low-level SEAndroid access control. This is the typical case for security functionalities, such as MDM functionalities in our evaluation (Section 5.5.1). No malicious operations exist since tests are executed in a clean state. Hence, all access patterns related to functionalities (exclude test-only operations) should be allowed.

5.4 SPOKE

SPOKE is a novel test-driven SEAndroid Policy Knowledge Engine that achieves the three capabilities:

C1: Automatically building an up-to-date knowledge base of various functionalities and their corresponding access patterns to bridge the knowledge gap between developers and policy

---

In practice, over-permissive rules lead to over-privileged subjects
5.4. **SPOKE**

**CHAPTER 5. SPOKE**

Figure 5.1 SPOKE consists of three components from collecting test-driven tracing logs to generating final report by the analytic engine.

**engineers.**

**C2:** Matching policy rules with corresponding functionality traces and access patterns. The output of this process is used as the basis to justify proper policy rules and reveal unjustified policy rules, with respect to the knowledge base.

**C3:** Analyzing the attack surface of unjustified policy rules to pinpoint risky rules that allow unnecessary and potentially over-permissive access patterns.

The design of SPOKE is based on a key insight that Android functional tests are rich semantic resources provided by functionality developers that can be used to enable automated domain knowledge collection. The collected domain knowledge can then help policy engineers systematically analyze SEAndroid policy rules, including revealing whether policy rules can be justified or not by corresponding functionality, and identifying over-permissive policy rules that are potentially exploitable by attackers.

**Automatic knowledge collection** is achieved by a two-step process. Firstly, given a functional test, SPOKE executes it in a scalable test running and multi-layer knowledge extraction platform. The platform collects both high-level functionality trace in Android framework layer and low-level access patterns in Android native layer. By doing so, SPOKE captures a full and detailed picture of how the functionality works, which represents the domain knowledge of the functionality.

Secondly, SPOKE parses the collected knowledge from different layers in different forms into a knowledge base and performs a cross-layer correlation to organize them in a unified form. The correlation is based on multiple global variables (e.g., timestamp, process/user id) shared across layers that align and match high-level functionality trace and low-level access patterns. In addition, SPOKE also parses SEAndroid policy rules into a structural form, and associates the rules with the
access patterns that these rules are defined to allow.

**Policy rule justification w.r.t. K** is the first analytic capability that uses the collected knowledge. It performs a matching between the access patterns defined in the policy rules and the access patterns correlated with functionality trace in the knowledge base. The output of this process checks whether the policy rules, specifically their defined access patterns, can be matched and therefore justified by the corresponding functionality. On the other hand, it also reveals the risky policy rules whose defined access patterns cannot be justified by current knowledge base.

**Attack surface analysis** is the second analytic capability that particularly focuses on the risky policy rules and their defined but unjustified access patterns revealed from the above matching output. Such unjustified access patterns not only lack critical tests, but could also be unnecessary and over-permissive, because they might allow potentially vulnerable subjects to access valuable objects. To identify such over-permissive access patterns, SPOKE again turns to the knowledge base to find valuable or critical objects, which are typically correlated with important functionality trace. Then SPOKE searches the unjustified access patterns and highlights the ones that can access these critical objects. This helps policy engineers pinpoint risky rules and corresponding over-permissive access patterns with concrete evidence.

Figure 5.1 shows the major components in SPOKE that implement the above three capabilities. The device farm with multi-layer logging realizes the first step of domain knowledge collection. The knowledge base with cross-layer correlation is the second step. The analytic engine provides both policy rule justification w.r.t. K and attack surface analysis. In practice, it also provides visualization to illustrate the analytic results. The following sections explain more details of each component.

### 5.4.1 Automatic Knowledge Collection

#### 5.4.1.1 Distributed Multi-layer Collection

To build a knowledge base, collecting runtime logs of functional tests is the first step for knowledge extraction. As we focus on functionalities involving multi-layer operations, we need to capture logs from each layer to get a full picture of the functionality. To this end, SPOKE collects logs from three layers: Linux kernel layer, Dalvik VM layer and Android native layer, as explained in the following sections.

Furthermore, since SEAndroid checks access events based on system calls, given the high calling rate and large volume of system calls, it is necessary to support both high-rate and large-volume logging to capture all access events, with the capability of keeping track of specific logging targets to identify different process subjects and file objects. Unfortunately, the logging buffer in one device has upper-bound limitations of both speed and volume, which may miss critical access events.
To address this challenge, we design a distributed logging mechanism. Inspired by distributed computing, we group a set of identical Android devices (same model with same setting) as a device farm. A centralized manager configures each device to focus on specific logging targets. The overall work load of logging a functional test is then divided and distributed to each device with a specific logging work load, so that different logging targets are collected in parallel without reaching each device's logging limitation. For example, one device is configured to focus on subjects of system daemons, while another device focuses on application subjects. Then all logging results are aligned and merged together.

5.4.1.2 Kernel-layer Access Event Collecting

SEAndroid uses a Linux security module loaded into kernel with the policy rules to check and log native-layer access events. However, in our case, such policy-rule-based logging mechanism has a major drawback that its completeness and granularity are heavily affected by how the security labels and rules are defined in an existing policy. Critical access events can be easily missed or confused if coarse-grained labels are assigned to different subjects/objects. For instance, existing policy assigns some application processes with coarse-grained domain labels without package names, causing different apps to be indistinguishable.

As our goal is to collect sufficient access pattern knowledge for policy analysis, the logging mechanism itself should be independent from any existing policy. To this end, we modify Linux kernel and design a policy-less logging mode that supports fine-grained access event logging. To distinguish different subjects and objects, unique labels are derived for every process subject based on the process's executable binary. Fine-grained file object labels are derived based on their absolute file paths. We modify the kernel to assign these fine-grained labels without relying on a policy. For Android applications, we log their process and user ids with timestamps and correlate with package names logged in native layer (Section 5.4.1.4). We also configure each device to focus on specific fine-grained subjects/objects to distribute the logging work load.

Note that, after all access events are collected and merged, we de-duplicate and transform them into more structured access patterns as mentioned in Section 5.2.1. In addition, since no malicious accesses are assumed during functional testing, the policy-less logging skips rule-based permission checking and directly dumps all access events to the logging buffer.

5.4.1.3 Dalvik-layer Functionality Tracing

As mentioned above, functional tests inherently carry rich semantics of functionalities under tests. The functionality execution contains descriptive items that can be collected as a functionality trace,
including metadata of functional tests, key API calls/control flow.

To collect such functionality traces in a systematic and automatic way, we place multiple hooks into existing Android testing framework to monitor the execution of a functional test, to obtain a detailed view of how the test proceeds. As shown in the top layer in Figure 5.2, this enables us to be aware of different phases in testing and focus on the phase when the target functionality is executing, while filtering out non-functional test `setUp`, `assertion` and `tearDown` phases.

To precisely capture the control flow within the target functionality, we further leverage a runtime method tracing facility in Dalvik (or ART) VM. Originally, this facility was designed to profile every method call’s time usage in Android framework [Deb]. We enhance it to be configurable to log specific Java classes and methods, which can focus on key functional APIs. For example, in our evaluation, we focus on core APIs listed in the MDM as the major descriptive items for functional knowledge (e.g., `android.app.enterprise.Firewall`).

### 5.4.1.4 Cross-layer Correlation via Native-layer Global Variables

Since logs from the kernel layer and the Dalvik layer are independently recorded in different forms, it is necessary to correlate high-level functionality traces and low-level access patterns together, so
5.4. SPOKE can store them as a unified form in the knowledge base.

As shown in Figure 5.2, native layer is the intermediate layer between Dalvik layer and kernel layer. As its main task is to transform high-level Java requests into low-level system calls, less semantics can be extracted from this layer. However, several global variables in this layer are of great importance for achieving cross-layer correlation. Such variables include wall-clock timestamps, process/thread ids, user ids and package names.

Specifically, wall clocks are globally available across all three layers. This enables logs collected from each layer to be aligned. Process/thread ids (pid) and user ids (uid) are also global variables in a device. When coupled with timestamps, they are able to index and correlate every specific logging event in both Dalvik layer and kernel layer. Package names are important information but missing from SEAndroid kernel logging. Fortunately, as system daemon zygote keeps track of every launched application, we instrument it to dump the package name, process and user id with precise timestamp whenever an application is launched, which are then correlated with access patterns in kernel logs.

In practice, the above global variables can be collected using Android shell commands (e.g., pm, busybox). The native layer is also a suitable place for the device farm manager to synchronize each device's state, such as loading logging configuration and resetting buffers.

5.4.1.5 Non-Functional Event Filtering

For most functional tests, Android devices are required to be in the same state as if operated by normal users. This means that built-in system applications and daemons are actively running in the background during the testing. For example, system_server periodically checks background status such as WiFi and battery. Unfortunately, such background activities could introduce noise to SPOKE's correlation, especially in kernel-layer logs.

To distinguish background activities, before running any tests, we perform a long-period logging on devices in the idle state to identify background access patterns in kernel-layer logs and its native-layer processes based on their periodic occurrences. Then during functional tests, these background access patterns are put in a filter list of the logging configuration so that each device can skip them in the logs.

Access patterns triggered by test-only operations should be filtered as well. These access patterns are not related to the actual functionality but only caused by the phases of test preparation and cleanup. As shown in Figure 5.2, by correlating with test-only methods (setUp, tearDown) in Dalvik layer, we perform a dummy test to explicitly capture those non-functional access patterns and filter them during functional tests.
5.4.2 Policy Rule Justification w.r.t. K

We design an analytic engine to use the knowledge base for policy rule analysis. The first analytic usage is to match policy rules with collected functionality trace to help policy engineers justify the rules.

Intuitively, if a policy rule is defined to allow a set of access patterns, which are found correlated with a set of functionality traces in the knowledge base, we say that these access patterns of this rule are justified by the corresponding functionalities. If the policy rule defines some access patterns that have no correlated functionality trace, then these access patterns are unjustified and subject to attack surface analysis discussed in Section 5.4.3.

Recall Definition 5.6, given a policy rule $r$, we further denote the justification result as $J_r = \{(a_r, f_k) \mid a_r \in r.A, a_r = a_k, \text{Correlated}(a_k, f_k) = True\}$. $J_r$ contains every justified access pattern $a_r$ defined by the rule $r$, paired with the corresponding functionality trace $f_k$. An access pattern $a_r$ is justified if it is matched with an observed access pattern $a_k$ in a pair $(a_k, f_k)$ in the knowledge base.

5.4.2.1 Similar Access Pattern Generalization

Theoretically, to justify an access pattern $a_r$ defined in a policy rule $r$, $a_r$ should exactly match with an access pattern $a_k$ collected in the knowledge base with the exact same subject (i.e., $s_r = s_k$), object, class, and permission. However, in practice, multiple access patterns triggered by the same functionality could be slightly different but semantically equivalent. One example is the auto-generated files or pseudo file system (/proc/pid), whose file names are generated with randomness but semantically the same. Therefore, they should be generalized based on their file paths, so that they can be matched as equivalent.

We develop $General(a)$ to realize the generalized matching $General(a_r) = General(a_k)$. Given an access pattern, we generalize its subject, object and permission based on the following empirical rules: (1) all process subjects from the same Android application is generalized to the same application subject; (2) auto-generated file objects are generalized by removing the random parts while keeping the static parts in their file paths (e.g., /proc/1234/stat $\Rightarrow$ /proc/pid/stat). (3) similar permissions of an object class are generalized as one set (e.g., (write, append) $\Rightarrow$ write-like for file).

These rules are derived and extensible based on empirical experience and facts about Android file system hierarchy (e.g., shared prefix on file paths) and “macros”, a policy language feature in SEAndroid used by policy engineers to group similar permissions. In practice, the generalization can be applied when access patterns are being stored into the knowledge base or parsed from policy
rules to save the effort of matching.

5.4.2.2 Justification by Knowledge Base Querying

As mentioned in Section 5.5, access patterns act as the semantic bridge connecting both policy rules and functionality trace. This enables two ways of policy rule justification. In one way, given one policy rule, we can justify the rule by matching its defined access patterns with corresponding functionality trace. In the other way, given a tested functionality, we can identify all policy rules whose defined access patterns are correlated with this functionality. In practice, both cases help policy engineers check whether policy rules are consistent with corresponding functionalities.

We realize both cases using SQL queries to the knowledge base with a set of constraints. To justify a given policy rule $r$, the query (standard SQL with pseudo code constraints in WHERE and ON clauses) is:

$$
J_r \leftarrow \text{SELECT } A_r.a, K.a_k \text{ FROM } K, r.A_r \text{ WHERE } \text{General}(A_r.a) = \text{General}(K.a_k) \text{ AND } \text{Correlated}(K.a_k, K.f_k) = True
$$

This query realizes the justification definition and takes into account the similar access pattern generalization.

To identify all related rules of a functionality trace $f$, the query is:

$$
R_f \leftarrow \text{SELECT } T.a_k, R.r \text{ FROM (SELECT } K.a_k \text{ FROM } K \text{ WHERE } \text{Correlated}(K.a_k, f) = True) \text{ AS } T \text{ LEFT JOIN } R \text{ ON } \text{General}(R.r.A_r.a) = \text{General}(T.a_k) \text{ [WHERE } R.r \text{ IS NULL]}
$$

This query first extracts all correlated access patterns of the given functionality trace $f$ into an intermediate table $T$. It then uses a LEFT JOIN to match every $a_k$ in $T$ with rules in $R$ whose access patterns can match $a_k$. Policy engineers can also use the optional WHERE clause to further identify the access patterns that current rules cannot cover (e.g., for a newly developed functionality).
5.4.3 Attack Surface Analysis of Policy Rules

The second task of the analytic engine is attack surface analysis. It identifies risky policy rules that allow unjustified w.r.t. \( K \) and over-permissive access patterns.

5.4.3.1 Unjustified Access Patterns in Policy Rules

Ideally, every well-defined policy rule can be justified when every functionality is tested and all access patterns are collected. However, in reality, the above justification process often reveals some policy rules whose defined access patterns cannot be justified by current knowledge base. This is due to two reasons: 1) incomplete functional test coverage, 2) mistakenly developed policy rules.

The first case, as mentioned in Section 5.2.3, can be mitigated by various coverage-measuring tools [Emm; Jac] that ensure the high quality of functional tests. Research efforts like test generation [Mac13; HN11] can also be used. Since SPOKE uses high-covered functional tests, we found that the majority of unjustified access patterns are caused by the following case.

The second case, as mentioned in Section 5.1, is due to policy engineers’ knowledge gap and the conservative approach of developing over-permissive policy rules such as using default/coarse-grained labels [Res15] to avoid breaking uncertain functionalities. This causes the rules to allow unnecessary access patterns, which would never be justified by any functionality.

No matter which case, if the unjustified access patterns defined by certain rules can be potentially exploited by attackers to achieve privilege escalation, they need to be identified and fixed by policy engineers. Hence, we design an attack surface analysis to pinpoint such over-permissive access patterns and the corresponding rules.

5.4.3.2 Attack Surface Analysis

Originally, an attack surface is defined as the entry points accessible to attackers in three dimensions: targets, channels and access rights [How05; MW11]. The case of SEAndroid policy falls in the dimension of access rights. Access patterns defined by policy rules are the concrete representation of the access rights between subjects and objects.

The attack surface analysis has two steps. First, it selects the defined access patterns with their rules that are unjustified by current knowledge base. Second, it identifies over-permissive access patterns that allow potentially vulnerable subjects to access valuable or critical objects.

The first step is achieved by a SQL query to subtract the set of collected access patterns in the
knowledge base from the set of defined access patterns by the rules:

\[
U \leftarrow \text{SELECT } R.r, R.r.A.r, a_r \text{ FROM } R \text{ LEFT JOIN } K \\
\text{ON } \text{General}(R.r.A.r, a_r) = \text{General}(K.a_k) \\
\text{WHERE } K.a_k \text{ IS NULL}
\]

The query first uses \textit{LEFT JOIN} to attempt to match every rule \( r \in R \) and its access pattern \( a_r \in r.A_r \) with an access pattern \( a_k \) in the knowledge base. Then it filters the join result with the \textit{WHERE} clause to only select the set of \( a_r \) that have no matched \( a_k \) (\( a_k \text{ IS NULL} \)).

The second step is to identify over-permissive access patterns from the result of the first step. Based on Definition 5.7, we start with identifying valuable and critical objects which could be attackers’ potential targets. Fortunately, as we collect domain knowledge from tests of critical functionalities, such as the MDM security functionalities in our evaluation, the knowledge base already collects some valuable or high-privileged objects, which we can identify based on the correlated functionality trace. Then we search all unjustified access patterns that allow to access these critical objects as the over-permissive access patterns.

In practice, policy engineers can also input extra knowledge to guide the above searching. For example, if a subject has a new vulnerability, we can search all unjustified access patterns related to the subject and check whether any valuable objects are accessible by the subject. Starting from Android 6.0, multi-level security (MLS) [McC87] will be introduced to SEAndroid. The new knowledge of different privileged subjects and objects can also be leveraged to guide the searching in future.

To present a more intuitive result of the identified over-permissive access patterns for policy engineers, we model the analysis as a bipartite graph shown in Figure 5.3, where the vertices of all subjects are on one side shown as red and the vertices of all objects are on the other side shown as
blue. Edges labeled with access permissions represent access patterns between subject and object vertices. Justified access patterns are grey lines. Over-permissive access patterns are highlighted as red lines. Here, a vulnerable keyboard app is allowed to access critical system files. Later, we use this bipartite graph to present a real-world findings in Section 5.5.4.

## 5.5 Evaluation

We implement a prototype of SPOKE using 3.8K SLOC Python and 2K SLOC Impala SQL on a 8-node Hadoop cluster, using a device farm with 4 devices from a major Android vendor running Android 5.1.1. This experiment environment allows us to perform a thorough analysis on the evaluation results with the help of policy engineers. In practice, SPOKE can easily scale up to a bigger device farm and a production cloud.

We evaluate our prototype of SPOKE using the following functional test set, which has a moderate scale for us to verify the result. Note that, by design, SPOKE can work with any Android functional tests as long as the functionality requires SEAndroid access control. We first show the construction of the knowledge base. Then we present a case study of using the analytic engine to match policy rules with MDM domain knowledge, followed with a real-world finding by the attack surface analysis.

### 5.5.1 Data Set and Research Questions

SPOKE is designed to leverage the knowledge in functional tests to help policy engineers. To evaluate the effectiveness of SPOKE in real world, we use a suite of functional tests of a Mobile Device Management (MDM) framework developed by a major Android vendor.

The MDM functional test suite is a representative case that meets SPOKE’s goal and assumptions. MDM is an advanced device administrative framework added to Android OS for security-critical use cases in enterprise. It provides administration functionalities such as deploying enterprise applications, enforcing firewall policies and protecting enterprise data access. We choose MDM tests to evaluate SPOKE because MDM is a major target of access control protection by SEAndroid policy. The policy rules require careful development to ensure the low-level access patterns are enforced and consistent with the high-level MDM functionalities. Otherwise, attackers may compromise the backend of MDM and bypass its control.

The MDM functional test suite contains 665 functional tests covering 90% APIs from 28 MDM functionality categories such as enterprise application management, firewall/vpn/wifi configuration, and exchange account management. Using this functional test suite, we evaluate SPOKE by asking three research questions:
5.5. EVALUATION

R1: What domain knowledge is collected from MDM functional tests to build the knowledge base? What is the time and space cost of this process?

R2: How many justified and unjustified policy rules related to MDM functionalities are revealed by the collected domain knowledge?

R3: What over-permissive access patterns allowed by risky policy rules in real world are discovered by SPOKE’s attack surface analysis?

5.5.2 Knowledge Base Construction from MDM Functional Tests

The knowledge base construction is divided into two phases: 1) Collecting non-functional activities for filtering in functional tests; 2) Running functional tests with multi-layer logging to extract access patterns and functionality trace with cross-layer correlation.

In the first phase, we perform 10-round collection of non-functional access patterns on the four devices in idle state. In each round, each device is factory-resetted and rebooted. After the device is booted into home screen, the logging starts and lasts for an hour, during which the device is left untouched on home screen. Similarly, we also run 10-round dummy test (test_Dummy with empty function body) to get the non-functional access patterns triggered by JUnit setUp and tearDown. In total, we collect 896 background and dummy test access patterns, which are filtered in the next phase. Table 5.1 shows some examples of access patterns from these two runnings. Several daemons and apps such as android.bg, dhcp periodically check status of device processes and network. Binder IPC between system daemons and apps are also common and expected. installd
5.5. EVALUATION

<table>
<thead>
<tr>
<th>Subject</th>
<th>Access Permission</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.bg</td>
<td>read</td>
<td>file:/proc/pid/stat</td>
</tr>
<tr>
<td>dhcp</td>
<td>read,write</td>
<td>file:/proc/sys/net/ipv4/conf</td>
</tr>
<tr>
<td>bluetooth_app</td>
<td>read</td>
<td>file:/data/misc/bt_config.xml</td>
</tr>
<tr>
<td>system_server</td>
<td>call, transfer</td>
<td>binder:service manager</td>
</tr>
<tr>
<td>appdomain</td>
<td>call, transfer</td>
<td>binder:system_server</td>
</tr>
<tr>
<td>installd</td>
<td>execute</td>
<td>file:/system/bin/dex2oat</td>
</tr>
<tr>
<td>test_app</td>
<td>read</td>
<td>file:/data/local/log_config</td>
</tr>
</tbody>
</table>

operations happen during test_app installation. The loading of logging configuration is also captured and filtered.

In the second phase, we run the 665 functional tests in 28 categories on the four devices. In each running, all devices are resetted and rebooted as the same above. In the kernel-layer logging, two devices are configured to log access events of various system daemons, while the other two are configured to focus on access events of all Android applications. In the Dalvik-layer logging, all devices are configured to focus on android.app.enterprise.* classes and methods which are the key classes in MDM functionalities.

The total number of functional access patterns collected in the knowledge base is 12,491, with a total of 1,492 android.app.enterprise.* methods extracted as functionality trace. Figure 5.5 shows the average logging rate of raw access events per second, and test running time of the 28 functional test categories. The 12,491 unique access patterns are actually filtered, derived and de-duplicated from 481,216 raw access events. Given the test running time, we found that the highest logging rate is 1,005 raw access events per second (ApplicationPolicyTest produces 76,578 raw access events in 76.14 seconds). Thanks to the logging work load is distributed to four devices, we are able to scale up to this rate without hitting the logging buffer limitation.

Figure 5.4 shows the overall summary of the knowledge base by the 28 functionality categories. By checking the detailed access patterns and their correlated functionality trace, we found that some functionality categories have more operation steps and involve different subjects, causing more access patterns to be collected. In particular, ExchangeAccountPolicyTest has 1,020 access patterns since they involve multiple steps such as typing account information using UI interaction, creating and encrypting the account, which includes multiple file operations. RestrictionPolicyTest has the most number of access patterns. It actually tests a collection of various types of common operations under restriction mode such as installing whitelisted packages.
5.5. EVALUATION

Figure 5.5 The average logging rate and running time of 28 MDM test categories in SPOKE knowledge extraction platform

configuring limited network settings. Such restricted operations involve permission checkings from MDM admin subjects across multiple different objects and functionalities, thus causing more access patterns under the hood.

We also found that there are 142 access patterns and 32 functionality trace shared across all 28 functionality categories, showing that they are the core part in MDM. For instance, access patterns that MDM subject system_server read & write two critical file objects (names are anonymized for confidentiality), are two core access patterns captured in all functionality categories. They are correlated with functionality traces EnterpriseManager and DeviceAccountPolicy under android.app.enterprise. This finding is confirmed by MDM developers that the above two file objects are the core MDM configuration files.

5.5.3 Justifying Policy Rules w.r.t. MDM Domain Knowledge

With the above knowledge base, we match MDM-related SEAndroid policy rules with corresponding functionality trace to justify the access patterns defined in these rules.
5.5. **EVALUATION**

<table>
<thead>
<tr>
<th>Justification Result</th>
<th>Num Rules</th>
<th>Rule Characteristics &amp; Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Justified</td>
<td>187</td>
<td>Fine-grained labels of subjects and objects with privileged classes (e.g., chr_file, netlink)</td>
</tr>
<tr>
<td>Partially Justified</td>
<td>198</td>
<td>Rules defined using attribute group (e.g., domain, system_domain, appdomain)</td>
</tr>
<tr>
<td></td>
<td>269</td>
<td>Coarse-grained labels for application subjects (e.g., platform_app, system_app, vendor_app)</td>
</tr>
<tr>
<td></td>
<td>382</td>
<td>Default labels for different file objects (e.g., system_file)</td>
</tr>
<tr>
<td>Unjustified</td>
<td>320</td>
<td>Irrelevant subjects accessing MDM objects</td>
</tr>
</tbody>
</table>

With the help of policy engineers and developers, we first identify in total 1,356 MDM-related allow rules in the policy of the vendor’s device running Android 5.1.1. These rules are identified because they allow access patterns whose subjects or objects are involved in MDM functionalities.

Then SPOKE’s analytic engine uses the SQL queries and access pattern generalization mentioned in Section 5.4.2 to attempt to match these rules with the knowledge base. Table 5.2 shows the summary and reasoning of the justification result. In all, 1,036 policy rules (Justified + Partially Justified) are matched with the total 12,491 access patterns in the knowledge base. Figure 5.4 shows the number of policy rules in each functionality category respectively. There are also 320 rules that SPOKE cannot find corresponding access patterns in the knowledge base. We further take a deep analysis of these rules and present our findings of the rules’ characteristics and potential problems for each category of the result.

**Justified Rules** There are 187 policy rules, in which every defined access pattern is matched with an access pattern collected in the knowledge base, and thus justified by corresponding functionality trace. These rules are typically written with fine-grained labels, which are one-to-one mapping to unique or privileged subjects and objects (e.g., in the target policy, label tz_user_device maps to /dev/trustzone_node).

In addition, the object classes in these rules are mostly privileged classes. The access patterns defined in such rules are very specific. As an example in the target policy rules, allow tz_daemon tz_user_device:chr_file {ioctl read write} only defines three access patterns between the TrustZone daemon and /dev/trustzone_node with three permissions, which are all found correlated with functionality trace of android.app.enterprise.Certificate.

**Partially Justified Rules** The majority of the rules are partially justified. Some access pattern defined by these rules are justified by the knowledge base but not all of them. We further find out three specific reasons.
5.5. EVALUATION

Firstly, 198 rules are defined using attributes. Attribute is an SEAndroid policy language feature that defines a group of labels [Sele]. Rules defined using attributes can involve a wide range of various subjects and objects. For example, domain is an attribute grouping all subject labels in a policy. allow domain logd:unix_stream_socket {connectto} allows any subjects to connect to a log daemon via unix socket.

Secondly, 269 rules are defined for Android applications but with coarse-grained subject labels. We found such coarse-grained labels are over-used to assign different privileged applications. For example, system_app is assigned to all applications with system user id, while only three of them are related to MDM. This causes the rules to be partially justified.

Thirdly, 382 rules use default labels for different file objects. Default file object labels (e.g., label system_file maps to /system/bin/*) are assigned to all files under /system/bin, while only few files are related to MDM. Some access patterns defined by the rules of accessing other files are not observed in the tests.

**Unjustified Rules** There are 320 unjustified rules. All access patterns defined by these rules are not justified in the knowledge base. Due to the same reasons as above, the rules use coarse-grained labels and thus define potentially unnecessary and over-permissive access patterns related to MDM subjects/objects. Such rules are subject to attack surface analysis.

To conclude, with the help of SPOKE, previously unclassified policy rules are now differentiated into different cases based on their justification results. This helps policy engineers analyze the rules with contexts, and prioritize the analysis on the unjustified access patterns. Some unjustified policy rules and access patterns exposed by SPOKE are still under analysis by both policy engineers and functionality developers. We are requested by them not to disclose the details due to potential security risks. We are planning for a long-term process to closely work with multiple teams to run new tests and analyze these unjustified (and partially) rules. Nevertheless, SPOKE prioritizes such important cases in policy analysis and also pushes the functional test development forward.

5.5.4 Analyzing attack surface of Policy Rules

For the partially justified and unjustified policy rules shown above, we conduct a case study to analyze their attack surface, and present our critical findings of over-permissive access patterns defined by these rules.

We first identify 5 critical file objects in one system directory. These system file objects are identified based on MDM’s security functionality trace of android.app.enterprise. These

\footnote{Since the analyzed policy is currently used in real-world Android devices, we are requested by the vendor of the used Android devices to anonymize some specific file names that contain critical data.}
system file objects contain device configuration, password and encryption keys of the security-related functionalities.

Then we find that there are 210 over-permissive access patterns from 106 policy rules that allow 94 unjustified subjects to read, write and even execute the 5 critical file objects. This is the first time of finding such problems in a real-world SEAndroid policy rules related to MDM functionalities with concrete evidence. The result has been confirmed by the developers and policy engineers. The policy rules have been revised to revoke these over-permissive access patterns in the updated policy.

Figure 5.6 shows a detailed bipartite graph illustrating the above attack surface analysis result. In the bipartite graph, we pick 11 easy-to-understand subjects (out of other vendor-specific and confidential subjects) shown as red nodes on the left, and 17 file objects shown as blue nodes on the right, including the 5 critical system files (top 5 anonymized node on the right). The grey edges
between subjects and objects are the justified access patterns. The red highlighted edges are the identified over-permissive access patterns defined by 10 rules related to the 10 subjects of the red edges (except the top one, which is a high-privileged MDM subjects).

These subjects are observed with normal and justified access patterns as grey edges with right-side objects. However, they are also allowed to access critical system files, which are unjustified and over-permissive. Attackers can exploit vulnerabilities in these subjects to compromise critical files via these access patterns. In particular, without prior knowledge of any vulnerabilities or attacks, SPOKE identifies mediaserver, which is the subject that was previously found having the notorious libstagefright vulnerability [Lib] (CVE-2015-1538). Attackers can first compromise mediaserver with this vulnerability as a step stone, and then use the over-permissive access patterns defined by a rule allow mediaserver ANONYMIZED_LABEL : file {ioctl read write create getattr setattr lock append unlink link rename open} to modify MDM critical system files and eventually control the enterprise device. This risky rule with other ones have been confirmed and removed by policy engineers.

5.6 Discussion

Native functional tests and other knowledge inputs Currently, SPOKE mainly focuses on dynamic analysis of Android functional tests for applications and framework whose functionalities involve multi-layer operations. However, functional tests for native executable binaries can also be used to extract domain knowledge for pure native functionality in an Android system. SPOKE can be enhanced with techniques such as ptrace/ltrace and native library hooking to achieve this feature, which we leave as future work. Other dynamic analysis techniques can provide useful domain knowledge as well. For example, dynamic taint analysis [Enc14] can provide detailed information flow of a series of access patterns. Static analysis such as symbolic execution [Mir12; Yan13] can identify code-level functionality and access patterns and provide extra knowledge of how access patterns and control flow are affected by specific inputs.

Data mining and machine learning possibilities We design an analytic engine in SPOKE to leverage the knowledge base for policy rule justification and attack surface analysis. Apart from these, other data mining and machine learning techniques can also be applied within the analytic engine. For example, outlier/anomaly detection [HA] can find suspicious or mistakenly defined access patterns from certain subjects or objects that are different from the majority of the access patterns in the knowledge base. Bayesian networks [Fri; Fri99] can also be applied for learning the relationship between access patterns and inferring whether a new access pattern defined by a new rule is likely to be justified or over-permissive.
5.6. DISCUSSION

Justified but exploited access patterns. It is possible that a justified access pattern defined by a policy rule could still be exploited by attackers. This can be caused by the deficiency in functionality design. However, if the policy rules are not over-permissive, the damage of the exploit can be contained with limited impact. Nevertheless, SPOKE can still help policy engineers quickly analyze the impact by identifying all accessible objects from the vulnerable subject. After the vulnerable functionality is patched by developers, SPOKE can take the new functional tests to update the knowledge base and re-assess the justification result and attack surface to help policy engineers refine the policy rules consistently.

Policy rule synthesis and generation. Ideally, when functional tests can cover all access patterns required by the target functionality (i.e., 100% coverage of required access patterns), it is possible to generate or synthesize policy rules by collecting all required access patterns from the functional tests. In practice, although it is difficult to collect every required access pattern of various functionalities for the entire Android system, it is possible to modularize this process to synthesize (or partially synthesize) a specific set of policy rules for specific functionality modules with full (or nearly full) test coverage. In fact, the desktop SELinux supports “loadable policy module”, which allows the policy to be managed on a modular basis [Selb], though it is not supported by SEAndroid yet.

User-based access pattern collection. As the SEAndroid policy is eventually deployed to user devices for access control enforcement, human users can also be asked to involve the testing and refinement process of SEAndroid policy development. With the user agreement of data collection during testing (e.g., private data anonymization and no deliberate malicious usage), access patterns representing device’s daily use can be collected to help synthesize and refine policy rules. Existing user-based testing is already available for pre-released Android applications (e.g., Google Play Store Beta Testing [Goo]). We envision that SEAndroid policy development can also benefit from similar user beta testings.

Attack surface analysis by comparing different policy versions. Since the SEAndroid policy is dynamically refined to adapt new access patterns for new functionality, it is important to see if a new policy update is correctly refined to allow new access patterns, while not increasing the attack surface by mistakenly allowing existing or new attacks. Previously, VulSAN [Che09] was a tool developed to compare the effectiveness of different MAC policies, such as SELinux and AppArmor, given the same attack scenarios. In SPOKE, since we introduced using bipartite graphs to model and visualize the access patterns defined by the policy, algorithms of graph comparison [Kou11] can be applied to analyze the different between two policies to identify whether the attack surface is increased or not.
5.7 Related Work

In general, SPOKE’s knowledge extraction platform is a dynamic analysis system for Android. Plenty of research efforts have been made in this field. DroidScope [YY12] proposed an emulation-based inspection to analyze both Java and native components of Android applications. CopperDroid [Tam15] also used QEMU and focused on system call analysis of Android malware. TaintDroid [Enc14] provided a dynamic taint tracking system for information flow analysis in Android. In our case, we require the domain knowledge from real devices since some security functionalities require hardware features, and thus cannot use virtualization-based approach. Besides, these approaches are mainly designed for malware analysis. We focus on a fundamental new problem of collecting domain knowledge for SEAndroid policy, and thus develop new techniques specific for knowledge extraction.

Although SEAndroid is relatively new, SELinux has been researched for years, such as SELinux policy analysis and verification [Zan04; Hic10; Ala08; Sas06; Jae04], policy comparison [Che09], policy visualization [Xu08], policy information flow integrity measurement [Jae03; Jae06; Sha06]. These work mainly analyzed SELinux reference policy itself, which has been refined by the community for years. In contrast, SEAndroid policy is fairly new and under active development by vendors. It is necessary to analyze SEAndroid policy together with the original domain knowledge to ensure the labels and rules defined in the policy are consistent with the real case. In addition, by collecting and leveraging domain knowledge, SPOKE creates a new dimension to policy development and analysis.

EASEAndroid [Wan15] is a recent work that applied machine learning to analyze large-volume access events collected from user device logs to refine SEAndroid policy. SPOKE is orthogonal to EASEAndroid. EASEAndroid focuses on the post-deployment policy analysis to refine the policy against attacks in the wild. SPOKE focuses on the pre-deployment analysis of the policy to bridge the knowledge gap for policy engineers during policy development and analysis. SPOKE can help policy engineers have better understanding and analysis of the developed policy in the first place before the policy is deployed to user devices. Nevertheless, the knowledge from both SPOKE and EASEAndroid can be shared with each other to provide better analytic results.

5.8 Summary

SEAndroid policy development and analysis require domain knowledge. In this chapter, we presented SPOKE, an analytic knowledge engine that automatically collects domain knowledge from functional tests, and provides attack surface analysis through policy rule justification. We evaluated SPOKE using real-world functional tests. SPOKE successfully collected detailed domain knowledge.
It also revealed over-permissive rules, helping policy engineers analyze and revise the policy.
CHAPTER

6

FUTURE WORK AND CONCLUSION

6.1 Future Work

We believe the new exploration, techniques and results presented in this dissertation can be further extended and generalized for various real-world use cases. However, security policy development in computer systems keeps evolving with new technologies and challenges. It requires continuous and innovative endeavors. We envision that new possibilities can be further explored based on the efforts made in this dissertation.

6.1.1 Big Data Analytics for Security Policy Development

Given the increasing scale of various systems and security policy deployment, more and more data requires to be analyzed for security policy development. In our second work, we proposed EASEAndroid to analyze millions of SEAndroid audit logs collected from user devices for SEAndroid policy refinement. In our third work, we proposed SPOKE to collect and leverage domain knowledge, which creates a new dimension for security policy analysis. We believe these efforts point out a new direction of utilizing big data analytics for security policy development.

Big data analytics[Man11] includes data warehouse infrastructure and data mining techniques, have become prevalent recently and are applied to various fields. However, it is non-trivial to apply
big data analytics to security policy development. Particularly, it requires careful feature selection
to correctly model and represent the semantics of security policy and its deployed system. In
EASEAndroid and SPOKE, we modeled the accessing behaviors enforced by SEAndroid policy as
access patterns. We envision that other valuable characteristics of the accessing behaviors can also
be used, such as the temporal and causal relations between different accesses.

In practice, the effectiveness and usability of security policy may variate in different use cases
and user’s subjective preference. Previously, several research efforts have studied the usability of
security policy [Sch11; Bra07]. Since today’s security policy such as SEAndroid may affect millions of
users, it becomes increasingly critical to take into consideration a huge amount of user’s feedback
and usability reports for better policy development.

6.1.2 Artificial Intelligence for Security Policy Development

Apart from the scalability, we are also seeing the opportunities of applying artificial intelligence (AI)
and machine learning to security policy development. Currently, policy engineers are responsible
for all major tasks (i.e., writing, refining and analyzing policy) in the lifecycle of security policy
development. With recent advances in AI and machine learning, it is possible that the manual and
heavy work load of policy engineers can be automated. Even the final decision making can be (semi-
)automated by intelligent machines. In EASEAndroid, we explored this possibility by designing three
machine learning algorithms, while more algorithms (i.e., SVM [Joa99], Adaptive boosting [Fre99],
Random Forest [LW02]) can be further leveraged and adapted to enhance the learning (i.e., classifica-
tion on benign and malicious access patterns) and eventually enable EASEAndroid to be a decision
maker for security policy refinement.

Recently, deep learning [Are10] (and deep neural network) has become one of the most hottest
fields in machine learning, particularly in image and natural language processing. Researchers
also start to explore the possibility of using deep learning to address computer security challenges.
[Yua14] proposed to use deep learning for Android malware detection. [Xia15] proposed to use
Artificial Neural Networks on Co-occurrence Matrices to find correlation of system calls, in order to
classify benign and malicious Android applications. We believe that given the large-scale rich data
resources related to security policy, deep learning also has the potentials to be applied to security
policy development.

6.1.3 New Attacks Against Security Policy Development

With the above mentioned new techniques and opportunities, we are also aware of the adversary side,
who could possibly leverage these techniques in a malicious way to undermine the effectiveness or
increase the difficulty of security policy development. In our first work, we proposed MetaSymploit targeting a new type of attacks in the form of attack scripts, which can be rapidly developed and distributed in a faster speed and larger scale than the deployment of security patches and policies. Other new-type and large-scale attacks (e.g., attacks against industrial control systems (ICS) and internet of things (IoT) [Zhu11; Rom11; MV14], large-scale DDoS [Nuc09], cyber warfare [AW13]) draw increasing attentions recently. How to create and develop security policies and access controls to defend against these new attacks is still an open question.

In addition, new techniques and algorithms designed for security policy development can also become the targets of new attacks. For example, as we apply machine learning algorithms to analyze data logs related to security policy, data poisoning attacks have emerged to fool the learning algorithms to deliberately escape from the access control of the policy [KL07; Big12]. Therefore, it is important to ensure the quality and validity of input data. Another possible countermeasure is to apply ensemble learning that groups a set of different learning algorithms analyzing different features or dimensions to dilute the data poisoning and increasing the cost of launching poisoning attacks.

6.2 Conclusion

In this dissertation, we hypothesized that the development lifecycle of security policy can be automated to help reduce human policy engineers’ workload and ensure the high quality of the security policy. We proved this hypothesis from three specific perspectives: policy generation, policy refinement and policy analysis.

In the first work, we presented MetaSymploit for automatic attack script analysis and IDS security policy generation. MetaSymploit focuses on a new type of attacks written in scripting languages, which can be quickly created and spread to exploit zero-day vulnerabilities. MetaSymploit can automatically analyze newly created attack scripts and generate IDS policy rules to defend against the attack scripts. We evaluated MetaSymploit using 45 real-world attack scripts from Metasploit, the most popular attack framework. The results showed the effectiveness of MetaSymploit in real-world attacks, and also the practical use in improving current IDS policy rules.

In the second work, we presented EASEAndroid, the first SEAndroid analytic platform for automatic policy analysis and refinement. EASEAndroid focuses on mandatory access control (MAC) policies in security-enhanced Android (SEAndroid), a new security feature in Android systems. Developing SEAndroid policy is a challenging task, requiring many iterations of refinement. EASEAndroid models and automates the policy refinement innovatively using semi-supervised learning. We evaluated EASEAndroid on 1.3 million audit logs from real-world Samsung devices. EASEAndroid
successfully learned 2,518 new access patterns and also discovered two new attacks in real world
directly targeting SEAndroid MAC mechanism.

In the third work, we presented SPOKE, an SEAndroid Policy Knowledge Engine that addresses
the challenge of the knowledge gap between policy engineers and functionality developers. SPOKE
automatically extracts domain knowledge from rich-semantic functional tests. Then SPOKE uses
the knowledge to provide policy rule justification and attack surface analysis. We evaluated SPOKE
using 665 functional tests targeting a set of security functionalities developed by a major Android
vendor. SPOKE successfully collected 12,491 access patterns as domain knowledge, and used the
knowledge to reveal 320 unjustified policy rules and 210 over-permissive access patterns defined by
those rules. These findings have been confirmed by policy engineers.

The new exploration, techniques and results presented in this dissertation point out the new
challenges and opportunities in security policy development, and can be further extended for
various real-world use cases of different security policies.


<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snoa</td>
<td>SNORT.</td>
<td><a href="https://www.snort.org/">https://www.snort.org/</a>.</td>
</tr>
</tbody>
</table>


APPENDICES
A.1 Example of Rule Comparison

```ruby
def exploit
  ...
  trigger = '/ldap://localhost/%3fA%3fA%3fCCCCCCCCCC%3fC%3f%90'
  # Sending payload
  send_request_raw({
    'uri' => '/',
    'version' => '1.0',
  }, 2)
  ...
end
```

Listing A.1 The code snippet from a Metasploit attack script `apache_mod_rewrite_ldap.rb`
A.1. EXAMPLE OF RULE COMPARISON

Listing A.2 One MetaSymploit Rule (MR) for an attack payload of `apache_mod_rewrite_ldap.rb`.

```plaintext
alert tcp any any -> any 80 (  
  msg: "Metasploit apache_mod_rewrite_ldap, 
  Target: [Apache 1.3/2.0/2.2], 
  Behavior: [HTTP request with Vul-specific bytes]");  
  content: "GET";
  content: "/ldap|3A|//localhost/%3fA%3fA%3fCCCCCCCCC%3fC%3f%90";
  refence: cve,2006-3747;
  sid: 5000539; rev: 0;)
```

Listing A.3 One Official Snort Rule (OR) related to the Metasploit attack script in Listing A.1.

```plaintext
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (  
  msg: "WEB-MISC Apache mod_rewrite buffer overflow attempt";
  content: "GET";
  content: "/ldap|3A|";
  pcre: "/(ldap|x3A|x2F|x2F[^\x0A]*(%3f|\x3F)^\x0A)*(%3f|\x3F)^\x0A)*(%3f|\x3F)^[\x0A]*(%3f|\x3F)/sm1";
  reference: cve,2006-3747;
  sid: 11679; rev: 5;)
```

In Appendix A, we give a simple example to illustrate the comparison between an official Snort rule containing general patterns with a MetaSymploit rule containing specific patterns.

Listing A.1 shows the code snippet of the `exploit` method in the Metasploit attack script `apache_mod_rewrite_ldap.rb`. The script launches the attack by sending an HTTP GET request packet that contains a special URI byte string to trigger the vulnerability. Here `send_request_raw` is a Metasploit HTTP output API method that is symbolically extended by MetaSymploit to dump the entire payload packet.

Listing A.2 is a MetaSymploit Rule (MR) based on the attack payload composed by the script. It contains the constant byte string patterns, especially the vulnerability triggering string that can identify the specific payload packet. Listing A.3 is the corresponding Official Rule (OR) based on CVE matching. It contains a regular expression (regex) pattern generalized by security analysts based on large amounts of samples.

According to the Snort rule manual, a rule can have multiple `content` byte string patterns. By default, given a packet, Snort searches these `content` patterns in order. A rule can also have one `pcre` regex pattern. Snort searches the entire packet for the `pcre` pattern.

In the example rules, the first `content` in both rules share the same pattern “GET”. The second `content` of the MR captures the triggering string, which includes the second `content` of the
OR "ldap\|3A\|" as a substring. Furthermore, the second content of the MR is also matched by the general pcre regex of the OR. In addition, there is another content in the MR that captures the HTTP protocol version of the packet.

Although both rules can detect the attack payload of this script, the MR has multiple specific patterns that can precisely pinpoint the attacks launched by this script, thus having very low false-positive rate compared to the general OR. In practice, the MRs can help identify what attack scripts are used by attackers, providing a way for the defense side to profile and obtain more knowledge of the attacker side.
APPENDIX

B

EASEANDROID

B.1 Audit Log Collection

Collection of audit logs used in this research strictly followed the Privacy Policy of the vendor with whom we collaborated, and conformed to the conditions described in the User Agreement. Logs were collected anonymously, and are used to provide better security for customers. Only employees of the vendor had access to logs, using the vendor's computer systems. No individual audit log information was released to non-employees.