SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android

Ruowen Wang[1] Ahmed M. Azab[1] William Enck[2] Ninghui Li[3]
Peng Ning[1] Xun Chen[1] Wenbo Shen[1] Yueqiang Cheng[1]
[1]Samsung Research America
{ruowen.wang, a.azab, peng.ning, xun.chen, wenbo.s, y.cheng}@samsung.com
[2]North Carolina State University [3]Purdue University
whenck@ncsu.edu ninghui@purdue.edu

ABSTRACT

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 systematically 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 28 different categories of functionalities developed by Samsung Android Team. SPOKE successfully collected 12,491 access patterns for the 28 categories 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.

Keywords

Mandatory Access Control; Android; SELinux; SEAndroid;

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 [7]. 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 [35] 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 mistakenly impacting user experience are significant. Designing a good SEAndroid policy requires domain knowledge of the entire set of various Android functionalities, which also 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 the policy, cannot obtain the domain knowledge on every system functionality. At the same time, functionality developers usually lack SEAndroid and security 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 [34] that showed multiple Android devices have vendor-modified policies that are less strict and have wider attack surface than Google's baseline policy.

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3052973.3052991

In this paper, 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 constructs a knowledge base of functionality-required access patterns extracted from semantically rich functional tests. Second, SPOKE uses the knowledge base to identify potentially unnecessary access patterns but allowed by certain policy rules. Finally, SPOKE analyzes the attack surface of these policy rules by generating 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 Samsung Android Team, and evaluated it by taking inputs of 665 functional tests for 28 different categories of functionalities in Android framework, including application installation, bluetooth/WiFi/location/firewall management, etc. SPOKE successfully collected 12,491 low-level access patterns correlated with 1,492 high-level functionality traces of rich-semantic APIs as the domain knowledge. With this knowledge base, SPOKE first identified 1,036 rules out of a total of 1,356 relevant policy rules that are necessary (or partially so) to corresponding functionalities. 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 [5]. Policy engineers have confirmed the findings and revised the policy.

In summary, this work makes three contributions:

1. We propose SPOKE, a novel knowledge collection and analysis engine that bridges the knowledge gap between policy engineers and functionality developers.
2. We implement SPOKE by first building a knowledge extraction platform that systematically and scalably collects domain knowledge from rich semantic functional tests, and second, creating an analysis 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 provided by Samsung Android Team. 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 coverage of functional tests used to define the knowledge base. However, a perfect set of functional tests is not required to benefit from SPOKE. Even using functional tests with low coverage, SPOKE can still work. As test coverage increases, the value of SPOKE linearly increases by collecting more domain knowledge and analyzing more policy rules iteratively. Thus, we design SPOKE as a dynamic and scalable system for continuous operation. Empirically, with reasonable test suites used in the industry, SPOKE can provide valuable insights and new findings for policy engineers and functionality developers, as we show in Section 5.

```
type=1400 audit(1445635785.573:220):
avc:  denied  { write } for  pid=4685
comm="ContactsProvide" name="contacts.db"
path="/data/data/contacts_app/contacts.db"
scontext=u:r:untrusted_app:s0
tcontext=u:object_r:app_data_file:s0
tclass=file

type=1300 audit(1445635785.573:220):
syscall=25 success=yes exit=0 pid=4685
uid=10024 gid=10024 comm="ContactsProvide"
exe="/system/bin/app_process64"
subj=u:r:untrusted_app:s0
```

Listing 1: A simplified access event example recorded at the epoch time 1445635785.573 in an audit log, with two entries: subject & object with labels and permission (1400), syscall info (1300).

2. BACKGROUND AND DEFINITIONS

2.1 SEAndroid Basics

SEAndroid is a port of SELinux [38] to Android with extensions to support Android-specific features, such as Binder IPC [37]. The goal of SEAndroid is to reduce attack surface and contain damage if any vulnerability is exploited for privilege escalation, via MAC enforcement on native accesses (i.e., system calls) between subjects (e.g., processes) and objects (e.g., files, sockets) in Android system.

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 [28]. 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 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 that should not be allowed and are checked against `allow` rules at compile time. During runtime, if no `allow` rule can match an access event, the access event will be denied and logged [8] (Section 4.1.2 introduces a new way of logging access events).

An access event usually has two entries[1] as shown in Listing 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`.

Traditionally, policy engineers develop and refine policy rules by manually analyzing the access events. Only a few basic tools (e.g., `setools` [9]) previously in SELinux can be used for SEAndroid. Such tools can only perform manual and syntactic analysis with no domain knowledge. 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 coarse-grained security

[1]Previously, there was an object entry, which is merged into `1400`.

labels to be used inappropriately, which is one security issue mentioned in [34].

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 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 1 is `('contacts_app', '/data/data/contacts_app/contacts.db', 'file','write')`. The second case is explained as following.

Definition 2 (SEAndroid Policy). *A policy is $\mathbb{P} = (L_s, L_o, M, S, O, R)$, where L_s, L_o are the set of security labels of subjects and objects, $M : L_s \cup L_o \mapsto S \cup O$ is a mapping that assigns security labels to concrete subjects S and objects O, $R = \{r\}$ is the set of policy allow rules.*

In SPOKE, 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).

Now given a policy rule $r \in R$: "`allow` $\mathtt{l_s}$ $\mathtt{l_o}$: $\mathtt{c_r}$ $\mathtt{P_r}$", we further define it as the following.

Definition 3 (Policy Rule). *A 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 certain functionality) could be inconsistent with the access patterns defined by the policy rules, due to the knowledge gap, which SPOKE is designed to address.

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. With standard libraries such as AndroidJUnitRunner, UI Automator [1], Android functional tests are well organized and closely associated with design requirements and end user operations. This makes such tests self-explanatory and inherently carry rich semantics of the functionality under test. Examples include checking specific API functions (Unit test), clicking or typing on UI widgets (UI test), and setting up an email account (Integration test).

We hypothesize that Android functional tests can enable a systematic way to synchronize domain knowledge between developers and policy engineers, providing a knowledge foundation for the attack surface analysis of SEAndroid policy.

However, it is non-trivial to extract domain knowledge from functional tests. Android functional tests are originally designed to test high-level functional operations in Android application or framework layer, while low-level access patterns in native layer are implicitly involved and thus still obscure behind the scene. How to extract low-level functionality-related access patterns while excluding test-only/non-functional noise is one task that SPOKE is designed to address.

Impact of Functional Test Coverage: By design, SPOKE relies on functional tests as inputs. Test coverage can affect SPOKE's performance. However, test coverage is *orthogonal* to SPOKE, because one of our contributions is to leverage the outcomes of both industrial practice and research efforts in the field of software testing, to enhance the security analysis of SEAndroid policy.

Specifically, in the software industry, multiple coverage-measuring tools [3] are developed to ensure the high test coverage. As test-driven development (TDD) [13] is a popular software engineering practice, many Android testing tools and frameworks are actively used in the industry (e.g., Testdroid [11], AWS device farm [2]).

Increasing test coverage is also an active research topic in software testing [14, 17, 32, 41]. Various techniques have been developed for automated testing and test input generation of mobile applications. For example, Dynodroid [29] is an automated test input generation system for Android apps. Swifthand [16] 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 [27].

2.4 Definitions introduced by SPOKE

We further use the following definitions to introduce several new concepts used in this work.

Definition 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 code-level items can be collected as descriptive items:*

- *Metadata of a functional test, such as `test_class`, `test_case`, `@annotation` in a JUnit test*
- *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 first item shows a high-level functional operation under test, and the rest two items provide code-level details of 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 (Knowledge Base). *A knowledge base is a set of pairs $\mathbb{K} = \{(a, f)\}$, where 'a' denotes an access pattern extracted from runtime access events in kernel layer (Section 4.1.2) and 'f' denotes a functionality trace collected from Dalvik layer (Section 4.1.3).*

In practice, $\mathbb{K}$ is stored in a database. We can query the database to find all access patterns correlated with a given functionality trace f, i.e., $A_f = \{a \mid isCorrelated_{\mathbb{K}}(a, f) = True\}$, where $isCorrelated_{\mathbb{K}}(a, f)$ is used to denote whether they are correlated in $\mathbb{K}$ (used in Section 4.2.2).

A potentially unnecessary access pattern is one that is defined by a policy rule but not found in the knowledge base and thus cannot be justified by the knowledge base. To be more precise and formal, we define "*potentially unnecessary*" as "*unjustified w.r.t. the knowledge base*".

Definition 6 (Policy Rule Justification w.r.t. $\mathbb{K}$). *A policy rule r is said to be justified with respect to a knowledge base $\mathbb{K}$, if for each access pattern a_r defined in A_r, there is an equivalent access pattern a in $\mathbb{K}$, correlated with at least one functionality trace ($a_r = a$ and $isCorrelated_{\mathbb{K}}(a, f) = True$). A rule is said to be partially justified (or unjustified) if only a subset (or none) of A_r have equivalent access patterns in the knowledge base.*

As an example, by running a functional test `test_addFirewallRule()`, policy engineers are able to learn the domain knowledge based on a functionality trace:

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

correlated with the access pattern:

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

and thus justifies a policy rule `allow system_server iptables_exec:file {execute}`, which allows this access pattern.

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

Definition 7 (Over-permissive Access Pattern). *An over-permissive access pattern is an unjustified access pattern (s, o, c, p) defined by a policy rule that can potentially allow attackers to misuse or exploit the subject 's' to maliciously access the object 'o' with permission 'p', in order to compromise the confidentiality or integrity of 'o'* [2].

3. PROBLEM AND ASSUMPTIONS

Problem Statement: In this paper, we seek to reduce the attack surface of an SEAndroid policy by identifying potentially unnecessary, and particularly over-permissive access patterns allowed by the policy. To this end, we need to bridge the knowledge gap between policy engineers and functionality developers, which causes the necessary access patterns to be unclear. Although functional tests can be helpful with valuable domain knowledge, it is non-trivial to extract functionality-required low-level access patterns, as they are implicitly involved in such high-level tests with irrelevant noises. Once the domain knowledge is extracted, a system is also needed to match the knowledge with policy rules and provide attack surface analysis. SPOKE is designed to address the aforementioned problems.

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 functionality-required access patterns can be extracted by running such tests. As noted in Section 2.3, test coverage is orthogonal to SPOKE. We also assume that tests need to be 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 can incur native access patterns and thus are visible by low-level SEAndroid access control. This is the typical case for system functionalities in Android framework. 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.

4. SPOKE

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

C1: Building an up-to-date knowledge base of various functionalities and their corresponding access patterns to bridge the knowledge gap between developers and policy 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 potentially unnecessary and over-permissive access patterns that could be misused by attackers.

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 systematic and scalable domain knowledge collection. The collected domain knowledge can then help policy engineers 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.

Domain knowledge collection is achieved by a two-step process. First, 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.

[2] In practice, over-permissive rules lead to over-privileged subjects

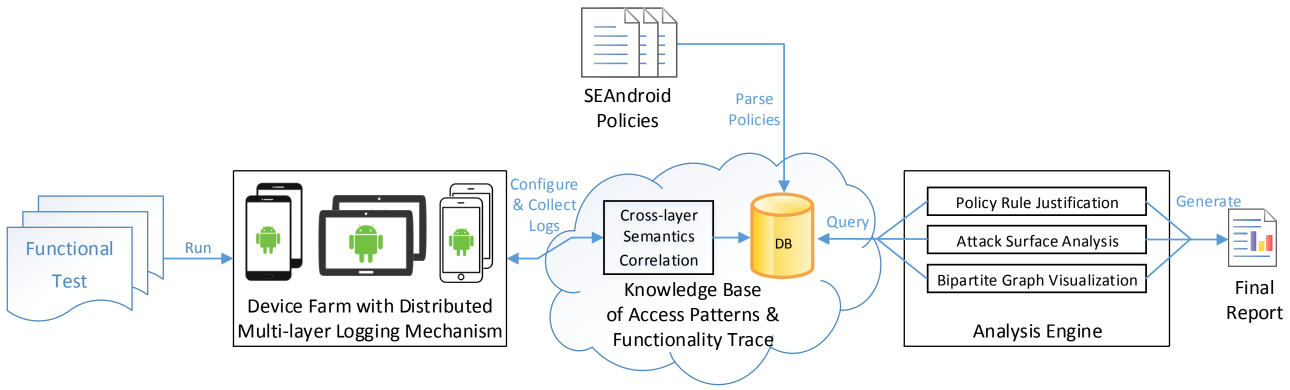

SEAndroid Policies
Parse Policies
Functional Test
Run
Device Farm with Distributed Multi-layer Logging Mechanism
Configure & Collect Logs
Cross-layer Semantics Correlation
DB
Knowledge Base of Access Patterns & Functionality Trace
Query
Policy Rule Justification
Attack Surface Analysis
Bipartite Graph Visualization
Analysis Engine
Generate
Final Report

Figure 1: SPOKE consists of three components from collecting test-driven tracing logs to generating final report by the analysis engine.

Second, 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 with 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. $\mathbb{K}$ is the first analysis capability designed to use 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 analysis capability that further focuses on the risky policy rules and their defined but unjustified access patterns revealed from the above step. 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, policy engineers use SPOKE to find valuable/critical objects in the knowledge base, based on correlated and rich-semantic 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 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 analysis engine provides both policy rule justification w.r.t. $\mathbb{K}$ and attack surface analysis. In practice, it also provides visualization to illustrate the attack surface results. The following sections explain more details of each component.

4.1 Domain Knowledge 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 sufficient logs from each layer to get a full picture of the functionality. Thus, SPOKE collects logs from three layers: Linux kernel layer, Dalvik VM layer and Android native layer with a distributed collecting mechanism, as explained in the following sections.

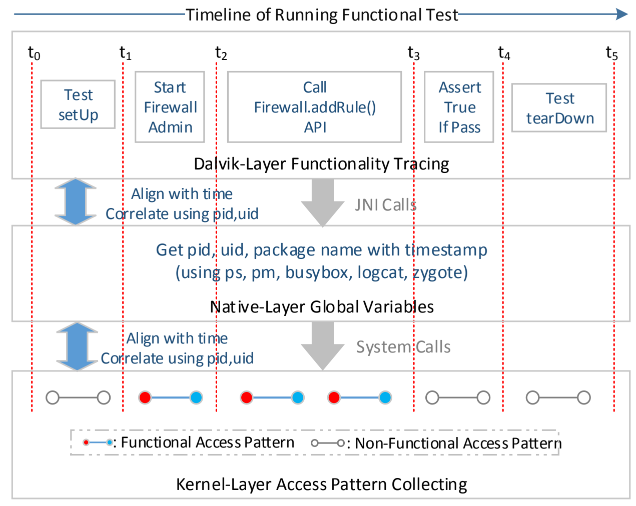
4.1.1 Distributed Multi-layer Collection

Collecting runtime logs is not a straightforward task, especially in the case of low-level access events, which are based on system calls. Given the high calling rate and large volume of system calls (i.e., GB-sized), 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. Even with the maximum setting of the device, critical access events are found to be missed in practice.

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 reduced 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. All logs are dumped directly through pipes and sockets to a desktop (or a cloud gateway), aligned and merged together.

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 directly 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.



Timeline of Running Functional Test
t0
t1
t2
t3
t4
t5
Test setUp
Start Firewall Admin
Call Firewall.addRule() API
Assert True If Pass
Test tearDown
Dalvik-Layer Functionality Tracing
Align with time
Correlate using pid,uid
JNI Calls
Get pid, uid, package name with timestamp
(using ps, pm, busybox, logcat, zygote)
Native-Layer Global Variables
Align with time
Correlate using pid,uid
System Calls
: Functional Access Pattern
: Non-Functional Access Pattern
Kernel-Layer Access Pattern Collecting

Figure 2: Cross-layer correlation between low-level access patterns and high-level functionality traces during a functional test `test_addFirewallRule()`.

As our goal is to collect sufficient access pattern knowledge for policy analysis, the logging mechanism itself should be independent from any existing policy. For this reason, 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 4.1.4). We also configure each device to focus on specific fine-grained subjects/objects to distribute the logging work load.

After all access events are collected and merged, we deduplicate and transform them into more structured access patterns as mentioned in Section 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 channel.

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 way, we place multiple hooks into existing Android testing framework to monitor the execution of a functional test, to obtain a detailed temporal view of how the test proceeds. As shown in the top layer in Figure 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 [6]. We enhance it to be configurable to log specific Java classes and methods, which can focus on key functional APIs as the major descriptive items of functionality traces (e.g., `android.app.enterprise.Firewall`).

4.1.4 Cross-layer Correlation via Native-layer Global Variables

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

As shown in Figure 2, native layer is the intermediate layer between Dalvik layer and kernel layer. Its main task is to transform high-level Java requests into low-level system calls. Although less semantics can be extracted from this layer, 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. When coupled with timestamps, they are able to index and correlate every specific logging event in both Dalvik layer and kernel layer in each process. 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.

4.1.5 Irrelevant Logging 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 2, by correlating with test-only methods (`setUp`,`tearDown`) in Dalvik layer based on the temporal phases, we can explicitly capture those non-functional access patterns and filter them during functional tests.

4.2 Policy Rule Justification w.r.t. $\mathbb{K}$

We design an analysis engine to use the knowledge base for policy rule analysis. The first analysis 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 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 4.3.

Based on Definition 6, given a policy rule r, we further denote the justification result as $J_r = \{(a_r, f) \mid a_r \in r.A_r, a_r = a, isCorrelated_{\mathbb{K}}(a, f) = True\}$. J_r contains every justified access pattern a_r defined by the rule r, that can be matched with an access pattern a correlated with functionality trace f in the knowledge base $\mathbb{K}$.

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 nondeterministically 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 only 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.

4.2.2 *Justification by Querying Knowledge Base*

In the knowledge base, access patterns act as the semantic bridge connecting policy rules with functionality traces. 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:

$$\begin{aligned} J_r \leftarrow\ & \texttt{SELECT } A_r.a_r\texttt{, } \mathbb{K}.f \texttt{ FROM } \mathbb{K}\texttt{, } r.A_r \\ & \texttt{WHERE } General(A_r.a_r) = General(\mathbb{K}.a) \\ & \texttt{AND } isCorrelated_{\mathbb{K}}(a, f) = True \end{aligned}$$

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

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

$$\begin{aligned} R_f \leftarrow\ & \texttt{SELECT } T.a\texttt{, } R.r \texttt{ FROM} \\ & \texttt{(SELECT } \mathbb{K}.a \texttt{ FROM } \mathbb{K} \\ & \texttt{ WHERE } isCorrelated_{\mathbb{K}}(a, f) = True\texttt{) AS } T \\ & \texttt{LEFT JOIN } R \\ & \texttt{ON } General(R.r.A_r.a_r) = General(T.a) \\ & \texttt{[WHERE } R.r \texttt{ IS NULL]} \end{aligned}$$

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 in T with rules in R whose access patterns can match a. 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).

4.3 Attack Surface Analysis of Policy Rules

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

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 2.3, is orthogonal to SPOKE and can leverage the outcomes of industrial and research efforts. Functionality developers can also identify what functional tests are missing based on these unjustified access patterns. The second case, as mentioned in Section 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 [34] 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 misused 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.

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 [23,30]. The case of SEAndroid policy falls

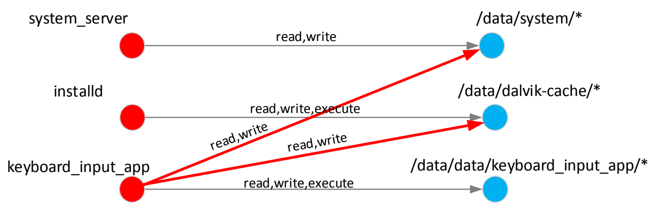


Figure 3: A bipartite graph illustrating two over-permissive access patterns (red) by a policy rule allowing a vulnerable keyboard app to read and write critical system files.

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:

$$\begin{aligned} U \leftarrow\ & \texttt{SELECT } R.r, R.r.A_r.a_r \texttt{ FROM } R \texttt{ LEFT JOIN } \mathbb{K} \\ & \texttt{ON } General(R.r.A_r.a_r) = General(\mathbb{K}.a) \\ & \texttt{WHERE } \mathbb{K}.a \texttt{ IS NULL} \end{aligned}$$

The query first uses `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 in the knowledge base. Then it filters the join result with the `WHERE` clause to only select the set of a_r that have no matched a (a `IS NULL`).

The second step is to identify over-permissive access patterns from the result of the first step. Based on Definition 7, we start with identifying valuable or critical objects which could be attackers' potential targets. Fortunately, since we collect domain knowledge from tests of critical functionalities, the knowledge base already captures the critical objects, which can be identified by the correlated critical functionality trace. For example,

`Firewall.addRule()<=>/data/system/firewall.db`

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) [31] is introduced to SEAndroid. The new knowledge of different privileged subjects and objects can also be leveraged to guide the searching in practice.

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 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. We use this bipartite graph to present a real-world findings in Section 5.4.

5. EVALUATION

We implement a prototype of SPOKE using 3.8K SLOC Python and 2K SLOC Impala SQL on a 8-node Hadoop cluster, with 1K SLOC modification in Linux kernel and Android framework, using a device farm with 4 Samsung Galaxy S6 devices running Android 5.1.1. This experiment environment provides a moderate scale for the evaluation with the help of policy engineers. In practice, SPOKE can easily scale up to a bigger device farm and cloud.

We evaluate SPOKE using the following functional test set. 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 analysis engine to match policy rules with the collected domain knowledge, followed with a real-world finding by the attack surface analysis.

5.1 Data Set and Research Questions

To evaluate the effectiveness of SPOKE in real world, we use a suite of 665 functional tests provided by Samsung Android Team, covering 28 different categories of functionalities in the Android framework. The functionalities include application installation, bluetooth/WiFi/firewall/location configuration, exchange/email/multi-user setting, enterprise device management, etc. The functional tests cover 90% APIs defined in these 28 functionality categories. Tests are executed in both JUnit-based API calling and UI automation. Different functionalities and test cases are developed by different teams. SPOKE is aimed to collect different domain knowledge of both high-level functionality trace and low-level access patterns in a centralized way.

Using this functional test suite, we evaluate SPOKE by asking three research questions:

R1: What domain knowledge is collected from the 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 the 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?

5.2 Knowledge Base Construction

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 a dummy test to get the non-functional access patterns triggered by

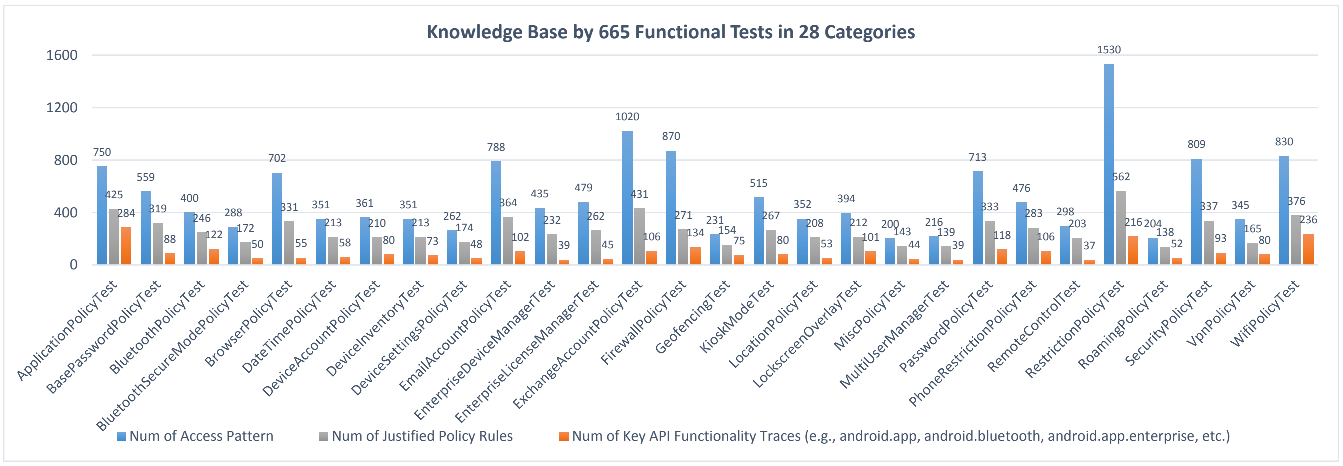


Figure 4: The summary of access patterns, functionality trace and justified policy rules in the knowledge base using 665 functional tests

JUnit `setUp` and `tearDown`. In total, we collect 896 background and dummy test access patterns, which are filtered in the next phase. 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` 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 functional API classes and methods (e.g., `android.app.*`, `android.bluetooth.*`, `android.app.enterprise.*, etc.`), which are selected from corresponding rich-semantic documentations (`Javadoc`).

The total number of functional access patterns collected in the knowledge base is 12,491, with a total of 1,492 API methods extracted as functionality trace. 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 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 (e.g., for enterprise use) such as installing whitelisted packages, configuring limited network settings. Such restricted operations involve permission checkings from device admin subjects across multiple 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 Android framework. For instance, access patterns that system 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 functionality developers that the above two file objects are the core system configuration files.

5.3 Justifying Policy Rules w.r.t. $\mathbb{K}$

With the above knowledge base, we match SEAndroid policy rules with corresponding functionality trace to justify the access patterns defined in these rules.

We first identify in total 1,356 relevant `allow` rules in the policy of the Galaxy S6 device running Android 5.1.1. These rules are identified because the access patterns defined in these rules have subjects or objects that are found in tested functionalities.

Then SPOKE's analysis engine uses the SQL queries and access pattern generalization mentioned in Section 4.2 to attempt to match these rules with the knowledge base. Table 1 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 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, as shown in the following.

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 map-

Table 1: Summary and reasoning of policy rule justification

Matching Result	Num Rules	Rule Characteristics & Reasoning
Matched	187	Fine-grained labels of subjects and objects with privileged classes (e.g., chr_file, netlink)
Partially Matched	198	Rules defined using attribute group (e.g., domain, system_domain, appdomain)
	269	Coarse-grained labels for application subjects (e.g., platform_app, system_app)
	382	Default labels for different file objects (e.g., system_file, system_data_file)
Unmatched	320	Irrelevant subjects accessing functionality-related objects

ping 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.

Firstly, 198 rules are defined using attributes. Attribute is an SEAndroid policy language feature that defines a group of labels [10]. 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 the tested functionalities. 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 a subset of files are related to the tested functionalities. 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 critical subjects/objects. Such rules are subject to attack surface analysis.

It is worth noting that some unjustified and partially unjustified policy rules and access patterns exposed by SPOKE were analyzed by both policy engineers and functionality developers. The analysis result has been integrated in the following updated policies in new Android releases.

With the help of SPOKE, previously unclassified policy rules can be differentiated into different categories based on their justification results. This helps policy engineers analyze the rules with semantic contexts.

5.4 Analyzing attack surface of Policy Rules

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

As a case study, we select 5 critical file objects in one system directory[3]. These system file objects are identified based on enterprise security-related functionality trace of `android.app.enterprise`, as explained in Section 5.2. These system file objects contain device configuration, password and encryption keys.

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 security 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.

In Appendix, Figure 5 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 system 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 [5] (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 append unlink link rename open}` to modify critical system files and eventually control the enterprise device. This risky rule with other ones have been confirmed and removed by policy engineers.

6. DISCUSSION

Native functional tests and other knowledge inputs: Currently, SPOKE mainly focuses on Android functional

[3]Since the analyzed policy is currently used in real-world Android devices, we are requested by the vendor to anonymize some specific file names.

tests for applications and framework. 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 [18] can provide detailed information flow of a series of access patterns. Static analysis such as symbolic execution [33, 45] 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 analysis 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 analysis engine. For example, outlier/anomaly detection [22] 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 [19, 20] 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.

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 [4]). We envision that SEAndroid policy development can also benefit from similar user beta testings.

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 [44] proposed an emulation-based inspection to analyze both Java and native components of Android applications. CopperDroid [39] also used QEMU and focused on system call analysis of Android malware. TaintDroid [18] 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, it is non-trivial and insufficient to port previous techniques, as we focus on a fundamental new problem of collecting domain knowledge for SEAndroid policy, which requires new techniques specific for knowledge extraction.

Although SEAndroid is relatively new, SELinux has been researched for years, such as SELinux policy analysis and verification [12, 21, 26, 36], policy comparison [15], policy visualization [43], policy information flow integrity measurement [24, 25, 40]. 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 [42] 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 analysis results.

8. CONCLUSION

SEAndroid policy development and analysis require domain knowledge. In this paper, we presented SPOKE, a knowledge engine that 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.

Acknowledgements We would like to thank colleagues in Samsung Research America for their valuable input and resource. We also like to thank anonymous reviewers for their support to publish this paper. William Enck's work in this paper was supported in part by NSF grant CNS-1253346 and ARO grant W911NF-16-1-0299. Ninghui Li's work was supported by NSF grant No. 1314688 and ARO grant W911NF-16-1-0127. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of Samsung or the funding agencies.

9. REFERENCES

[1] Android Testing. http://developer.android.com/tools/testing/index.html.
[2] AWS Device Farm of Mobile App Testing. https://aws.amazon.com/device-farm/.
[3] EMMA: a free Java code coverage tool. http://emma.sourceforge.net.
[4] Google Play Store Beta Testing. http://developer.android.com/distribute/googleplay/developer-console.html.
[5] Joshua Drake, Stagefright: Scary Code in the Heart of Android. https://www.blackhat.com/us-15/briefings.
[6] Profiling with Traceview. http://developer.android.com/tools/debugging/debugging-tracing.html.
[7] Security-Enhanced Linux in Android. https://source.android.com/security/selinux.
[8] SELinux Access Vector Rules. http://selinuxproject.org/page/AVCRules.
[9] SELinux Policy Analysis Tools. https://github.com/TresysTechnology/setools.

[10] SELinux Type Statements. http://selinuxproject.org/page/TypeStatements.

[11] Testdroid. http://testdroid.com/.

[12] M. Alam, J.-P. Seifert, Q. Li, and X. Zhang. Usage Control Platformization via Trustworthy SELinux. In ASIACCS '08, pages 245–248. ACM, 2008.

[13] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[14] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*. San Diego, 1998.

[15] H. Chen, N. Li, and Z. Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS '09*, 2009.

[16] W. Choi, G. Necula, and K. Sen. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

[17] R. DeMilli and A. J. Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, 1991.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[19] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2):131–163.

[20] N. Friedman, I. Nachman, and D. Peér. Learning Bayesian Network Structure from Massive Datasets: The Sparse Candidate Algorithm. In UAI'99, pages 206–215. Morgan Kaufmann Publishers Inc., 1999.

[21] B. Hicks, S. Rueda, and L. S. Clair. A logical specification and analysis for SELinux MLS policy. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):1–31, 2010.

[22] V. J. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126.

[23] M. Howard, J. Pincus, and J. M. Wing. *Measuring relative attack surfaces*. Springer, 2005.

[24] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-reduced Integrity Measurement Architecture. In SACMAT '06, pages 19–28, 2006.

[25] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security '03*, 2003.

[26] T. Jaeger, R. Sailer, and X. Zhang. Resolving constraint conflicts. In SACMAT '04, pages 105–114, New York, New York, USA, 2004. ACM Press.

[27] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In ISSTA '13, pages 67–77. ACM, 2013.

[28] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference '01*, number February, pages 29–42, 2001.

[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In ESEC/FSE '13, pages 224–234, 2013.

[30] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[31] D. McCullough. Specifications for multi-level security and a hook-up. In *Security and Privacy, 1987 IEEE Symposium on*, pages 161–161. IEEE, 1987.

[32] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[33] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[34] E. Reshetova, F. Bonazzi, T. Nyman, R. Borgaonkar, and N. Asokan. Characterizing SEAndroid Policies in the Wild. *ArXiv e-prints arXiv:1510.05497*, Oct. 2015.

[35] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), Sept. 1975.

[36] A. Sasturkar, S. D. Stoller, C. R. Ramakrishnan, C. Science, and S. Brook. Policy Analysis for Administrative Role Based Access Control. In CSFW '06, 2006.

[37] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS '13*, 2013.

[38] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[39] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS '15*, 2015.

[40] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies. In ASIACCS '12, pages 75–76, 2012.

[41] W. Visser, S. Corina, and S. Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[42] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *In USENIX Security '15*, pages 351–366, Aug. 2015.

[43] W. Xu, M. Shehab, and G.-J. J. Ahn. Visualization based policy analysis: case study in SELinux. In *Proceedings of the 13th ACM Symposium on Access control models and technologies*, pages 165–174, 2008.

[44] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In USENIX Security '12, pages 29–29, 2012.

[45] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In CCS '13, pages 1043–1054, 2013.

APPENDIX

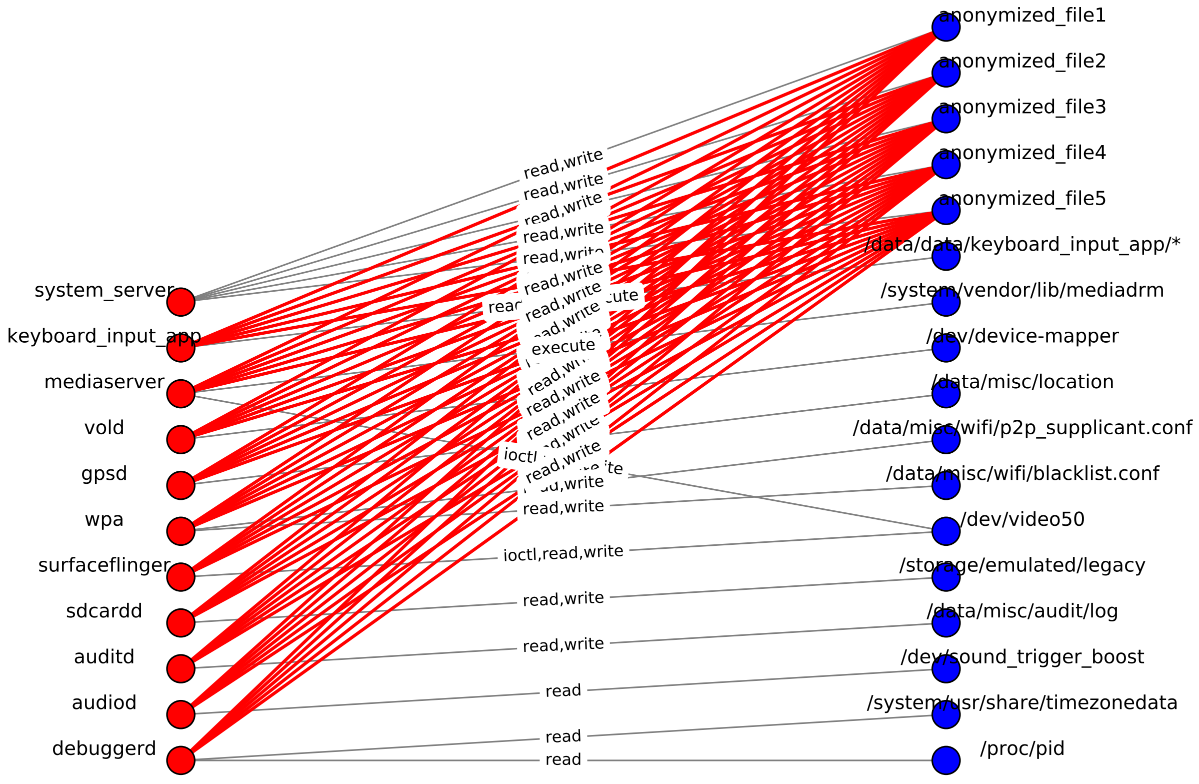


Figure 5: The representative bipartite result of attack surface analysis generated by SPOKE. Subjects are red nodes in left. Objects are blue nodes in right. Justified access patterns are in grey. Over-permissive access patterns in red allow unjustified subjects to access anonymized critical files, which have been confirmed and revoked by policy engineers. With the help of SPOKE, policy rules in new releases are more strict than the old ones.