Toward Monitoring, Assessing, and Confining Mobile Applications in Modern Mobile Platforms

by

Yiming Jing

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2015 by the
Graduate Supervisory Committee:

Gail-Joon Ahn, Chair
Adam Doupé
Dijiang Huang
Yanchao Zhang

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Smartphones are pervasive nowadays. They are supported by mobile platforms that allow users to download and run feature-rich mobile applications (apps). While mobile apps help users conveniently process personal data on mobile devices, they also pose security and privacy threats and put user's data at risk. Even though modern mobile platforms such as Android have integrated security mechanisms to protect users, most mechanisms do not easily adapt to user's security requirements and rapidly evolving threats. They either fail to provide sufficient intelligence for a user to make informed security decisions, or require great sophistication to configure the mechanisms for enforcing security decisions. These limitations lead to a situation where users are disadvantageous against emerging malware on modern mobile platforms. To remedy this situation, I propose automated and systematic approaches to address three security management tasks: monitoring, assessment, and confinement of mobile apps. In particular, monitoring apps helps a user observe and record apps' runtime behaviors as controlled under security mechanisms. Automated assessment distills intelligence from the observed behaviors and the security configurations of security mechanisms. The distilled intelligence further fuels enhanced confinement mechanisms that flexibly and accurately shape apps' behaviors. To demonstrate the feasibility of my approaches, I design and implement a suite of proof-of-concept prototypes that support the three tasks respectively.

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor Dr. Gail-Joon Ahn for his insights, patience, enthusiasm, and knowledge that navigate me through the academic world. He constantly encourages me with his invaluable visions and I could not finish this degree without the support and understanding from him. I also thank my dissertation commmittee members, Dr. Adam Doupe, Dr. Dijiang Huang, and Dr. Yanchao Zhang for their unselfish assistance and guidance during the preparation of my dissertation proposal and this dissertation.

This dissertation would not be possible without the current and past members of the Laboratory of Security Engineering for Future Computing (SEFCOM), in particular, Dr. Hongxin Hu, Dr. Ziming Zhao, Ruoyu Wu, Michael Mabey, Carlos Rubio, Jeong-Jin Seo, Wonkyu Han, Jeremy Whitaker, Marthony Taguinod, Justin Paglierani, Michael Sanchez, Yeganeh Safaeisemnani, and Bernard Ngabonziza. It has been my pleasure to work with them and their help and insights help me through the journey as a doctoral student.

Many thanks to my colleagues and friends outside SEFCOM: Yetian Xia, Huijun Wu, Bing Li, Lingjun Li, Lei Liu, Qiang Zhang, Xiang Zhang, Ruozhou Yu, Xiaowen Gong, Jingchao Sun, Yang Cao, and Mengyuan Zhang. I would not survive the Ph.D. grind without their encouragement and support. Last but not the least, I would like to thank my parents Jing Jing and Min Yang. None of my achievement would have been possible without their moral support and warm encouragements that enabled me to surpass the hard times in the past five years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Modern mobile platforms, such as Apple iOS and Google Android, have evolved significantly to support a computing paradigm that is different from those supported by desktop platforms. Today's mobile devices are always connected to the Internet thanks to the prevalence of 3G/4G cellular networks. Meanwhile, they generate local contexts derived from a collection of sensors such as microphones, cameras, GPS receivers, gyroscopes, and accelerometers. Mobile apps digest the contexts and provide persistent and personalized services to users. Application stores, as an integral part of modern mobile platforms, provide central and trusted app distribution channels through which users can discover and purchase apps. The possibilities enabled by modern mobile platforms lead to an explosive growth of feature-rich apps that stores and processes user's sensitive information.

As mobile devices are rich of sensitive data, they also become an appealing target for adversaries. Android malware has increased by 600% to a total of more than 6M pieces from 2013 to 2014 [16]. These malware steals almost everything stored on mobile devices, including photos, messages, browsing histories, banking accounts, and two-factor authentication tokens. There is even evidence showing that recent Android malware is "mutating and getting smarter" [16]. In 2013, Sophos reported sophisticated malware strains that employ heavy obfuscation, encryption, and polymorphism techniques to resist detection and analysis. The security community also reported malware samples that are able to detect the presence of emulators and hide themselves from emulator-based screening tools deployed at application stores [47, 57].

To protect users against adversaries, modern mobile platforms implement *sandboxes* and *permissions*. A sandbox enforces a fixed set of default rules and isolates an app's data and code in a unique and small protection domain, so that the app's capabilities are confined and minimized to accessing its own files and security-insensitive APIs. However, apps inevitably communicate with other apps and the mobile OS. Therefore, a user can explicitly grant permissions to an app so that the app can access resources outside its own sandbox. The concept of permission is widely adopted in modern mobile platforms. For example, iOS uses the name "entitlements," and Windows Phone uses "requirements."

The problem of these security mechanisms is that they rely on the user to evaluate the security and privacy implications of apps, but are too complex for users to take actions. Before installing an app, a user only sees the app's meta information, such as the description of the app and/or a list of permissions requested by the app. Note that it is the app's developers who specify such meta information; and they may lie or hide their true purpose. Even after installing the app, the user only see the app's interface and limited visual hints from the mobile OS, while the background behaviors of the app are invisible. As a result, it remains a crushing burden for users to assess an app and to tame the app to do only what the user expects it to do. The burden comes from both the sophistication required to understand the internals of mobile OSs and the consistent attention required to address rapidly evolving apps and threats. Moreover, apps do not run by themselves; an app's behaviors are also shaped by its communication from and to other apps, which further stacks up the complexities in taming apps. And failing to address inter-application communication could lead to data leaks and privilege escalation attacks. In summary, the limitations of existing security mechanisms in modern mobile platforms render users disadvantageous with respect to protecting themselves against rogue apps.

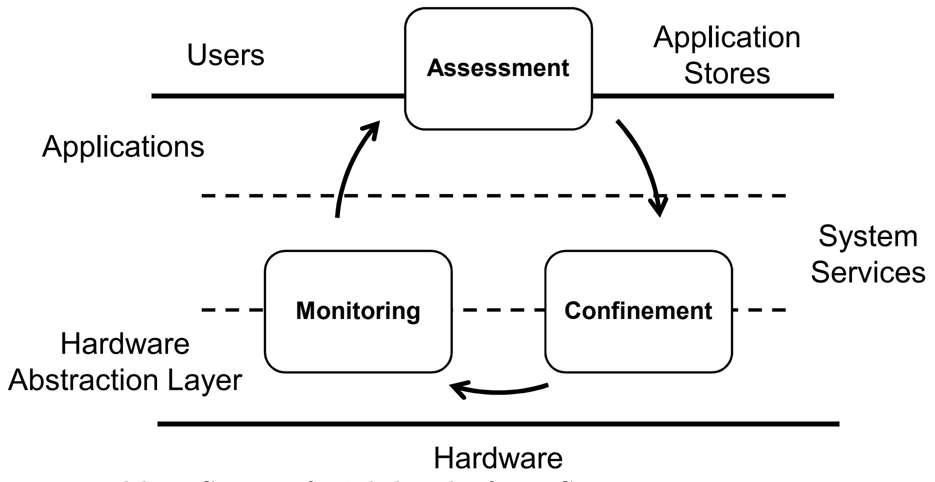


Figure 1.1: Problem Space of Mobile Platform Security

1.1 Thesis Statement

The security situation on modern mobile platforms is dire. While the adversaries are getting smarter, the users expect to regain control of the mobile devices and they demand more power to inspect and tame apps. The discoveries and observations made during our studies of modern mobile platforms inform the following thesis statement:

> *Modern mobile platforms need systematic and automated approaches to monitor, assess, and confine the behaviors of mobile applications.*

In this dissertation, we focus on the Android platform because it is one of the most representative modern mobile platforms and it provides the necessary openness of documentation and platform source code that facilitate our analysis, modifications, and experiments. Figure 1.1 depicts the problem space of Android system security and where *monitoring*, *assessment*, and *confinement* fit in the space. Specifically, we propose to monitoring apps interactions with other apps and the mobile OS to acquire the intelligence about apps' actual purpose. We further propose assessment mechanisms to help users better digest the collected intelligence, make informed security decisions and, verify the effectiveness of the decisions. Finally we propose confinement mechanisms to flexibly enforce the security decisions. The three tasks constitute a security management lifecycle that can be applied continuously and iteratively to address emerging threats and rapidly evolving mobile apps.

Evidence of each task can be found in our subsequent studies. In our first study, we attempt to understand the limitations of the current monitoring and confinement mechanisms and exploit these limitations for attacks. In particular, we systematically generate a set of heuristics that allow malware to evade emulator-based dynamic analysis. Based on the discovered limitations, our second study proposes enhanced monitoring and confinement mechanisms. We propose a suite of three comprehensive reference monitors that mediate app-to-app and app-to-OS communication at three respective layers of Android. Our third study implements an automated risk assessment framework. It evaluates each individual app's security risk with an automatically generated risk assessment baseline derived from a user's security requirements and the user's trusted apps' runtime behaviors. The final study goes beyond individual apps and assesses inter-application communication. We propose a generic intent space model and an automated policy checker to assess intent-based inter-application communication that is controlled by multiple incompatible security mechanisms.

1.2 Previous Publications

This dissertation incorporates materials from my previous conference and journal papers. The concepts and techniques of attacking the loopholes of the existing monitoring and confinement mechanisms in Chapter 3 were discussed in the following conference paper:

- Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu, Morpheus: Automatically generating heuristics to detect android emulators. *In Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

The system for comprehensively monitoring and confining app's runtime behaviors in Chapter 4 was discussed in the following journal paper that is currently in submission.

- Yiming Jing, Gail-Joon Ahn, and Hongxin Hu, TripleMon: A multi-layer security framework for mediating inter-process communication on android. In submission to *Journal of Computers and Security*, Elsevier, 2015.

The ideas of a risk assessment framework for individual apps and a holistic policy checking framework for inter-application communication were discussed in two conference papers, a journal paper, and a conference paper that is currently in submission.

- Yiming Jing, Gail-Joon Ahn, and Hongxin Hu, Model-based conformance testing for android. *In Proceeding of the 7th International Workshop on Security (IWSEC)*, Springer, 2012.

- Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu, RiskMon: Continuous and automated risk assessment of mobile applications. *In Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY)*, ACM, 2014. (Best Paper Award)

- Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu, Towards automated risk assessment and mitigation of mobile applications. *In IEEE Transactions on Dependable and Secure Computing*, IEEE, 2015.

- Yiming Jing, Adam Doupé, Gail-Joon Ahn, Checking Intent-based Communication in Android with Intent Space Analysis. In submission to the 6th ACM Conference on Data and Application Security and Privacy, 2016.

Chapter 2

BACKGROUND

This chapter gives a brief introduction about the Android platform that facilitates understanding of the remaining chapters.

2.1 Android Fundamentals

Android is a mobile platform that consists of a monolithic Linux kernel and a loosely-coupled middleware layer. An app can use functionalities provided by both the middleware and Linux. The basic building blocks of Android apps are *components*. Android defines four types of components to address different requirements and scenarios [1] .

- *Activities* are components that provide graphic user interfaces (GUI). The Android GUI is implemented as a stack of activities starting one after another, where each activity is typically presented as a window on the screen.
- *Services* are components that run in the background for long-running operations. They expose remote procedure call (RPC) interfaces to be called by other apps.
- *Broadcast Receivers* are components that asynchronously receive broadcasts sent from other components.
- *Content Providers* are components that provide public data interfaces to other components. A content provider supports common database operations such as query, insert, update and delete.

[1] `http://developer.android.com/guide/components/index.html`

A component can be exported to other apps. Each exported component of an app is an entry point for intents through which the other apps or the Android system can send intents. Typically, an app exports its components to other apps by statically declaring the exports in the app's manifest. However, an app can also dynamically create and export components in its code. Two system services, PackageManagerService (PMS) and ActivityManagerService (AMS), maintain the information about each installed app's components regardless of how the components are exported—either statically or dynamically.

2.2 Inter-Application Communication in Android

The sandbox in Android mediates apps' accesses on resources and isolates an app's data and code execution from other apps. It disallows an app to *directly* access the data outside its sandbox. Instead, it allows the app to send data requests to other apps and system services through inter-process communication (IPC) channels. In this section, we briefly describe the major types of Android IPC mechanisms. Figure 2.1 shows the examples of these IPC channels in Android. On the top of the figure we show the data owned by apps and system services. The sandboxed apps and system services are listed in the middle. The arrows demonstrate data requests sent by apps via IPC mechanisms. Next, we describe the IPC mechanisms and their corresponding examples in the figure.

2.2.1 Intent-based Inter-Component Communication

Intent play a leading role in connecting the components of apps. An app creates an intent and sets its embedded attributes. The intent is then processed by the Android system and the security extensions, which automatically resolve an intent's recipients based on the following intent attributes:

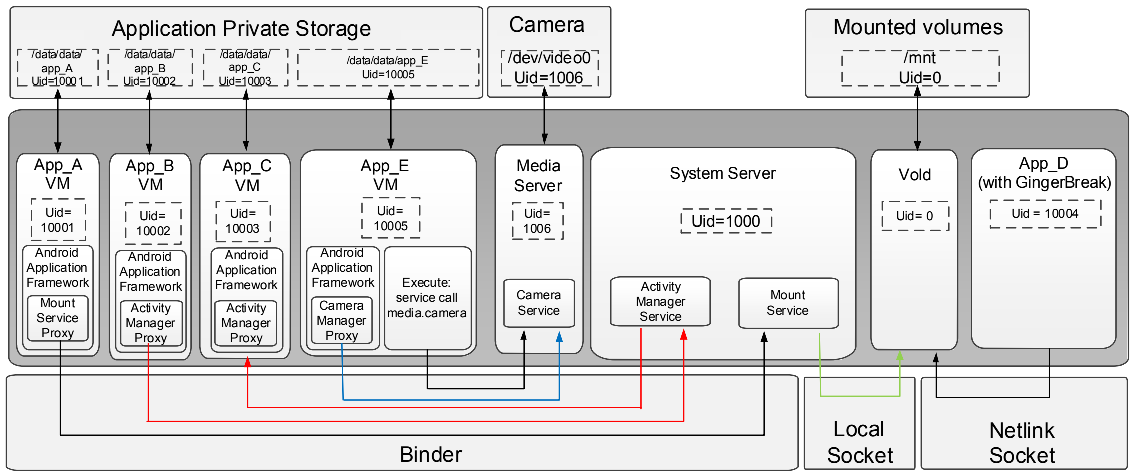


Figure 2.1: Examples of Android Inter-Process Communication

- **Component name:** This attribute explicitly specifies the expected recipient of the intent.
- **Action:** This attribute describes the general action to be taken by a recipient component, such as `PICK`, `VIEW`, `EDIT`, or `SHARE`.
- **Scheme:** This attribute describes the protocol that serves the data, such as `http`, `mailto`, or `tel`.
- **Authority:** This attribute describes the location of the data, such as `www.google.com` or `paypal`.
- **Type:** This attribute describes the MIME type of the data, such as `audio/ogg`, `video/*`, or `*/*`. Note that wildcards are allowed.
- **Category:** This attribute provides additional information about the data. For example, a category `BROWSABLE` implies the data that can be opened in a web browser, such as a link to an image.

Two types of intents exist in Android. *Explicit intents* specify the component name only. Android delivers an explicit intent directly to its specified component regardless of the presence of any other attributes. *Implicit intents* specify the attributes other than component name. Thus, an implicit intent's recipients are implicit and must be resolved at intent-sending time; Android must search the registered components to resolve the recipient components.

Out of all the existing security extensions that control intents, four security extensions are part of the Android Open Source Project (AOSP) and ship with almost every recent Android device. *Intent filters* assigned on an app's component specify the implicit intents that the component can *receive*. Each filter corresponds to only one component, while a component can have multiple filters. An intent filter describes its accepted intents with the same attributes as those of implicit intents. *Permissions* further constrain an app from *receiving* explicit and implicit intents. Specifically, a permission assigned on a component requires the component to only receive intents from the apps that hold the same permission. *Protected broadcasts* are a set of special implicit intents reserved by the Android framework and system apps; they cannot be *sent* from any third-party apps. In addition, *IntentFirewall* enables policy-driven access control over both types of intents. It denies specific apps from *sending* certain intents as specified in its policy, which can be defined by users or enterprise IT. Overall, an intent is processed by one security extension after another before it reaches any recipient component. As shown in Figure 2.1, App_B and App_C utilize ICC to communicate with each other. The arrows show that App_B sends an intent to App_C via AMS.

2.2.2 Binder IPC

Although the sandbox prevents an app from directly accessing anything outside its sandbox, it allows the app to interact with system services via Android APIs which are implemented based on an IPC framework called *Binder*. Binder includes proxies implemented as parts of the app framework, stubs implemented as parts of system services, and a kernel module to manage Binder IPC transactions. The kernel module identifies IPC transactions with the user IDs and process IDs of sender and recipient processes as well as a command code that specifies the action to be performed by the receipt process.

A typical Binder IPC channel is established with 6 steps: (1) an app invokes the proxy that encapsulates the destination system service's Binder IPC handle, a command code, and optional data in a parcel; (2) Binder resolves the destination using the IPC handle; (3) Binder delivers the parcel and invokes the stub in the destination system service; (5) the stub unpacks the parcel, takes the action as instructed by the command code, and sends results back to the proxy via Binder; and (6) the app receives the results. As depicted in Figure 2.1, App_E can use the camera via the CameraManagerProxy of the app framework. As of this writing, Binder IPC is solely controlled by a kernel-based Mandatory Access Control (MAC) implementation called SEAndroid [109].

2.2.3 Linux IPC

Traditional Linux IPC is still available in Android. Examples include local sockets, Unix-domain sockets, and Netlink sockets. In a broader sense, Linux IPC also includes communication via signals and files. System services may use Linux IPC to communicate with other services and the kernel. Note that apps may also use Linux IPC, which allows them to bypass the Android middleware. Recent attacks [124, 125] have demonstrated the feasibility of exploiting such IPC channels. Figure 2.1 shows that MountService communicates with Volume Daemon (vold) through a local socket. In addition, vold exposes a Netlink socket to receive events (e.g., unplugging storage media) from the kernel. Similar to Binder IPC, Linux IPC mechanisms are mediated by SEAndroid [109].

2.3 Prominent Threats

Android includes a monolithic Linux kernel and a loosely-coupled middleware layer. Due to the mixed nature of this platform, Android not only inherits security issues rooted in the Linux kernel, but also faces unique security problems introduced by the middleware.

2.3.1 Attacks Bypassing the Sandbox

Before Android 4.2, the Android sandbox is based the discretionary access control (DAC) provided in Linux kernels by default. Files that are set globally accessible render the DAC-based sandbox useless. For example, a vulnerability in Skype exposes the user's profile and messages to every installed app [7]. Similarly, a file that stores the list of installed apps is mistakenly set globally readable in early Android versions [68].

Linux kernel vulnerabilities also make it possible to bypass the sandbox [1, 2, 6, 8, 9, 12]. These exploits (also known as "jailbreaks") allow an app to escalate their privilege to root privileges. Recent reports have shown cases of root-capable apps that download additional malware [14] and replace system binaries [13]. Indeed, the occurrence of such exploits is quite high [126], and almost every popular Android device had a publicly available exploit for at least 74% of the device's lifetime [59].

2.3.2 Attacks Bypassing Permissions

As apps may unintentionally expose private components, ICC can be abused for unauthorized intent receipt and intent spoofing [42]. For example, a malicious app can sniff, modify, and replace messages sent by a benign app. Moreover, Lineberry *et al.* show that a zero-permission app may upload without the `INTERNET` permission by sending an intent to the Browser app [5]. Several similar zero-permission attacks have been discovered as well, such as making phone calls [56] and setting an alarm [60].

A coarse-grained permission covers multiple capabilities for accessing resources with different sensitivity levels. For example, `READ_PHONE_STATE` allows an app to access both the phone's state (e.g., if there is an incoming call) and unique device identifiers [51]. The former is trivial but the latter introduces potential privacy risks. For apps that utilize the phone's state only, holding unnecessary capabilities such as reading the identifiers violates the security principle of least privilege. In addition, permissions are indivisible and irrevocable. A user has no way to control each individual capability covered by a single permission.

Finally, some sensitive APIs are not well-protected. For example, the `RingerMode` setting of AudioService is not protected by any permission, allowing any app to silence the phone without the user's consent. Felt *et al.* point out that only 6.45% of all API methods in Android are protected by at least one permission [58].

Chapter 3

MONITORING AND CONFINEMENT: ATTACKS

This work analyzes the limitations of existing security mechanisms in Android and attempts to exploit the limitations. We found that apps are over-privileged despite being confined. The excessive privileges allow an app to sense the presence of Android emulators and thus make it possible to evade emulator-based dynamic analysis. In this work, we propose a systematical approach to exploit these privileges and to discover heuristics to detect Android emulators. Our approach and preliminary results are published in [80].

3.1 Problem Statement

The rise of mobile computing should partially give credit to application stores such as Apple AppStore and Google Play. With these services, users enjoy a centralized and trusted source for browsing and purchasing apps. Unfortunately, such advantages also make application stores an appealing place for distributing malicious mobile apps. To infect more unsuspecting users, adversaries would attempt to publish their malware in application stores without being detected. To effectively mitigate such attempts, the application stores have deployed emulator-based dynamic analysis, which vets runtime behaviors of apps on a large scale.

However, a flaw of emulator-based dynamic analysis lies in the discrepancies between emulators and real devices. Such discrepancies, if observable by apps, may lead to detection heuristics (*a.k.a.*, “red pills”) that indicate the fabricated reality of Android emulators. Taking advantage of these heuristics, Android malware can build split personalities and circumvent dynamic analysis. For example, a malicious app could stay dormant or exhibit legitimate behaviors. Furthermore, this app can use dynamic external code loading to evade both static and dynamic analysis, because it only downloads the malicious payload when it is in a real device. Alternatively, it can perform reconnaissance within the emulator and phone home to facilitate generation of up-to-date detection heuristics for future attacks. Indeed, the security community has already discovered Android malware samples that use such heuristics to evade dynamic analysis.

Due to the peculiarities of Android, we argue that Android malware would be reluctant to reuse previous PC emulator detection heuristics. First, Android malware faces a unified runtime environment whose underlying implementation details (*e.g.*, hardware differences) are concealed by the Android middleware and APIs. At the same time, Android malware has been deprived of many capabilities that allow accessing low-level system artifacts by the Android application sandbox. In addition, Android malware would prefer detection heuristics implemented with Java code rather than native code. As native code is used by only a small fraction of benign Android apps but most malicious root exploits [128], native code would draw attention of analysis tools, breaking a detection heuristic's basic purpose of evading analysis.

The detection heuristics found in newly discovered malware samples seem to be in line with our argument. They allow an app to detect emulators without bypassing the application sandbox and without the assistance of native code. For example, a popular detection heuristic involves an Android API `getDeviceId` that returns the IMEI of an Android device. This heuristic calls `getDeviceId` and tests whether "`000000000000000`" is a substring of the returned value of `getDeviceId`. It can be implemented with only two lines of Java code and thus leaves relatively small footprints. Despite that researchers have discovered similar detection heuristics and evaluated their effectiveness against Android SDK emulators, the magnitude and accuracy of such heuristics remain unknown, which results in an impediment to the development of comprehensive countermeasures.

Regrettably, all known detection heuristics that target Android emulators are discovered piece by piece in an ad-hoc fashion. For example, some heuristics are discovered through dissecting malware samples [47, 57]. Such a reactive approach cannot predict unknown heuristics. Other known heuristics are derived from manual analysis on specific components of Android emulators [88, 112]. Even though this approach is proactive, manual analysis inevitably cannot address the multitude of components in Android emulators.

3.2 A Proactive and Automated Approach to Generate Heuristics

In our threat model, we assume emulators that run Android with default configurations. We also assume the presence of passive anti-detection techniques, which do not proactively instrument the application to suppress the execution of detection heuristics. This is also the common setup of the existing deployed emulator-based dynamic analysis systems.

In addition, we assume a malicious Android application that does not bypass the application sandbox or carry any native code. Meanwhile, we allow this application to request any Android permission. granted. In other words, this application's capabilities are no more than those of the benign applications in application stores. Afterwards, it applies detection heuristics that check the presence or contents of certain artifacts. Based on the result, it determines where it is running.

As the app is properly confined, it cannot exploit the artifacts that it cannot observe from within the application sandbox. Here we define *observable artifacts* as artifacts (*e.g.*, files, APIs) whose presence can be probed or whose contents can be read by any Android application in its sandbox. For example, suppose a file is not

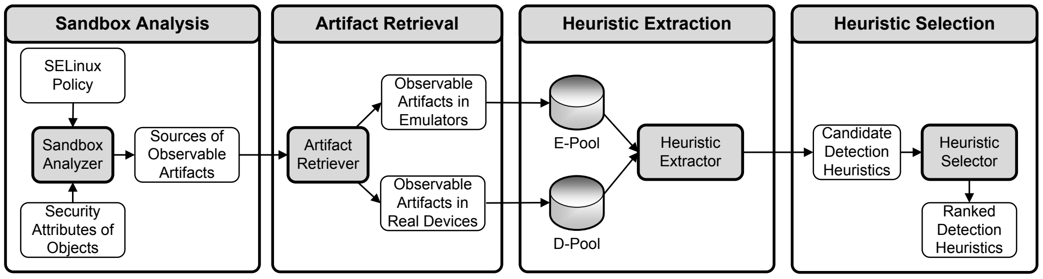

Sandbox Analysis
SELinux Policy
Sandbox Analyzer
Security Attributes of Objects
Sources of Observable Artifacts
Artifact Retrieval
Artifact Retriever
Observable Artifacts in Emulators
Observable Artifacts in Real Devices
Heuristic Extraction
E-Pool
D-Pool
Heuristic Extractor
Heuristic Selection
Candidate Detection Heuristics
Heuristic Selector
Ranked Detection Heuristics

Figure 3.1: Proposed Heuristic Generation Framework

readable but its parent directory is listable, this file is still an observable artifact. Leveraging this observation, the key of this work is to retrieve and analyze observable artifacts and verify whether they can be used to indicate emulators.

As depicted in Figure 3.1, our framework would consist of four components. The sandbox analyzer analyzes the default configurations of the Android application sandbox to identify sources of observable artifacts. For respective sources, the artifact retriever enumerates observable artifacts and retrieves their contents. The retrieved observable artifacts are uploaded to two pools for both emulators and real devices, respectively. The heuristic extractor then analyzes the pools by finding the artifacts or substrings of their contents that appear in most emulators but a small fraction of real devices, and vice versa. These artifacts and substrings constitute candidate detection heuristics. Finally, the heuristic selector ranks the candidates.

3.2.1 Sandbox Analyzer

Applications' accesses on artifacts are regulated by the Android application sandbox, which is based on discretionary and mandatory access control (DAC and MAC). The Linux kernel provides DAC, which grants accesses by checking permissions of objects. Security-Enhanced Linux (SELinux) adds MAC over DAC starting from Android 4.3. SELinux grants accesses by checking domains of subjects (*e.g.*, `untrusted_app`), types of objects (*e.g.*, `wallpaper_files`), and SELinux permissions (*e.g.*, `open`, `read`, `ioctl`, `recv_msg`) [1] .

To identify sources of observable artifacts, we need to access all the objects in the Android OS. However, it is infeasible to do so in off-the-shelf Android devices due to the application sandbox and lack of root privileges. Instead, we propose the sandbox analyzer that analyzes the reference SELinux policy in Android and the security attributes (*e.g.*, owners, permissions, xattr) of objects in rooted reference Android devices (*e.g.*, Nexus devices). Specifically, we attempt to identify the objects whose security attributes expose themselves to third-party applications. Given that third-party applications are automatically assigned into the `untrusted_app` domain during installation, we simulate DAC and MAC checks to identify the following objects: (1) objects that are world-readable or under world-listable directories; and (2) objects that are accessible by `untrusted_app` using read-like SELinux permissions (*e.g.*, `read`, `recv_msg`, `ioctl`). From such objects, we then distill the sources of observable artifacts based on their owners and SELinux types, along with proper methods to retrieve them. For example, `/dev/binder` has the SELinux type `binder_device`. Its SELinux type indicates that it belongs to the Binder IPC subsystem that allows an application to access remote artifacts in system services. Such artifacts would require Binder-specific methods to retrieve. As variations in the hierarchy of objects across different Android versions are insignificant, the sources of observable artifacts derived from the reference inputs should be applicable in emulators and real devices.

We stress that the sandbox analyzer is much more conservative compared with the current Android application sandbox. SELinux in Android 4.3 is configured to permit every access. Even in Android 4.4, SELinux only protects several critical system

[1] We ignore users, roles, and security levels for brevity because they are rarely used in the context of Android.

daemons and does not confine third-party applications (*i.e.*, `untrusted_apps`). With that said, the true amount of observable artifacts in current Android devices could be much larger. However, considering the possibility that SELinux may extend its coverage in the upcoming versions of Android, we choose to be conservative for the future effectiveness of our detection heuristics.

3.2.2 Artifact Retriever

The artifact retriever is essentially a probe application. It requests all the available Android permissions to maximize its capabilities within the confinement of the application sandbox. Based on the identified sources of observable artifacts, we implement the corresponding methods in the artifact retriever to automatically retrieve the observable artifacts as well as their contents.

To address the various sources of observable artifacts, we propose three foundation modules in the artifact retriever: a directory walker, a Java function caller, and a Binder IPC caller. They are tailored to the peculiarity of Android and can be easily adapted and combined. Specifically, the directory walker traverses file-like artifacts. The Java function caller enumerates and manipulates both public and hidden Android APIs. The Binder IPC caller directly triggers remote system services (*e.g.*, `TelephonyManagerService`) with dynamically constructed Binder IPC messages.

We launch the artifact retriever into both Android emulators and real devices. It probes the surrounding observable artifacts with its carried modules. It technically captures the first 1KB of each artifact's contents if readable. Upon explicit errors (*e.g.*, denied access), it records the error messages as the retrieved contents. Upon implicit errors (*e.g.*, blocking read), it uses a timeout to ensure that it does not hang there infinitely. We note that, the artifact retriever must upload artifacts to the correct pool according to where the artifacts are observed. For example, artifacts collected from emulators should never go into D-Pool. This is critical for the heuristic extractor to work effectively, because arbitrary noises could make the problem of heuristic generation NP-hard [86].

3.2.3 Heuristic Extractor

The inputs of the heuristic extractor are two pools, namely E-Pool and D-Pool, which contain instances of observed emulators and real devices, respectively. Each instance is a collection of key-value pairs that map retrieved artifacts to their contents. A key (artifact) occurs in an instance once at most, although it can occur in multiple instances. And a value (content) can be null if the artifact retriever fails to read the contents. Next, we describe two categories of detection heuristics that generate decisions based on the artifacts and their contents, respectively.

We start from a category of heuristics that make decisions based on the presence of artifacts. First, we attempt to discover the artifacts that are exclusively used by emulators, such as emulator-specific hardware, software, and configurations. As we use their presence to imply emulators, we refer to them as Type E artifacts. Furthermore, we also look for the artifacts that appear in most real devices, which become our Type D artifacts.

We propose two metrics, $COV_E(a)$ and $COV_D(a)$ to denote the fractions of instances in E-Pool or D-Pool that contain artifact a, *i.e.*, $COV_E(a) = \frac{|E_a|}{|E|}$, and $COV_D(a) = \frac{|D_a|}{|D|}$. Intuitively, our heuristics should at least perform better than a 50/50 guess. Thus, we choose Type E and Type D artifacts from all the artifacts in both pools according to their values of $COV_E(a)$ and $COV_D(a)$ as follows:

- **Type E artifacts:** $COV_E > 50\%$, $COV_D < 50\%$
- **Type D artifacts:** $COV_E < 50\%$, $COV_D > 50\%$

However, there are plenty of artifacts that are prevalent in both emulators and real devices. For example, Android APIs would have both COV_E and COV_D larger than 50%. Inspired by Hamsa [86], we propose a category of detection heuristics whose decisions are based on tokens, where token is a contiguous byte subsequence in the contents of an artifact. Similar to what we introduce for artifact-based heuristics, we attempt to find Type E and Type D tokens.

Specifically, for an artifact a and its retrieved contents in E-Pool, we extract a set of tokens by computing common substrings among the contents. We then extract another set of tokens for D-Pool. Combining these two sets of tokens as a token set T, we compute $COV_E(a,t)$ and $COV_D(a,t)$, which are the fractions of instances in E-Pool and D-Pool whose contents of artifact a contain token t, *i.e.*, $COV_E(a,t) = \frac{|E_{a,t}|}{|E|}$ and $COV_D(t) = \frac{|D_{a,t}|}{|D|}$. Based on the values of $COV_E(a,t)$ and $COV_D(a,t)$, we select two type of tokens as our content-based heuristics as follows:

- **Type E tokens:** $COV_E > 50\%$, $COV_D < 50\%$
- **Type D tokens:** $COV_E < 50\%$, $COV_D > 50\%$

There are various algorithms that effectively compute common substrings. We opt for a suffix array in our heuristic extractor. Constructing a suffix array runs in $O(nlogn)$ time in worst case scenario and consumes $5n$ bytes of memory, where n is the total size of the contents of an artifact in a pool. Extracting tokens from the constructed suffix array can be implemented using a binary search. Furthermore, as we prefer longer tokens in the context of generating detection heuristics, we add one more step to prune tokens that are substrings of the other tokens as long as they share the same COV_E and COV_D.

The output of the heuristic extractor is a set of Type E and Type D heuristics. Each heuristic is represented as a 3-tuple $(artifact, token, type)$. $token$ can be null for artifact-based heuristics. *type* implies the decision to be made once the observed artifact/token matches the artifact/token specified in the heuristic. The matched Type E heuristics indicate emulators and the unmatched ones indicate real devices. Conversely, Type D heuristics imply the opposite decision.

3.2.4 Heuristic Selector

We propose the heuristic selector to rank the candidate detection heuristics generated by the heuristic extractor. In general, we reduce the problem of ranking the candidates to the problem of feature selection in supervised learning. E-Pool and D-Pool comprise a training set consisted of instances that are correctly labeled with "emulator" or "real device." Furthermore, we have extracted a set of detection heuristics that can be considered as binary features. Now we need to select the relevant and non-redundant detection heuristics that would best classify future observations.

We propose to use a random forest [74], which is an ensemble learning method that leverages a multitude of decision trees for classification. Each individual decision tree covers a random subset of the features and is trained with a random subset of training samples. Afterwards, the random forest fits the training set by letting each decision tree predict its unseen samples and evaluate the errors. During this process, an *importance score* for each feature is measured based on how significant the error rate would change if the feature is removed from the decision trees.

We use this importance score as a metric to rank the candidate heuristics. On one hand, relevant heuristics that contribute much to classification naturally get higher importance scores. On the other hand, redundant heuristics that exploit the same artifact/token as other heuristics are assigned zero or lower importance scores. As such, the final output of the heuristic selector is a set of relevant and non-redundant detection heuristics as sorted by their importance scores derived from the random forest.

As the number of detection heuristics is much larger than the number of instances in the pools, the random forest may suffer from over-fitting, which overestimates the importance level of some heuristics. To suppress over-fitting, we choose to increase the number of decision trees in the random forest. As more trees are added, its tendency to over-fit generally decreases as no single feature can affect every decision tree.

3.3 Finding Detection Heuristics

We ran our experiments with Morpheus against QEMU-based Android SDK emulators [17], VirtualBox-based Genymotion emulators [19], and real devices. In this section, we elaborate our experiments that lead to the findings of 10,632 detection heuristics. We then characterize the heuristics according to the underlying discrepancies that they exploit.

3.3.1 Experimental Setup and Findings

To understand the observable artifacts in the reference Android devices, we adopted an instance of the SDK emulator and a Galaxy Nexus phone that both run Android 4.4. We traversed their mounted file systems to obtain the security attributes of objects. We then acquired a copy of the default SELinux policy from the Android Open Source Project (AOSP). Using these as inputs, the sandbox analyzer identified 33 sources of observable artifacts. However, retrieving all of them requires plenty of domain-specific knowledge for tasks such as enumerating artifacts and constructing valid inputs. In this work, we only retrieved 3 sources that could possibly lead to discrepancies and cover a sufficient number of observable artifacts.

Procfs and Sysfs: Procfs and sysfs are both pseudo file systems that expose kernel objects to userspace programs. Specifically, procfs presents system information, such as loaded kernel modules, mounted filesystems, and network stacks. Sysfs exports hardware information such as connected block devices, buses, and power states. Our implementation of the artifact retriever traversed these two file systems mounted at `/proc` and `/sys`. In particular, we slightly adapted the directory walker to handle looped symbolic links that are prevalent in procfs and sysfs.

Android APIs: A large number public and hidden APIs are exposed by Android system services. For example, `TelephonyManagerService` exposes APIs that return unique device identifiers to applications. Actually, the APIs are implemented with the underlying Binder IPC framework, which handles the IPC between applications and system services through a Binder device node located at `/dev/binder`.

To probe APIs behind Binder, we implemented two approaches in the artifact retriever. We used the reflection-based Java function caller to enumerate and call APIs. We also adapted the Binder IPC caller to construct and send IPC messages to the remote system services. The returned Java objects and Binder IPC messages were converted into byte sequences as the retrieved artifacts' contents. For Java objects that are not of Java primitive types, we leveraged their `toString` method to acquire more information about them. In this chapter, we are particularly interested in Android APIs that do not have any input parameters. According to [102], such APIs are more likely to be "sources that return non-constant values into application code." As a result, we covered approximately 15% of the 1,326 APIs exposed by Android system services.

Android System Properties: Similar to the Windows registry, Android includes a subsystem that centrally stores system configurations and status. This subsystem, usually dubbed as "property system," has been extensively used by Android system services. For example, a system property `ro.kernel.qemu` is read by the Android debugging bridge daemon (adbd) to determine the presence of emulators. System properties also cover meta information about the hardware, such as device models and manufacturers. Despite that SELinux in Android protects system properties, we inspected the implementation of the property system and found that the security check is only in the function `property_set()`, meaning that SELinux does not prevent reading system properties at all. Moreover, applications are allowed to read `/dev/__properties__`, which is the interface to system properties. Therefore, system properties are observable by every installed application. To retrieve system properties, we adapted the artifact retriever to call a binary executable located at `/system/bin/getprop`. It enumerates system properties so that the Java function caller can read the contents of each property. We note that this executable is only for the artifact retriever. It is not required by the detection heuristics that read system properties.

Afterward, we ran the adapted artifact retriever against 16 instances of QEMU-based SDK emulators, 11 instances of VirtualBox-based Genymotion emulators, and 25 real devices. The SDK emulators covered three CPU architectures, namely ARM, x86, and MIPS. The Genymotion emulators covered x86, which is the only architecture they support. Both emulator types covered Android versions from 2.3 to 4.4. The real devices covered four manufacturers (Samsung, HTC, Motorola, and LGE), three ARM SoC families (Qualcomm Snapdragon, Texas Instruments OMAP, and Nvidia Tegra), and Android versions from 2.1 to 4.4. In particular, the real devices were borrowed from the participants we recruited through university mailing lists under the study protocol reviewed by our institution's IRB. Anecdotally, it took approximately 5-20 minutes for the artifact retriever to retrieve and upload the observable artifacts on each device.

Table 3.1: Discovered Detection Heuristics

Pools	Detection Heuristics			
	File	API	Property	Total
D-Pool + E-Pool	2,121	81	82	2,284
D-Pool + E_Q-Pool	2,961	163	132	3,256
D-Pool + E_V-Pool	4,782	150	160	5,092
Total	9,864	394	374	10,632

The retrieved artifacts contributed to four pools for QEMU-based emulators (E_Q-Pool), VirtualBox-based emulators (E_V-Pool), all the emulators (E-Pool), and real devices (D-Pool). We then fed these pools to the heuristic extractor and the heuristic selector. The heuristic selector ranked the candidate heuristics with 10,000 decision trees and pruned the heuristics with zero importance scores. Table 3.1 shows a breakdown of the discovered 10,632 detection heuristics. In the remainder of this chapter, we will respectively refer to these three categories of heuristics as *file* heuristics, *API* heuristics, and *property* heuristics, in the interest of brevity.

3.3.2 Characterizing Detection Heuristics

Next, we characterize the discovered heuristics based on the discrepancies they exploit. We first discuss the common detection heuristics that exploit the discrepancies shared by both QEMU-based and VirtualBox-based emulators. We then discuss the heuristics that leverage the QEMU-specific or VirtualBox-specific discrepancies, respectively. Our discussion does not aim to be exhaustive, instead we attempt to convey the scope of discrepancies in Android emulators. Considering that an attacker can possibly use this section as hints to craft detection heuristics, we suggest provisional but deployable countermeasures in Section 3.5.

Common Detection Heuristics

Network. These detection heuristics exploit the discrepancies in network interfaces, Netfilter modules, and kernel modules. For example, we found that all the emulators exclusively use `eth0`, whereas the real devices use `wlan0` and `rmnet`. The emulators also miss several IPv6-specific interfaces. In addition, the network interfaces in the emulators are not tetherable, because the emulators are missing the Remote Network Driver Interface (RNDIS) drivers that enable tethering. Netfilter is another source of discrepancies. The real devices include Netfilter modules for several network protocols that are rarely used in the context of mobile devices. Finally, Android introduces a kernel module to track data usage of installed applications. This module does not exist in the emulators.

Power management. This type of heuristics focuses on the power management subsystem. For example, the emulators lack the voltage and current regulators. The emulated CPU does not support frequency scaling. Another heuristic lies in the prevalence of multi-core CPUs in real devices. All the emulators only have a single core, whereas 75% of the real devices have at least two cores.

Audio. A handy feature of Android is headset detection, which allows the audio output to automatically switch between speakers and headsets. This feature is supported by GPIO/I2C buses. Notably, the emulators do not emulate these buses, while 95.6% of the real devices in our experiments have them. Furthermore, the differences in the implementation of audio subsystems between the emulators and real devices result in disparate audio drivers.

USB. Recently, USB On-The-Go (OTG) has been widely adopted in popular Android phones and tablets. It allows mobile devices to act as hosts and control USB peripherals. Intuitively, the mobile devices have to pre-install corresponding drivers of

USB peripherals. As a result, we found that the real devices in our experiments carry drivers for Apple Magic Mouse, joysticks, and external displays. On the contrary, the emulators do not have such drivers and do not support USB OTG.

Radio. The software-emulated radio can lead to detection heuristics as well. For instance, the name of the baseband in all the emulators is "`unknown`." Moreover, the emulators use a default reference implementation of the radio interface layer (RIL), while the real devices typically use customized ones with different names. Similarly, the phone numbers, voicemail numbers, device serial numbers of the emulators are also constants and can be fingerprinted.

Software components and configurations. Despite that most of the discovered detection heuristics are related to hardware, we also identified several heuristics that exploit certain software components and their configurations. For example, the emulators use unique input methods and search interfaces. Regarding configurations, a prominent example is the key that signs the Android OS. The emulators use test keys while the real devices use release keys.

QEMU Detection Heuristics

QEMU. We found various observable artifacts that are part of QEMU. For example, we found a device node that accelerates the virtual graphics. In addition, there are several system properties set by QEMU and read by Android system services. An example is a property that stores the pixel density of virtual screens.

Goldfish virtual hardware. Most existing QEMU-based Android emulators are built upon a virtual hardware platform called "Goldfish." This platform introduces a set of virtual hardware for QEMU to run Android as its guest operating system. For instance, this set of virtual hardware includes a framebuffer, an audio device, and a battery. They are a must for QEMU-based emulators but never appear in real Android devices.

Bluetooth, NFC, and vibrator. The current QEMU-based emulators do not support these hardware. Their corresponding Android APIs return null if called from within the emulators. In particular, the driver of the vibrator is based on a Linux driver model called `timed_output`, which is also missing from the emulators.

VirtualBox Detection Heuristics

VirtualBox. Similar to QEMU-based emulators, we also found plenty of VirtualBox-specific artifacts. For example, we found 4 kernel modules that belong to VirtualBox Guest Additions. As stated in VirtualBox's documentation, these modules "optimize the guest operating system for better performance and usability." However, their presence also indicates VirtualBox.

PC hardware. As we have discussed, QEMU-based emulators lack support for some popular hardware, such as Bluetooth and NFC. On the contrary, VirtualBox-based emulators support many types of hardware that Android does not need. We found hundreds of artifacts that indicate PC hardware and obviously should not appear in mobile operating systems. For example, we found artifacts related to ACPI, CPU fans, thermal sensors, CD-ROM drives, AC97 audio codecs, and PCI Express.

3.4 Measuring Detection Heuristics

As we have demonstrated the magnitude of the detection heuristic for Android emulators, we further measure their accuracies. To this end, we assembled a group of the top-ranked detection heuristics which are ranked by the heuristic selector. We then tested them against emulator-based malware analysis tools and real devices.

In this section, we describe our experiments along with an empirical study on the average accuracies.

3.4.1 Experimental Setup

As the generated common detection heuristics were already ranked by the heuristic selector, we selected the top 10 heuristics out of the **F**ile, **A**PI, and **P**roperty detection heuristics, respectively. Table 3.2 lists the artifacts, tokens, and types of the 30 selected detection heuristics.

We created a synthetic application to simulate the Android malware as we described in the threat model. Specifically, this application integrated the 30 heuristics with a heuristic matching engine based on Java's substring searching methods. It generated its decision using a majority vote among the 30 heuristics. In other words, an Android device is recognized as an emulator if more than half of the detection heuristics indicate so. Furthermore, it only needed four permissions: `READ_PHONE_STATE`, `ACCESS_NETWORK_STATE`, `ACCESS_WIFI_STATE`, and `INTERNET`, which are also frequently requested by benign applications in Google Play [127].

We ran this application in 9 emulator-based malware analysis tools and 128 distinct real devices. As shown in Table 3.3, the malware analysis tools covered two versions of an offline tool called DroidBox and 7 online services. Among the online services, 4 are derived from previous research work and 3 are security products. The 128 real devices were from AppThwack, TestObject, and Baidu MTC, all of which are online services that automatically test applications in real phones and tablets. Note that we did not run our artifact retriever on them due to their limited device minutes and bandwidth quota.

Table 3.2: Top 10 File, API, and Property Heuristics

	Artifact	**Token**	**Type**
F1	/proc/misc	"network_throughput"	E
F2	/proc/ioports	"0ff\0:"	E
F3	/proc/uid_stat		D
F4	/sys/devices/virtual/misc/cpu_dma_latency/uevent	"MINOR=5"	E
F5	/sys/devices/virtual/ppp		D
F6	/sys/devices/virtual/switch		D
F7	/sys/module/alarm/parameters		D
F8	/sys/devices/system/cpu/ cpu0/cpufreq		D
F9	/sys/devices/virtual/misc/android_adb		D
F10	/proc/sys/net/ipv4/tcp_syncookies		E
A1	isTetheringSupported()	"false"	E
A2	getAuthenticatorTypes()	"AuthenticatorDescription {type=com.g}"	D
A3	getSystemSharedLibraryNames()	"com.g"	D
A4	getGlobalSearchActivity()	".android.quicksearchbox/com.android.quicksearchbox"	E
A5	getWebSearchActivity()	".android.quicksearchbox/com.android.quicksearchbox"	E
	D		
A7	getTetherableUsbRegexs()	"rndis"	D
A8	getEnabledInputMethodList()	".android.inputmethod.latin/."	E
A9	getDeviceId() via Binder	"\0\0\03"	D
A10	getTetherableIfaces()	"wlan0"	D
P1	qemu.hw.mainkeys		E
P2	ro.build.description	"release-keys"	D
P3	ro.build.fingerprint	":user/release-keys"	D
P4	net.eth0.dns1		E
P5	rild.libpath	"/system/lib/libreference-ril.so"	E
P6	ro.radio.use-ppp		E
P7	gsm.version.baseband		D
P8	ro.build.tags	"release-key"	D
P9	ro.build.display.id	"test-"	E
P10	init.svc.console		E

3.4.2 Results and Empirical Analysis

We deem emulators as *positive* and real devices as *negative.* Given the measured true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN), we attempt to evaluate the detection heuristics with three metrics, namely *sensitivity*, *specificity*, and *accuracy*. For example, a Type E detection heuristic is

Table 3.3: Evaluated Emulators and Real Devices

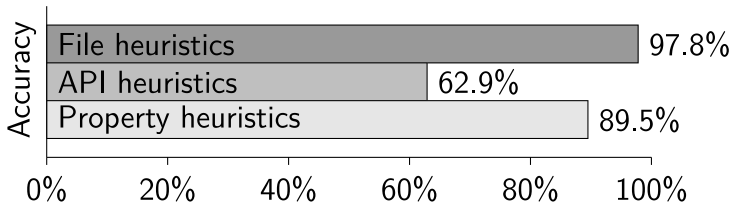
Emulators (9)	DroidBox [10] 2.3 and 4.1, Andrubis [11], CopperDroid [105], SandDroid [20], TraceDroid [21], Qihu 360, NVISO ApkScan, ForeSafe
Real Devices (128)	Samsung, HTC, LGE, Huawei, Motorola, Sony Ericsson, Lenovo, ZTE Hisense, Asus, Acer, OPPO, BBK, Meizu, Gionee, DOOV, YuLong, Haier, AMOI

sensitive if it matches all the emulator instances. And, it is specific if it does not match any non-emulator instances, *i.e.*, real devices. Simply put, we compute the values of the three metrics as follows:

- $Sensitivity = TP/(TP + FN)$;
- $Specificity = TN/(FP + TN)$; and
- $Accuracy = (TP + TN)/(TP + FN + FP + TN)$.

Table 3.4 demonstrates the measured accuracies of the 30 detection heuristics. We next present our empirical analysis on the average accuracies from three aspects.

File, API, and Property Heuristics



We first inspected the average accuracies of the heuristics according to the categories of their exploited observable artifacts. As shown in the above bar chart and Table 3.4, the file heuristics enjoyed both high sensitivities and specificities with an average accuracy of 97.8%. The API heuristics, despite of their acceptable sensitivities, suffered from significantly low specificities. For example, A2, A9, and A10 performed no better than 50/50 guesses as their accuracies were less than 50%. The property heuristics performed fairly good with an average accuracy of 89.5%.

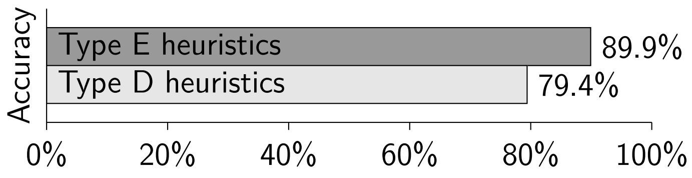
Table 3.4: Evaluation Results of the 30 Heuristics

	TP	FN	FP	TN	Sens.(%)	Spec.(%)	Acc.(%)
F1	9	0	2	126	100.0	98.4	98.5
F2	9	0	0	128	100.0	100.0	100.0
F3	9	0	7	121	100.0	94.5	94.9
F4	9	0	4	124	100.0	96.9	97.1
F5	9	0	1	127	100.0	99.2	99.3
F6	9	0	1	127	100.0	99.2	99.3
F7	9	0	7	121	100.0	94.5	94.9
F8	9	0	0	128	100.0	100.0	100.0
F9	9	0	0	128	100.0	100.0	100.0
F10	9	0	8	120	100.0	93.8	94.2
A1	9	0	3	125	100.0	97.7	97.8
A2	7	2	82	46	77.8	35.9	38.7
A3	5	4	24	104	55.6	81.3	79.6
A4	7	2	48	80	77.8	62.5	63.5
A5	7	2	45	83	77.8	64.8	65.7
A6	9	0	37	91	100.0	71.1	73.0
A7	9	0	64	64	100.0	50.0	53.3
A8	9	0	37	91	90.0	71.1	72.5
A9	6	3	72	56	66.7	43.8	45.3
A10	9	0	82	46	100.0	35.9	40.1
P1	8	1	2	126	88.9	98.4	97.8
P2	8	1	17	111	88.9	86.7	86.9
P3	8	1	21	107	88.9	83.6	83.9
P4	9	0	5	123	100.0	96.1	96.4
P5	9	0	2	126	100.0	98.4	98.5
P6	9	0	11	117	100.0	91.4	92.0
P7	9	0	14	114	100.0	89.1	89.8
P8	8	1	10	118	88.9	92.2	92.0
P9	8	1	0	128	88.9	100.0	99.3
P10	9	0	20	108	100.0	84.4	85.4

One possible explanation for the API heuristics' low accuracies is that the Android APIs are designed to provide some sort of hardware/software abstraction. An evidence is the Android Compatibility Program [2] , which precisely defines the behaviors of Android APIs to ensure that Android applications run in "a consistent and standard environment." To build such an environment, the APIs that reveal the underlying details are not necessary, and they are subject to be removed or deprecated. However, we argue that this environment also requires a well-configured application sandbox to prevent applications from bypassing the APIs. Unfortunately, our discovered file and property heuristics imply that the current sandbox should be reinforced.

[2]`https://source.android.com/compatibility`

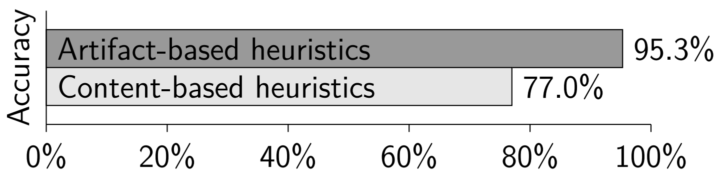
Type E and Type D Heuristics



We investigated the differences between the Type E and Type D heuristics. As we have discussed in Section 3.2.3, Type E and Type D detection heuristics respectively indicate emulators and real devices. In our experiments, the Type E heuristics outperformed the Type D ones.

We note that almost all of the heuristics in Table 3.4 with low specificities are of Type D. We believe that it is due to the diversified and fragmented nature of real devices. Type D heuristics expect the artifacts/tokens that are prevalent in real devices. However, device manufacturers inevitably customize devices and change artifacts, which makes it harder to find the artifacts that exist in every real device. On the contrary, emulators are much more unified in terms of customizations, which is possibly due to the difficulty in modifying and maintaining software-emulated hardware.

Artifact-based and Content-based Heuristics



Finally, we compared the artifact-based and content-based heuristics. The bar chart shows that the artifact-based heuristics had an average accuracy of 95.3%. The content-based heuristics fell behind with 77.0%. However, we stress that F1, F4, and

P9 are also content-based heuristics and their accuracies were among the top of the 30 heuristics.

In addition to the factors of abstraction and customization that we discussed earlier, a possible explanation is that content-based heuristics are more subtle and vulnerable to intended or unintended changes. Content-based heuristics exploit artifacts' contents (*e.g.*, configurations), which are subject to change in a rapidly evolving system like Android. On the contrary, artifact-based heuristics rely on the presence of certain artifacts. Compared with the contents, the artifacts (*e.g.*, kernel modules) are relatively consistent in emulators and real devices, because developers are usually reluctant to remove them as to avoid unexpected problems.

3.4.3 Case Study: A9

Finally, we present a case study on heuristic A9 because it involves an Android API `getDeviceId`, which has been popular among the known detection heuristics. A9 exploits the same API but in a slightly different way. Specifically, it looks for a token "`\0\0\03`" in the Binder IPC message returned from the implementation of `getDeviceId` in `TelephonyManagerService`. In other words, A9 uses the IMEIs whose first character is "`3`" to indicate real devices. However, it turned out that A9 had a sensitivity of 66.7% and an unbearable specificity of 43.8%. We investigated the evaluated emulators and real devices to find out how they reacted to A9. The investigation led us to flaws in an existing anti-detection technique and improvements for A9.

A9 vs DroidBox 4.1

As we discussed in Section 3.3.1, our implementation of the artifact retriever employs a Java function caller and a Binder IPC caller to probe Android APIs. When we tested A9 against DroidBox 4.1, we found that these two callers returned disparate values, namely “`357242043237511`” and “`000000000000000`.” We note that both callers should return the same value, because the application-side Binder proxies of Android APIs are not supposed to modify the IPC messages returned by the underlying Binder stubs in system services.

We inspected DroidBox’s source code and found that DroidBox rewrites the Binder proxy of `getDeviceId` to return a dummy IMEI without involving the Binder stubs. Although such a countermeasure could neutralize the detection heuristics that call `getDeviceId` in a normal way, it is not effective against the Binder IPC caller, which bypasses the countermeasure and reads the actual full-zero IMEI. Therefore, we believe that the countermeasure implemented in DroidBox 4.1 is not complete. We note that ApkScan demonstrated the same issue, implying that ApkScan might have integrated DroidBox 4.1 for its dynamic analysis.

However, A9 was not effective against DroidBox 2.3. We found that DroidBox 2.3 opts for a similar countermeasure but implements it in the service-side Binder stub. In such a case, bypassing the stub and observing the actual IMEI would require root privileges, *i.e.*, the actual IMEI is not observable. Therefore, such a countermeasure is effective and the dummy IMEI would appear realistic.

A9 vs Non-U.S. Devices

A9 assumes that an Android device whose IMEI starts with “`3`” is a real device, otherwise it is an emulator. We checked the IMEIs of the 128 real devices and found that this assumption is incorrect.

According to IMEI Allocation and Approval Guidelines [28], the first digit of an IMEI is part of Reporting Body Identifier (RBI), which identifies the GSMA-approved authority that issues the IMEI. Typically, IMEIs of mobile devices are issued by the authorities in the same area where the devices are sold. For example, IMEIs of the devices sold in the U.S. are issued by the British Approvals Board for Telecommunications (BABT) and thus start with BABT's code "`35`." Similarly, IMEIs of the devices sold in China start with "`86`." We note that about half of the 128 evaluated real devices were from Baidu MTC that uses Android phones sold in China. Given that A9 was based on the devices in the U.S., A9 naturally got a low specificity, and it could be improved with wild cards that match multiple RBIs.

The lesson of A9 indeed illustrates the future of the armed race between emulator detection and anti-detection. First, Android malware could check the semantics of the observed artifacts. For example, the dummy IMEI in DroidBox 4.1 is invalid and could be noticed by a sophisticated adversary. Second, emulator-based malware analysis tools should consider the observability of actual artifacts and the semantics of dummy artifacts to be less distinguishable.

3.5 Discussion

The evaluation results imply an imminent threat that Android malware may thwart existing emulator-based dynamic analysis systems. In this section, we suggest the potential countermeasures and discuss the limitations of our work.

3.5.1 Countermeasures

Provisional countermeasures. We suggest the methods that detect the usage of detection heuristics in Android malware as provisional countermeasures. Although they do not prevent Android malware from detecting Android emulators, they can raise alarms for analysts and thus thwart the malware's original purpose of evading analysis. For example, dynamic analysis systems could monitor accesses on files and properties seldom used by benign applications. API heuristics are much more stealthy because benign applications also frequently use them. In such a case, we suggest static data-flow analysis to locate branches that involve detection heuristics and lead to disparate code blocks.

Short-term countermeasures. Next, we discuss the countermeasures that allow an emulator to appear "realistic" to Android malware. First, we suggest a comprehensive deployment of dummy artifacts. Some existing works can be adapted to facilitate such countermeasures. For example, AirBag [118] supports a decoupled and isolated runtime environment based on OS-level virtualization. ASM [72] provides programmable interfaces that interpose Android APIs and return dummy values to applications. These works, if combined and extended, can enable a "brain in a vat" setup where an application runs in an emulator but gets dummy and valid data originated from real devices. Second, we suggest denying accesses on unnecessary observable artifacts with strict DAC and MAC policies. For example, artifacts in sysfs exploited by our file heuristics seem unnecessary for general Android applications. However, the usability impact of denying accesses still needs further verification.

Long-term countermeasures. The ideal countermeasure is to fix all the discrepancies in Android emulators. Although Garfinkel *et al.* [63] concludes its infeasibility in 2007 due to the inherent hardness of creating indistinguishable software-emulated hardware, hardware-assisted virtualization techniques (*e.g.*, Intel VT-x and VT-d) have evolved significantly to allow PC emulators to virtualize real hardware. Currently, ARM CPUs have integrated necessary virtualization extensions. Meanwhile, commodity ARM hypervisors are also in active development. We envision emerging Android emulators equipped with virtualized CPUs, sensors, and radios in the near future.

3.5.2 Limitations

Despite the robustness of Morpheus, the quality of the discovered detection heuristics is limited by the small number of real devices used in finding detection heuristics (Section 3.3). In general, Morpheus works like supervised learning, and its performance inevitably depends on the quality of the "training set," *i.e.*, the emulators and real devices observed by the artifact retriever. We note that the artifact retriever needs approximately 20 minutes to collect the artifacts on a single device. Unfortunately, online services like AppThwack (Section 3.4) do not allow the artifact retriever to run for such a long time or upload large bulks of data. As for future work, we plan to reach out to mobile carriers and device vendors to collect observable artifacts from more real devices.

Although Morpheus discovered more than 10,000 heuristics, we stress that they were derived only from 3 out of 33 sources of observable artifacts. To better understand the scope of detection heuristics for effective countermeasures, the artifact retriever could be enhanced to address more sources of artifacts as well as sophisticated usages of them. Examples include extended modules of the artifact retriever that can handle callbacks or construct valid input parameters for Android APIs.

Our heuristic generator produces relatively rigid heuristics, such as A9 that does not match multiple RBIs. This can be improved with more sophisticated and flexible heuristics. For example, a token-sequence heuristic matches an ordered set of tokens in the contents of an artifact. Moreover, a naïve Bayes heuristic enables probabilistic matching by aggregating the empirical probabilities of multiple artifact/token heuristics with the Bayes' law, assuming that the occurrences of artifacts/tokens are independent.

Considering that users may need to specify their preferences for selecting heuristics, the flexibility of the heuristic selector could be improved. For example, users may prefer sensitive heuristics with respect to prioritizing countermeasures. Unlike PC malware that might have incentives for infecting VMs, Android malware must evade emulators so they need sensitive heuristics to guarantee low false negatives. As our future work, we plan to extend the heuristic selector to support user-defined scoring functions.

3.6 Related Work

Behavior-based detection heuristics. Researchers have proposed several heuristics that exploit discrepancies in runtime behaviors rather than artifacts. For instance, a piece of specially crafted native code can identify QEMU-based emulators due to the discrepancies in QEMU's caching behaviors [88, 100, 107]. Low video frame rate indicates emulators because of the performance drawbacks in the SDK emulator's graphics rendering engine [112]. However, these heuristics are not evaluated against VirtualBox-based emulators and real devices. Thus, their sensitivities and specificities require further investigation. In addition, these heuristics do not return a decision until a sufficient number of events are observed, which tends to increase their footprints and attract analysis. Along these lines, Morpheus addresses artifact-based and content-based detection heuristics. More importantly, Morpheus generates detection heuristics automatically and systematically.

Dynamic analysis frameworks. Researchers have built several dynamic analysis frameworks to vet the runtime behaviors of Android malware. TaintDroid [52] tracks information flows that leak sensitive data to the Internet. VetDroid [123] further reveals information flows that involve permissions. AppIntent [122] helps determine if an information flow is user-intended. In particular, DroidScope [121] analyzes applications from outside emulators using virtual machine introspection. Some of these tools have been integrated into automated malware analysis systems such as DroidBox [10], Andrubis [11], CopperDroid [105], SandDroid [20], and Trace-Droid [21]. They are vulnerable to be evaded using the detection heuristics in this work as long as they are deployed in Android emulators.

3.7 Summary

Recent Android malware demonstrates the capabilities of detecting Android emulators using detection heuristics. To convey the severity of this problem, we have presented Morpheus, a system that automatically and systematically generates detection heuristics. Morpheus analyzes artifacts observable by Android applications and discovers exploitable discrepancies in Android emulators. Moreover, we have described a proof-of-concept implementation of Morpheus, along with extensive experiments and findings.

Chapter 4

MONITORING AND CONFINEMENT: DEFENSE

Based on the lessons we learned from the work discussed in the previous chapter, we propose comprehensive monitoring and confinement mechanisms that can be flexibly configured to reveal and to stop Android apps' attempts to abuse their privileges.

4.1 Problem Statement

Given the diversified attacks, it is imperative to remedy Android's default security mechanisms to provide better security guarantees. Recently, a wide spectrum of security extensions has been proposed to implement enhanced MAC in Android. Depending on how they implement MAC, we can divide them into two categories: MAC in the Android system, and MAC in the apps.

The effectiveness of MAC implemented in operating systems has been well proved by security frameworks such as SELinux. Such MAC implementations require patching and/or recompiling the system. In recent years, plenty of security frameworks [30, 36, 37, 44, 95, 109, 129] follow this approach to (1) add kernel MAC for reinforcing application sandboxes; and/or (2) add middleware MAC to remedy the shortcomings of permissions. The kernel MAC implementations reuse or extend previous Linux MAC frameworks such as SELinux and TOMOYO Linux. The middleware MAC implementations are usually tailored to the problems they attempt to address. In particular, FlaskDroid [37] is the first security extension that attempts to address diverse security requirements with a generic security framework. However, generic MAC implementations come with a cost of relatively large code base and difficulties in maintainability. For example, FlaskDroid includes 12 Userspace Object Managers

(USOM). Each USOM must be re-evaluated and patched once new Android versions are released. And it remains a question whether the 12 USOMs can completely cover all the attack vectors. Android Security Framework (ASF) [30] attempts to address the limitations of FlaskDroid with loadable security modules and a comprehensive set of security APIs. Despite that the loadable modules could reduce the work required to instantiate different security models, the security APIs themselves that are scattered in various Android system services still suffer from deployment and maintenance issues.

To address the deployment issues of system-centric MAC implementations, application-centric approaches [45, 46, 77, 104, 119] have been proposed. These security frameworks, which are also known as inlined reference monitors, rewrite apps by instrumenting apps' byte code or native code and thus do not require any changes to the underlying Android system. Despite their significant improvement in deployment, a recent study [69] shows that application-centric MAC implementations are subtle and they could be bypassed or subverted, because the instrumented code runs within the same process as that of the confined app. Apparently, an app is able to modify itself to remove or suppress the instrumented code.

In this chapter, we will re-explore the problem of designing and implementing a practical MAC framework for Android. We propose a multi-layer security framework called TripleMon. Unlike previous work, TripleMon opts for a system-centric and IPC-oriented approach. Specifically, TripleMon extends the IPC subsystem of Android to mediate various types of IPC channels that can be used by Android apps to access resources outside their sandboxes. While supporting complete mediation and tamperproofness, TripleMon also enables users to flexibly control the capabilities of apps in a fine-grained manner.

4.2 A Multi-layered Approach to Protect Android

Mandatory access control is a type of access control by which the operating system confines the abilities of subjects to access certain objects based on a centralized security policy. An effective MAC implementation, no matter whether it is system-centric or application-centric, should fulfill at least the following design goals:

G1 *Complete mediation.* We should implement mechanisms that completely mediate access vectors that could be used by a subject in the system, so that the subject cannot bypass our mechanisms.

G2 *Tamper proof.* We should prevent attackers from undermining our security mechanisms. This includes protecting the integrity of the security mechanisms themselves as well as protecting security policies.

Towards a practical MAC implementation, we further propose the additional design goals:

G3 *Generic and flexible.* We should be able to dynamically re-configure the security mechanisms using flexible security policies.

G4 *Unified policy scheme.* Composing and managing security policies could be tedious and error-prone for system administrators. To reduce the workload of policy management, we expect the security policy to use a unified scheme for different types of subjects, objects, and operations.

G5 *Easy maintenance.* The maintenance of the code should be simple across different OS versions. In particular, it should require little work to re-evaluate the effectiveness of the MAC implementation against newer OS versions.

To achieve these goals, we propose TripleMon, which is a multi-layer and policy-agnostic MAC implementation. Figure 4.1 depicts the proposed architecture of TripleMon, which consists of a set of reference monitors and a decision manager. TripleMon

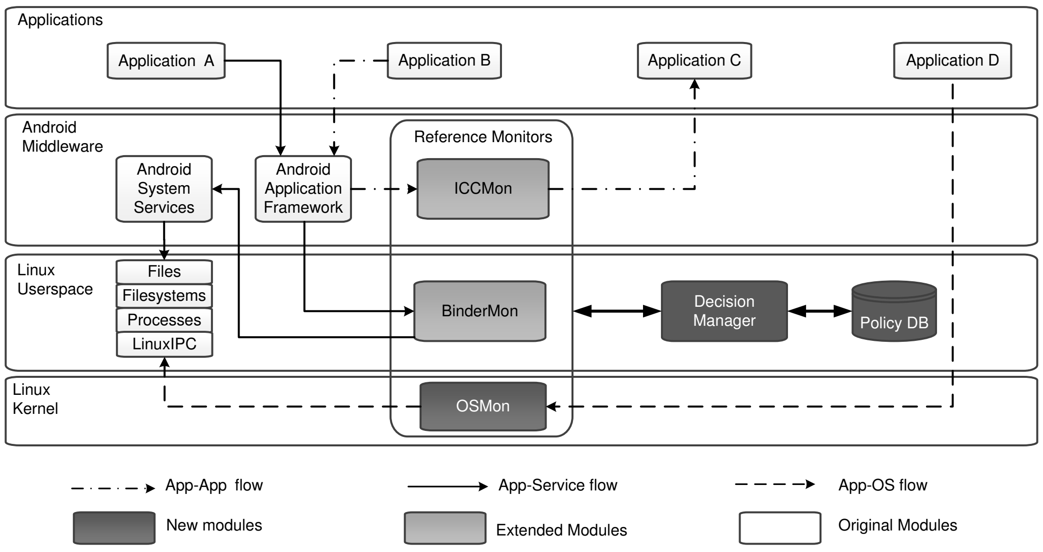


Figure 4.1: Proposed Multi-layer Security Framework for Android

opts to a multi-layer design because Android itself is composed of three layers: apps, middleware, and Linux. Each layer has its respective access control semantics and thus requires a dedicated reference monitor.

TripleMon's reference monitors use a different design compared with FlaskDroid. FlaskDroid achieves G3 by placing enforcement hooks in various Android system services, such as ActivityManager, SensorManager, and TelephonyManager. However, it is hard to justify the completeness (G1) of such an approach, because it is infeasible and futile to hook every function in more than 50 system services in Android. And maintaining various hooks here and there is also difficult in a fast evolving operating system like Android (G5). To address this issue, we are inspired by SELinux which uses few hooks that are actually "choke points" of sensitive operations. In particular, we identify the boundaries of IPC channels as our choke points because Android apps *must* use IPC to access sensitive resources outside their sandboxes. Furthermore, Android IPC mechanisms are part of the core libraries and they have not been changed remarkably compared to the system services that FlaskDroid depends on. Therefore, mediating ICC, Binder IPC, and Linux IPC channels allows TripleMon to minimize the cost of G5 while satisfying G1 and G3. Moreover, the reference monitors at lower layers (OSMon) can protect the integrity of reference monitors at higher layers (BinderMon and ICCMon).

TripleMon also includes a decision manager to address the semantic gaps among reference monitors. Semantic gaps arise when a resource corresponds to multiple objects at different layers. For example, an app can take a picture using (1) the camera app; (2) the system service that controls the camera; or (3) the device node of the camera (`/dev/video0`). Suppose we are to revoke an app's capabilities of taking pictures, the decision manager can issue decisions for all the reference monitors to block accesses at all layers. In addition, the decision manager also follows a unified policy scheme and resolves inconsistencies once conflicting access control decisions are generated for the same object. The unified policy scheme also makes it easier to analyze security policies at different layers to evaluate the system-wide assurance.

4.2.1 Reference Monitor for ICC

We propose ICCMon to handle ICC requests. We choose ActivityManagerService as the choke point for ICCMon because ActivityManagerService is the single system service responsible for establishing and shutting down ICC channels.

Figure 4.2 depicts the workflow of ICCMon. Specifically, ICCMon labels components with the UIDs of their corresponding app processes, and labels intents with their meta-information including action, category, and data URI. To ensure complete mediation, we analyze the call graph of ActivityManagerService and identify 4 functions that can capture all ICC requests, including `startActivityLocked()`, `startServiceLocked()`, `getContentProvider()` and `deliverToRegistered BroadcastReceiversLocked()`. From the names of these functions we can notice that they handle ICC channels to activities, services, broadcast receivers, and con-

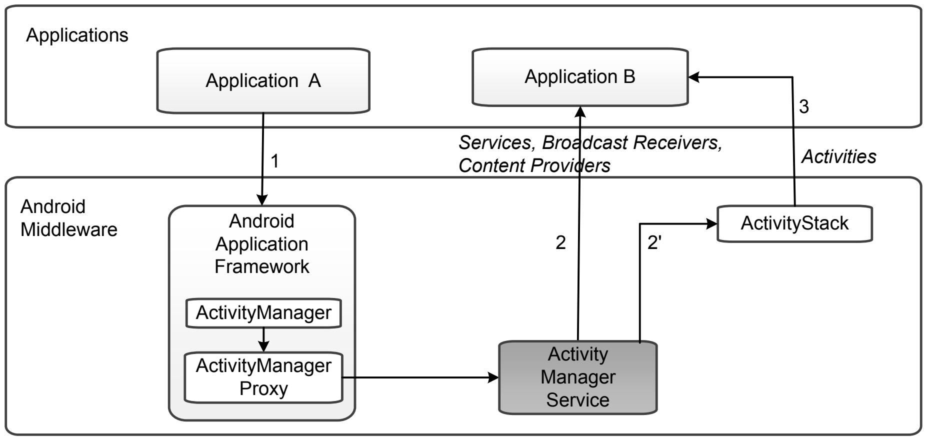


Figure 4.2: ICCMon Workflow

tent providers, respectively. These hooks placed just before ActivityManagerService is about to establish an ICC channel, which is similar to how hooks are placed in Linux Security Modules. Using these hooks, ICCMon enforces security decisions acquired from the decision manager. ICCMon returns control to ActivityManagerService if an ICC request is accepted. Otherwise, ICCMon generates a security exception for the caller app to shut down the ICC channel.

4.2.2 Reference Monitor for Binder IPC

Android APIs are implemented based on Binder IPC. The mappings between Android APIs and Binder IPC transactions are specified in the AIDL [1] files provided with the source code of Android. For example, Binder IPC requests on *whose destination is a system service called* ***`iphonesubinfo`*** *and whose command code is 1* correspond to an API `getDeviceID`. Therefore, we can infer the semantics of any Binder IPC request by checking corresponding Android API that it can be mapped to.

[1] Android Interface Definition Language

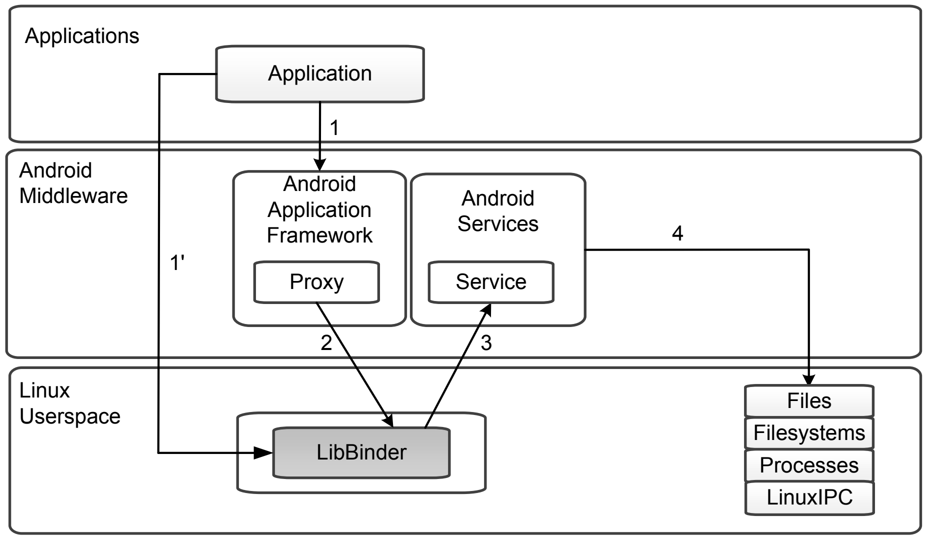


Figure 4.3: BinderMon Workflow

To identify choke points of Binder IPC, we analyze the entire Binder IPC subsystem and we choose to place the enforcement hooks in LibBinder (`/system/lib/libbinder.so`), because it is the dynamic library linked to all the system services. As long as LibBinder and the system services are not compromised, BinderMon can intercept all the Binder IPC requests sent to the system services.

BinderMon mediates Binder IPC before the permission framework is consulted. Therefore, it enables dynamic permission revocation without affecting the existing permission framework. Furthermore, BinderMon offers finer granularities at the API level, whereas the granularity of the Android permission framework is a set of permissions where each permission maps to multiple APIs. Moreover, BinderMon protects 1,448 public and private Android APIs implemented by more than 70 system services, while FlaskDroid only protects 136 APIs in 12 system services.

A detailed workflow of BinderMon is shown in Figure 4.3. To make an API call, an app sends an Binder IPC request using the proxy of the API's corresponding remote system service (Step 1). Alternatively, this app can send a request without involving the proxy (Step 1'). BinderMon intercepts both types of requests and queries the decision manager with the caller app's UID, the callee service's name, and the command code which indicates the API to be called. If the request is allowed, BinderMon returns control to LibBinder. Otherwise, BinderMon shuts down the IPC channel and returns an error code `PERMISSION_DENIED` to the caller app.

4.2.3 Reference Monitor for Linux IPC

The default Android app sandbox is flawed. Although Android apps are expected to use Android APIs to access system resources, they still possess capabilities to bypass APIs and access sensitive Linux IPC channels that general Android apps should not necessarily use. However, ICCMon and BinderMon cannot revoke all of these unnecessary capabilities. Therefore, we propose a kernelspace reference monitor, OSMon, to mediate Linux IPC channels and enforce the principle of least privilege for Android apps.

Figure 4.4 depicts the workflow of OSMon. Similar to SELinux [110] and TOMOYO Linux [70], we identify choke points as the hooks defined by Linux Security Module (LSM) [117]. However, OSMon only use the hooks are are specific to the Linux IPC mechanisms. Table 4.1 shows the proposed 23 hooks which cover objects like inodes, file systems, tasks, local sockets, Unix-domain sockets, and Netlink sockets. These hooks can be enabled or disabled individually in the kernel configuration file. In addition, OSMon plays as the trusted computing base (TCB) of the entire TripleMon framework. It protects all the userspace components of TripleMon, including ICCMon, BinderMon, the decision manager, and the policy database. Moreover, OSMon protects itself against root exploits by depriving untrusted apps' capabilities to access IPC channels that may subvert OSMon.

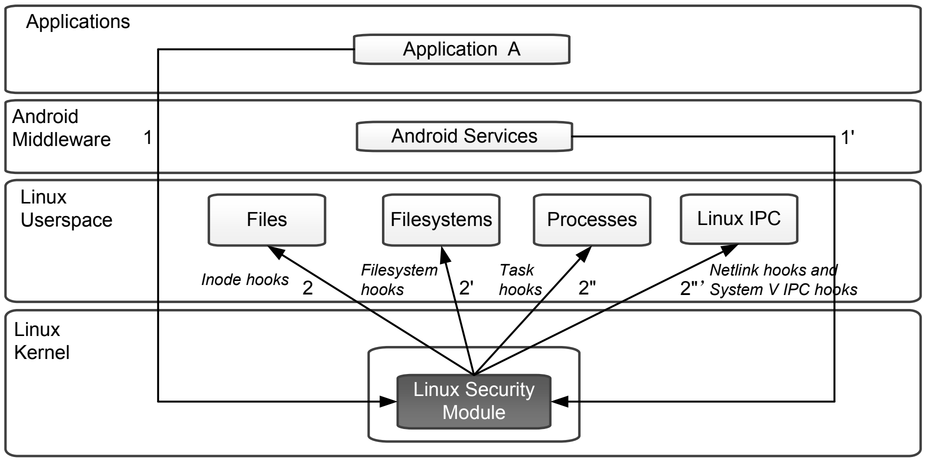


Figure 4.4: OSMon Workflow

Table 4.1: Implemented Hooks in OSMon

Hook category	**Hook name**
Local socket	socket_create, socket_connect, socket_bind, socket_send
Unix-domain socket	ud_connect, ud_send
Netlink socket	netlink_send, netlink_recv
Task	task_create, task_setuid, task_setgid, task_kill
File	inode_create, inode_rename, inode_mkdir, inode_rmdir, inode_link, inode_symlink, inode_unlink, inode_setattr, dentry_open
File System	sb_mount, sb_unmount

4.2.4 Multi-layer Policy Modules

Our unified policy scheme for TripleMon is defined as follows:

Definition 1 *(**Target**). A target is a 3-tuple $< Subject, Resource, Action >$, where*

- Subject *is a set of entities to which the authorization is granted;*
- Resource *is a set of entities to which accesses need to be mediated; and*
- Action *is a set of actions being authorized or forbidden.*

Definition 2 ***(Access Control Policy)***. *An access control policy is a 3-tuple* $\{Target, Condition,\ Effect\}$*, where*

- $Target$ *decides whether an access request is applicable to the policy. The target specification is defined in Definition 1;*
- $Condition$ *specifies restrictions on the attributes in the target and refines the applicability of the policy; and*
- $Effect \in \{accept, deny\}$ *is the authorization effect of the policy.*

To evaluate an access request over access control policies, if the request satisfies both the target and condition of a policy, the response is sent with the decision specified by the effect element in the policy. Otherwise, the response yields "deny".

We next describe three kinds of TripleMon policy: ICC policy, Binder policy and OS policy in detail. Table 4.2 summarizes the major components contained in three kinds of TripleMon policies.

ICC Policy

ICC policies regulate the intent-based IPC between applications. Thus, *application group*, *application*, and *component* comprise the `subject` and `resource` of an ICC policy. `Actions` correspond to the APIs that initiate ICC to four types of components, namely activities, services, broadcast receivers and content providers. `Condition` is defined as the attributes used in intents. Table 4.2 illustrates the elements of ICC policy.

For example, an adversary who does not possess the INTERNET permission may still access the Internet by sending an intent with action `ACTION_VIEW` and the url to the Browser application. We can mitigate such a privilege escalation attack by

Table 4.2: Components in TripleMon Policies

		ICC Policy	**Binder Policy**	**OS Policy**
Target	Subject	Application Group Application Component	Application Group Application	Application Group Application System File
	Resource	Application Group Application Component	Service	Linux IPC channel Process File Filesystem
	Action	startActivity bindService sendBoradcast accessContentProvider	Call	Linux IPC Task Inode Filesystem
Condition		Action Category Data	Service cmd code	System call's name Parameters
Effect		Accept Deny	Accept Deny	Accept Deny

specifying and enforcing an ICC policy to prevent the adversary from sending such an intent to the Browser application.

Binder Policy

Binder policies specify the behaviors of Binder IPC channels between applications and system services. Thus, *application group* and *application* comprise the `subject`, and *system service* comprises the `resource`. `Action` has only one instance, *call*. And `condition` specifies the *command code* of the command to be executed by the remote services, as shown in Table 4.2.

For instance, as we discussed previously, the permission `READ_PHONE_STATE` allows an application to access resources such as the unique device IDs and the phone's state. We can set a Binder policy to revoke an application's privilege of accessing the device IDs but keep an access privilege to the phone's state without reinstalling the application. Compared with the default permission framework which treats the capabilities for a permission as an indivisible block, our approach obviously enables a more fine-grained and revocable access control.

OS Policy

OS policies mediate Linux IPC channels. In an OS policy, as illustrated in Table 4.2, `subject` consists of *application group*, *application*, *system* and *file*. `Resource` consists of *file*, *filesystem*, *process* and *Linux IPC channel*. `Action` has four types: *inode*, *filesystem*, *task*, and *Linux IPC*. These actions correspond to different categories of operations on Linux IPC channels. `Condition` specifies the name of the exact operation and optional parameters.

As we mentioned earlier, GingerBreak is a root exploit that attacks a system service called Vold by sending forged and malformed Netlink messages. Such an exploit can be easily prevented by defining an OS policy that disables applications from sending Netlink messages. Indeed, Android applications are currently not strictly prohibited from using some operating system features. Malicious applications may take advantage of these features to launch attacks. Thus, we need to define corresponding OS policies to prevent applications from abusing these features.

4.2.5 Decision Manager

The decision manager is the policy decision point of TRIPLEMON where IPC requests are evaluated against authorization policies. It centrally issues access control decisions for each reference monitor, manages security policies, and resolves conflicts among reference monitors.

Communication with Reference Monitors

As the only policy decision point in TripleMon, the decision manager runs in a dedicated process and communicates with the reference monitors via a local socket interface. The communication follows the policy scheme we defined in Section 4.2.4. Indeed, this interface needs special attention because it can subvert TripleMon or cause denial of service if exploited. To protect this interface, OSMon enforces a set of top-priority policy rules to only allow the decision manager to write into the interface.

Policy Management

The decision manager parses a JSON-like plaintext policy file that follows the policy scheme defined in Section 4.2.4. The policy file is stored in an internal read-only filesystem. In addition, OSMon enforces rules to disallow access on the policy database from any process except the decision manager.

The decision manager maintains two tracks of security policies, namely *slow track* and *fast track*. The slow track is enforced by OSMon only. This track defines the common and least privileges of general installed applications. The fast track is enforced by ICCMon and BinderMon. This track expects more frequent policy updates because ICCMon and BinderMon are required to meet per-application security requirements. Specifically, ICCMon or BinderMon always consults the decision manager to acquire decisions generated based on the latest fast track policy. On the contrary, OSMon caches a copy of the slow track policy in the kernel and makes decisions by itself. It only consults the decision manager to get the latest slow track policy when it initializes itself.

Decision Reconciliation

Decision reconciliation is necessary to resolve conflicts when multiple reference monitors are involved to mediate accesses on the same resource. For example, an adversary attempts to access the camera via ICC and Binder IPC, expecting that there would be a capability leak somewhere. Suppose a strategy, `deny-overrides`, is applied. The decision manager ensures that ICCMon and BinderMon both deny the requests to access the camera if there is any applicable ICC or Binder policy that evaluates to deny. More flexible strategies[85], such as `first-applicable` and `strong-consensus`, can be adopted to resolve the decision inconsistencies.

4.3 Evaluation

In this section we evaluate TripleMon in terms of its coverage, effectiveness and performance overheads. Our experiments were performed on a Galaxy Nexus running Android 4.2.2 and Linux kernel 3.0.31.

4.3.1 Experimental Setup

Similar to other policy-driven MAC implementations, TripleMon requires a good security policy to be effective. In TripleMon, we opt to semi-automatically and iteratively derive policy rules from applications' runtime behaviors. We first configured TripleMon into a permissive mode where it only logs IPC requests. We then executed a set of benign applications and a set of malicious applications. By comparing the observed IPC requests, we identified the IPC requests to be allowed or denied in our security policy.

For the set of benign applications, we handpicked 10 applications (Table 4.3) from Google Play's top charts. These applications are from renowned developers and under different application categories. We assumed that these applications are trusted by

Table 4.3: Applications Assumed to be Benign and Trusted by General Users

Application	**Category**
AmazonMobile	Shopping
BejeweledBlitz	Game
ChaseMobile	Finance
Dictionary.com	Books & Reference
Dropbox	Productivity
Google+	Social
GooglePlayMovies&TV	Media & Video
Hangouts(replacesTalk)	Communication
MoviesbyFlixster	Entertainment
Yelp	Travel & Local

general users and we used them to outline the general expected runtime behaviors of Android applications. The malicious applications were 1,260 malware samples from the Android Malware Genome Project. We fuzzed each application through 5 iterations with randomized touch inputs and system events.

4.3.2 Coverage

The generated policy demonstrated the coverage of system resources protected by TripleMon. Compared to one of closely related work FlaskDroid [37], TripleMon provides the same level of protection on ICC channels and much more protection on Binder IPC channels. Table 4.4 shows the system services that appeared in TripleMon's security policy but cannot be protected by FlaskDroid. For example, AccountManager needs enhanced protection because it is a centralized registry of a user's online accounts.

In terms of Linux IPC channels, TripleMon provides relatively less coverage compared to FlaskDroid whose kernel MAC is built upon SELinux. Indeed, Triple-

Table 4.4: System Services Protected by TripleMon

Service	**Example APIs**
AccountManager	getAccounts, getPassword, peekAuthToken, invalidateAuthToken
AlarmManager	setTimeZone
BackupManager	setBackupEnabled, setAutoRestore
Bluetooth	createBond, isDiscovering, getUuids, getScanMode
ConnectivityManager	getActiveNetworkInfo, getProxy, tether, startUsingNetworkFeature
EmailService	searchMessages, loadAttachment, sendMeetingResponse
NFCManager	setForegroundDispatch, setNdefPushCallback
SipService	open, close, createSession, setRegistrationListener
VibrationService	vibrate, vibratePattern
WifiManager	setFrequencyBand, getWifiApConfiguratin, getScanResults

Mon only implements a subset of hooks used by SELinux. However we note that OSMon follows a policy schema that is consistent with BinderMon and ICCMon. And a consistent policy schema is necessary for efficient policy management. Compared to FlaskDroid that is more likely putting two incompatible reference monitors together, TripleMon's three reference monitors augment each other and behave as a single reference monitor.

4.3.3 Case Studies

To further validate the effectiveness of TripleMon, we tested TripleMon against real malware samples and synthetic applications that implement the attacks that we discussed in Chapter 2.

ICCMon vs Information Stealing Malware

To test ICCMon, we selected a malware strain called Gone60 (short for "gone in sixty seconds"). Gone6- accesses the Contact applications via ICC channels and uploads user's contacts to remote servers. In our experiments, we put the malware samples into an application group designated for testing suspicious applications. Then, we applied the following ICC policies to revoke this application group's capabilities to access user's contacts. The experiments on 9 Gone60 samples demonstrated that TripleMon successfully denied the accesses on the contacts.

```
"ICCPolicy_Gone60": {
    "type"  : "ICC"
    "target": {
        "subject"   : ["GROUP_suspicious"],
        "resource"  : ["com.android.contacts"],
        "action"    : ["ContentProvider", "Activity"]
    },
    "condition"     : ["*"],
    "effect"        : "deny" }
```

BinderMon vs Coarse-grained Permissions

Next, we validated BinderMon by partially revoking the capabilities covered by a permission called `READ_PHONE_STATE` for privacy purposes. `READ_PHONE_STATE` is a commonly abused permission because it allows applications to call an API `getDeviceID` and read the unique device identifier which could facilitate user tracking. However, simply revoking this permission could break applications because this permission also covers APIs other than `getDeviceID`. To protect user's privacy, we employed the following Binder policies to revoke an application's capability to call `getDeviceID` without affecting the other APIs. In our experiments on 20 randomly selected malware samples that use `getDeviceId`, BinderMon denied every request to `getDeviceID`. We further verified the results by inspecting the files and network traces. And we discovered that no information related to device ID was leaked.

```
"BinderPolicy_CapabilityRevoking": {
    "type"  : "BINDER"
    "target": {
        "subject"   : ["GROUP_suspicious"],
        "resource"  : ["iphonesubinfo"],
        "action"    : ["Call"]
    },
    "condition"     : ["cmd=1"],
    "effect"        : "deny" }
```

OSMon vs Root Exploits

In Table 4.5, we show a list of known root exploits [126], vulnerabilities, and programs attacked by the exploits. We also show the hooks of TripleMon used to prevent corresponding exploits from gaining root privileges. Denying `setuid` is a straightforward countermeasure and can prevent all of exploits from escalating their privileges. We also employed alternative hooks to protect the target of exploits. For example, `Exploid` sends malformed Netlink messages to the kernel via `/proc/sys/kernel/hotplug`. We can mitigate this exploit by revoking its capabilities to use Netlink, and/or use `/proc/sys/kernel`. In addition, most exploits attempt to remount `system` as read-write to retain their root privileges even after the reboot. Thus, the filesystem hooks of TripleMon are helpful to neutralize such attempts.

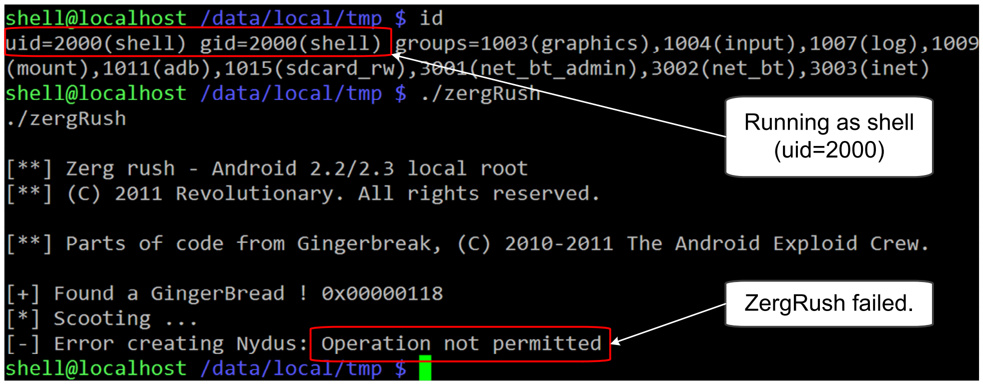
In our experiments, we tested two exploits, GingerBreak and ZergRush, which have been used by recent malicious applications [8, 9]. GingerBreak and ZergRush attack Vold's Netlink socket and local socket, respectively. Therefore, we defined the following policies to prevent unauthorized apps from accessing these sensitive sockets exposed by Vold.

Table 4.5: Root Exploits and Countermeasures with TripleMon

Root Exploit	Vulnerable Program	OSMon Hooks
Asroot [2]	kernel	localsocket, setuid
Exploid [1]	init	netlink, setuid, inode
Zimperlich [4]	zygote	task_create, setuid
RAtC [3]	adbd	task_create, setuid
KillingInTheNameOf [6]	ashmem	setuid
Psneuter [6]	ashmem	setuid
GingerBreak [8]	vold	netlink, setuid
ZergRush [9]	libsysutils	localsocket, setuid
Mempodipper [12]	kernel	inode, setuid

```
"OSPolicy_Gingerbreak": {
    "type"  : "OS"
    "target": {
        "subject"   : ["GROUP_suspicious"],
        "resource"  : ["vold"],
        "action"    : ["netlink"]},
    "condition"     : ["cmd=netlink_send"],
    "effect"        : "deny" }
```

```
"OSPolicy_ZergRush": {
    "type"  : "OS"
    "target": {
        "subject"   : ["GROUP_suspicious"],
        "resource"  : ["vold"],
        "action"    : ["localsocket"]},
    "condition"     : ["cmd=socket_connect"],
    "effect"        : "deny" }
```



```
shell@localhost /data/local/tmp $ id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009
(mount),1011(adb),1015(sdcard_rw),3001(net_bt_admin),3002(net_bt),3003(inet)
shell@localhost /data/local/tmp $ ./zergRush
./zergRush

[**] Zerg rush - Android 2.2/2.3 local root
[**] (C) 2011 Revolutionary. All rights reserved.

[**] Parts of code from Gingerbreak, (C) 2010-2011 The Android Exploid Crew.

[+] Found a GingerBread ! 0x00000118
[*] Scooting ...
[-] Error creating Nydus: Operation not permitted
shell@localhost /data/local/tmp $
```

Figure 4.5: ZergRush Fails to Exploit Vold

We tested the exploits using (1) samples of GingerMaster, and (2) the shell of Android Debugging Bridge (ADB). Our experimental results showed that OSMon can successfully intercept and prevent such exploits. Figure 4.5 depicts that ZergRush failed because OSMon denied its attempt to crash `vold`.

We further examined a list of known root exploits [2] . We analyzed their exploited vulnerabilities and source code to verify the feasibility of mitigating them with TripleMon. In general, we found that most exploits take advantage of certain system resources that general applications would not write into, such as `/data/data/recovery/log` and `/dev/graphics/fb0`. Indeed, Android applications are expected to access system resources (*e.g.*, device nodes) indirectly via Android APIs. Therefore, OSMon is able to revoke the unnecessary capabilities and prevent such exploits. Note that OSMon does not prevent exploits that use the necessary capabilities of applications. For example, libperf_event exploits the kernel using a crafted system call [15].

[2]`https://github.com/droidsec/droidsec.github.io/wiki/Vuln-Exploit-List`

Decision Reconciliation

Next we evaluated how the decision managers helped coordinate the reference monitors. An Android application may read/write the SMS storage with a two-step method: acquiring a handle to the SMS database via ICC, and then accessing the database file.

We defined a policy set to mediate applications' accesses on SMS. The policy rules in the same policy set protect the same resource (SMS) that corresponds to the objects at multiple layers. As each access involves multiple steps, we define a policy for each step. The decisions made by two policies were aggregated, and the final decision was then made by leveraging the `deny-overrides` strategy. Our experimental results showed that the access was finally denied because the second policy evaluated to deny which overrode the first policy.

```
PolicySet_SMS_ReadWrite: {
    "SMS_ReadWrite_1": {
        "type"  : "ICC",
        "target": {
            "subject"   : ["GROUP_suspicious"],
            "resource"  : ["com.android.providers.telephony"],
            "action"    : ["ContentProvider"]},
            "condition"     : ["URI=sms/*"],
            "effect"        : "accept"},
    "SMS_ReadWrite_2": {
        "type"  : "OS",
        "target": {
            "subject"   : ["GROUP_suspicious"],
            "resource"  : ["/data/data/com.android.providers.
                telephony/mmssms.db"],
            "action"    : ["file"]},
```

Table 4.6: Performance Overhead Compared to Related Work

	μ in ms	σ in ms
ICCMon	0.132	1.060
BinderMon	2.392	4.653
FlaskDroid [37]	0.452	4.887
XManDroid [35]	0.532	2.150
TrustDroid [36]	0.170	1.910

```
        "condition"     : ["cmd=dentry_open"],
        "effect"        : "deny"}
}
```

4.3.4 *Performance*

Our implementation of TripleMon imposes imperceptible runtime overhead. Table 4.6 presents the mean execution time μ and the standard decision σ for performing a policy check. ICCMon has its counterparts in the three closely related MAC implementations and it incurs less runtime overhead. BinderMon is a unique component in our design and its performance is incomparable to the related implementations. We did not evaluate OSMon because it is implemented as a standard Linux security module, and the runtime overhead of such modules has been well studied in the related kernel MAC implementations such as SEAndroid and TOMOYO Linux. In addition, the average footprint of TripleMon is negligible compared to the sizes of existing components, as shown in Table 4.7.

4.4 Discussion

The policy generation process using benign and malicious applications is limited by the fact that dynamic random fuzzy testing fundamentally fails to reveal all pos-

Table 4.7: Memory Overhead

File	**Original (KB)**	**TripleMon (KB)**	**Overhead (KB)**
Services.jar	1155.00	1157.82	2.82
LibBinder	1766.54	1782.02	15.48
Kernel (boot.img)	4571.14	4587.52	16.38
Decision manager	N/A	13.71	13.71

sible execution paths. Recent work [64] shows that the coverage can be lower than 40%. Thus, the policy still has a lot of space to improve. For the malware samples used in our evaluation, we observed that their malicious payloads usually executed immediately after the applications started. Therefore, fuzzy testing over a large number of malware samples is helpful for understanding common malicious behaviors to be denied in our policy.

In TripleMon, we manually and statically analyze the Andorid IPC subsystem to identify choke points, *i.e.*, places for inserting authorization hooks. We believe that an automated approach should be explored to comprehensively and systematically identify potential missing choke points. For example, we could introduce dynamic information flow tracking to identify various points that "forward" information flows between different domains. A similar approach has been proposed in [89] to specifically address client-server software. Such approaches could facility the design of reference monitors that can provide better assurance towards complete mediation.

The current implementation of TripleMon is a proof-of-concept prototype so usability analysis and improvement would be another area to be explored. Despite that TripleMon opts for a simple format of policy scheme with relatively rich expressiveness to ease the burden of policy management, it would be helpful to have user-friendly utilities for creating, maintaining and synchronizing policies. We have implemented a simple web-based management interface for TripleMon. We will continue improving its usability in our future work.

4.5 Related Work

Android security mechanisms have attracted significant attention in recent years. A large number of research projects have been conducted for designing and implementing security extensions on Android to tackle a variety of specific attacks.

FlaskDroid [37] is a two-layer MAC framework that provides flexible and fine-grained mandatory access control on both Android's middleware and kernel layers. It bears the most similarity with our framework. While FlaskDroid and TripleMon both opt for a multi-layer architecture to address the respective semantics of each layer, TripleMon utilizes the peculiarities of Android IPC to be generic and efficient. Specifically, TripleMon's middleware MAC interposes the Binder IPC and ICC channels between applications and system services. This design choice allows TripleMon to enforce system-wide access control polices without modifying system services. FlaskDroid's exemplary implementation provides 12 User Space Object Managers to monitor 40 APIs while TripleMon covers 1,448 public and hidden APIs.

Smalley et al. [109] proposed SEAndroid that extends SELinux as kernel-level MAC and adopts a set of middleware extensions to support middleware MAC. Unlike TripleMon, SEAndroid middleware MAC is only responsible for passing middleware contexts to kernel MAC. The underlying kernel MAC, despite that it has limited semantics of other layers, makes decisions for events occurred at the middleware layer. This situation limits SEAndroid middleware MAC in mitigating corresponding attacks in a fine-grained and accurate manner. In contrast, TripleMon provides reference monitoring at different layers to tackle their respective semantics and employs decision reconciliation to address the possible inconsistencies among them.

XManDroid [35] and TrustDroid [36] both are multi-layer security frameworks that adopt TOMOYO Linux as the underlying kernel MAC and a set of middleware extensions for middlware MAC. Their middlware MAC implementations are tailored to their specific problems: XManDroid [35] attempts to mitigate privilege escalation attacks and TrustDroid [36] establishs an isolated domain for business applications. Along these lines, TripleMon is a generic security framework and can adjust to different threat models and security requirements with user-specified access control policies.

QUIRE [48] enables provenance in Android IPC by propagating verifiable signatures along IPC chains. The signature provides context of the sender application so that a recipient can authenticate the origin of the data they received indirectly. However QUIRE requires that applications must be modified to support QUIRE-style IPC, which is infeasible for most applications whose source code is not available to general users. In contract to QUIRE, TripleMon works with unmodified applications to maintain the compatibility of existing applications.

IPC Inspection [60] is a security framework for tackling permission re-delegation attacks. IPC Inspection reduces the privileges of a recipient application to the intersection of permissions of applications along the IPC chain. However, automatically restricting permissions for collaborative applications is not a decent solution in some cases and may lead to usability loss. Our work employs a policy-based approach and users can specify a group of her trusted applications which can collaborate with each other without restrictions.

TISSA [129] is a policy-driven security extension that protects user's private data. TISSA implements a privacy mode where access to private data can be dynamically and independently controlled. TISSA puts hooks in several privacy-related system services, such as LocationManagerService and TelephonyManagerService. The hooks redirect control flows to a centralized decision maker. Compared to TISSA, TripleMon provides broader coverage of user's data by mediating most IPC channels used by Android applications.

Several recent work addressed fine-grained and context-aware ICC mediation. SAINT [94] is a policy-driven framework that enforces semantically rich policies on ICC at runtime and during installation. Apex [91] provides a similar solution where users can specify runtime constraints for applications. CRePe [44] enables context-aware ICC where environmental constraints such as location and time can be considered for policy enforcement. Although these extensions are not sufficient to cover all existing attacks discussed in Chapter 2 they demonstrate the necessity and value of flexible and fine-grained access control in ICC. Inspired by their work, TripleMon includes a dedicated sub-monitor that addresses the attacks at this layer.

TripleMon requires modifying the Android platform. Although an automated installation kit can reduce the deployment overhead, it cannot entirely eliminate it. Recent research [46, 77, 108, 119] proposed inlined reference monitors by placing hooks inside applications instead of TripleMon's system-centric approach. For example, Aurasium [119] inserts native bootstrapping code into compiled Android applications so as to interpose Libc and mediate Linux system calls. However, such reference monitors are prone to be subverted because they run with the same privileges as the code they are attempting to confine. And Hao et al. [69] demonstrates several potential attacks which may render such reference monitors ineffective or infeasible.

4.6 Summary

In this chapter, we have presented the design and implementation of a multi-layer security framework, TripleMon, that provides flexible and fine-grained access control on Android. TripleMon could mediate multiple Android IPC channels (namely, ICC, Binder IPC and Linux IPC) to prevent prominent attacks that could bypass the existing Android security mechanisms. TripleMon monitors and determines the suspicious behaviors of applications that would lead to appropriate policies for mitigating the attacks. Our experiments showed the common behaviors of Android malware in the wild, and demonstrated the effectiveness and practicality of our approach. The performance measurements also showed that our system has produced only a manageable performance overhead.

Chapter 5

RISK-DRIVEN ASSESSMENT

Helping users understand security and privacy risks of apps is still an ongoing challenge for modern mobile platforms. In this chapter, we propose an risk-driven assessment approach to cope with such a challenge and present a continuous and automated risk assessment framework called RiskMon that uses machine-learned ranking to assess risks incurred by users' installed apps. The preliminary results are published [78, 79].

5.1 Problem Statement

Primarily, Android relies on permissions to help users understand the security and privacy risks of apps. An app must request permissions to be allowed to access sensitive resources. In other words, it is mandatory for Android apps to present its expected behaviors to users. Even though permissions outline the resources that an app attempts to access, they do not provide fine-grained information about how such resources will be used. Suppose a user installs an app and allows it to access her location information. It is hard for her to determine whether the app accesses her locations on her demand or periodically without asking for her explicit consent. Therefore, it is imperative to continuously monitor the installed apps so that a user could be informed when rogue apps abuse her sensitive information. Previous work has proposed real-time monitoring to reveal potential misbehaviors of third-party apps [52, 76, 103, 121]. While these techniques partially provide valuable insights into a user's installed apps, it is still critical to answer the following challenge: *are the behaviors in mobile apps necessarily inappropriate*?

To answer this question, it is an end-user's responsibility to conduct risk assessment and make decisions based on her disposition and perception. Risk assessment is not a trivial task because it requires the user to digest diverse contextual and technical information. In addition, the user needs to apprehend *expected behaviors* of apps under different contexts prior to addressing her risk assessment baseline. However, it is impractical for the normal users to distill such a baseline. Instead, it is essential to develop an automated approach to continuously monitor apps and effectively alert users upon security and privacy violations.

Previous research concerning apps' behaviors specifies a set of risk assessment heuristics tailored to their specific problems. For example, TaintDroid [52] considers a case in which sensitive data is transmitted over the network. DroidRanger [128] and RiskRanker [67] assume that dynamically loaded code is a potential sign of malware. While these techniques provide valuable insights about runtime behaviors of mobile apps, they do not justify the appropriateness of the revealed behaviors. We argue that meta information can provide the necessary operational contexts that justify runtime behaviors for risk assessment. For example, a location-based app has good reasons to upload a user's locations for discovering nearby restaurants. In contrast, it does not make sense for a video player to use the locations and such behaviors should be considered as more risky.

Finally, we need to consider how users participate in risk assessment. Different users would have disparate security requirements. Thus, we should allow users to specify their preferences in terms of accessing their own sensitive information. Moreover, normal users do not possess the necessary technical knowledge for assessing apps' runtime behaviors and interpret numerical risk scores. Therefore, it is imperative to automate risk assessment in a way that requires less sophistication and intervention.

5.2 Risk Assessment of Android Applications

In this section, we describe our proposed risk assessment framework, called RiskMon, that lowers the required intervention and sophistication in risk assessment of mobile apps. IT risk assessment guidelines, such as NIST SP 800-30 [111] and CERT OCTAVE [26], provide a foundation for the development of effective risk management processes. They illustrate comprehensive methodologies that enable organizations to understand, assess and address their information risks. While these guidelines deal with the infrastructure and organizational risks by security experts, our framework attempts to adapt and automate the sophisticated risk assessment tasks for general users.

An underlying assumption of RiskMon is that a user's trusted applications could define her expected appropriate behaviors. Recent empirical analysis showed that applications of similar categories normally request a similar set of permissions [34], implying similar core functionalities. Hence, each of the user's trusted applications can be used as a reference point of appropriate behaviors for applications of similar categories. For example, Netflix application is under "Entertainment" category, and Pandora's Internet Radio application is under "Music & Audio" category. Even though they are not in the same category, each application similarly uses one of core functionalities such as the streaming service of personalized media contents from remote servers. If a user trusts Netflix application, it implicitly affirms that Pandora application may also incur commensurate risks caused by Netflix application. Thus, using Netflix application as a reference point, the deviation or "distance" of runtime behaviors between Netflix and Pandora applications indicates Pandora's additional inherent risks.

We now summarize the design goals as follows:

Continuous and fine-grained behavior monitoring: Applications access sensitive resources by calling APIs to communicate with each other and system services. To ensure continuous monitoring on API calls, RiskMon interposes Binder IPC on a user's device. The risks incurred by API calls are determined by the caller, the callee, and the data. To capture such information, RiskMon opts for a fine-grained scheme to capture various intelligence about applications. This provides a well-founded base for measuring the "distance" between two API calls in the space of runtime behaviors.

Simplified security requirement communication: It is a challenging task for users to specify security requirements for security tools. To tackle this problem, RiskMon adopts a simple heuristic that allows users to communicate security requirements through their coarse expectations. Although this reduces the burden on the user, we cannot entirely eliminate it. We note that acquiring a user's expectations is necessary since each user has diverse preferences on the same application. For instance, all users of Facebook application may have disparate expectations for controlling their location and camera utilities.

Intuitive risk representation: The way in which risk is presented significantly influences a user's perception and decision upon risky applications. A counterexample would be standalone risk scores, such as a risk indicator saying "Facebook incurs 90 units of risk" without proper explanation. As Peng et al. noted in [99], "it is more effective to present comparative risk information". Inspired by their approach, RiskMon presents a ranking of applications so that a user can compare the potential loss of using an application with other applications. In addition, the user can view the risk composition of an application for supporting evidences.

Iterative risk management: Risk assessment is an ongoing iterative process. As applications get upgraded and bring more functionalities, they introduce new risks that should be measured. To this end, the risk assessment baseline should evolve to continuously monitor installed applications and update the risk assessment baseline periodically. Moreover, users need to provide their feedbacks to RiskMon by adding or revising their security requirements.

Figure 5.1 depicts the proposed architecture of RiskMon. RiskMon consists of three components: an app intelligence aggregator, a baseline learner, and a risk meter. The application intelligence aggregator compiles a dataset from API traces collected on a user's device and meta information crawled from application markets. API traces cover an application's interactions with other parts of the system via API calls and callbacks. To complement API traces with contextual information, RiskMon uses meta information on application markets such as ratings, number of downloads and category which provide a quantitative representation of applications' reputation and intended core functionalities. The baseline learner combines a user's coarse expectations and aggregated intelligence of her trusted applications to generate a training set. Afterwards, the baseline learner applies a machine-learned ranking algorithm to learn a risk assessment baseline. Then the risk meter measures how much an application's behaviors deviate from the baseline. Using the deviation to provide risk information, the risk meter ranks a user's installed applications by their cumulative risks and presents the ranking to the user intuitively.

5.2.1 Application Intelligence Aggregator

This component aggregates intelligences about a user's installed applications, including their runtime behaviors and contextual information. As RiskMon monitors runtime behaviors by interposing Binder IPC, we propose a set of features for API traces tailored to the peculiarity of Binder. Also, we seek contextual information from application markets and propose corresponding features to represent and characterize them. The proposed features build a space of application intelligences and enables

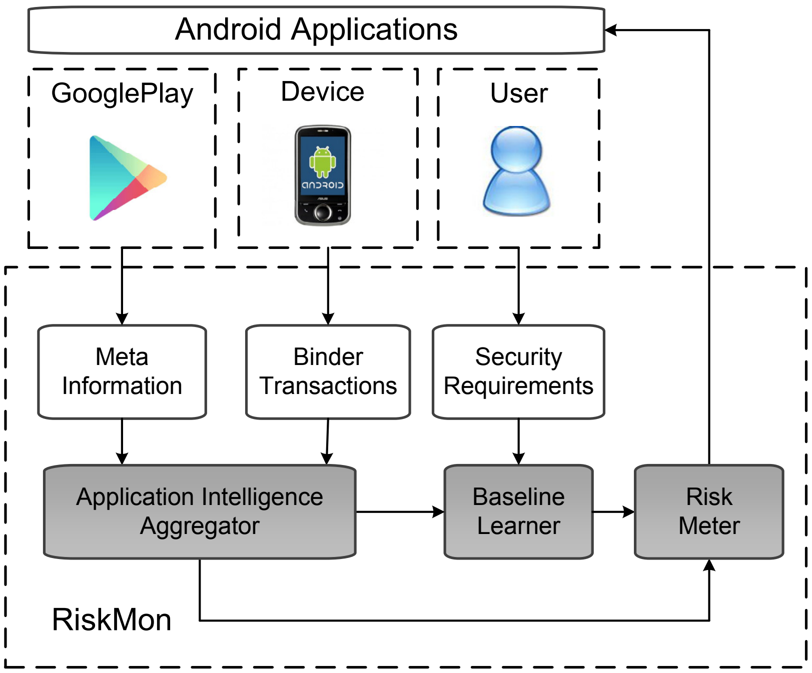


Figure 5.1: Proposed Risk Assessment Framework

subsequent baseline generation and risk measurement. Unless explicitly specified, all features are normalized to [0,1] so that each of them contributes proportionally.

Features for API Traces

Android applications frequently use APIs to interact with system services. Considering that using most APIs does not require any permission, we assume that resources protected by at least one permission are a user's assets.

We are interested in runtime behaviors, i.e. Binder transactions, that are used by APIs to reach the assets. However, APIs do not carry information about Binder transactions. To bridge this gap, we adopt existing work [29, 58] to provide mappings from permissions to APIs. Meanwhile, we analyzed the interface definitions of Android system services and core libraries to generate a mapping from APIs to Binder transactions. As a result, we extracted 1,003 permission-protected APIs, of which each corresponds to a type of Binder transactions. Each type of Binder transaction is identified by the corresponding system service, direction of control flow, and a command code unique to the service. For example, an API named `requestLocationUpdate` is identified as Binder IPC transaction `(LocationManager, callback, 1)`.

We attempt to represent a Binder transaction with its internal properties and contents. For a specific Binder transaction between an application and a system service, we are interested in its type so as to identify the corresponding asset. Also we need to know the direction of control flow for determining who initiates the transaction. As users trust the system services more than applications, RiskMon should differentiate Binder transactions initiated by applications and system services. Thus, internal properties are represented with the following features:

- **Type of Binder transaction**: 1,003 boolean features as a bit array, where one bit is set to 1 for the corresponding transaction type and others are 0; and
- **Direction of control flow**: another boolean feature: 0 for transactions initiated from applications (API calls), 1 for transactions initiated from system services (API callbacks).

Note that we use 1,003 boolean features to represent the type of Binder transactions instead of using one integer value. This is because Binder transactions are independent from each other, and the Binder command codes are simply nominal values. By using the array of 1,003 boolean values, the distances between any two Binder transaction types are set to the same value, which is important for our learning algorithm (Section 8).

In terms of contents, parcels in Binder transactions are unstructured and highly optimized, and it is hard to restore the original data objects without implementation details of the sender and recipient. Therefore, we use length as one representative feature of parcel. A motivating example is accesses on contacts. From the length of a parcel we can infer whether an application is reading a single entry or dumping the entire contacts database. Thus, we propose the following two features for parcels:

- **Length of received parcel**: length of the parcel received by an application in bytes; and
- **Length of sent parcel**: length of the parcel sent by an application in bytes.

Features for Meta Information

Although meta information on application markets cannot describe applications' run-time behaviors, it is still viable to use such information as contextual properties that capture users' and developers' opinions and complement runtime behavior information.

In terms of representing the opinions of users, we use the following features in correspondence with their counterparts of meta information on application markets:

- **Number of installs**: a range of total number of installs since the first release [1] . We use logarithmic value of the lower bound, i.e., *log(1+lower bound of #installs)*;
- **Number of reviews**: a number of reviews written by unique users. We use the logarithmic value, i.e. *log(1+#reviews)*; and
- **Rating score**: a number indicating the user-rated quality of the application ranged from 1.0 to 5.0.

These three features capture an application's popularity and reputation. The first two features are similar to number of views and comments in online social networks. Recent studies [116] demonstrated that online social networks and crowd-sourcing

[1]Number of installs is specified with exponentially increasing ranges: 1+, 5+, . . . , 1K+, 5K+, . . . , 1M+, 5M+.

systems expose a long-tailed distribution. Therefore, we assume they follow the same distribution and use the logarithmic values.

We emphasize that we do not attempt to extract risk signals from these features. Instead, we adopt these features to capture the underlying patterns of a user's trusted applications as specified by the user and apply the patterns for the subsequent risk assessment.

Next, we propose a feature to capture the developer's opinion:

- **Category:** a tuple of two numerical values normalized to [-0.5, 0.5].

Google Play uses an application's category to describe its core functionalities (e.g. "Communication"). As of this writing, Google Play provides 27 category types. We choose Self-Organizing Map (SOM) to give a 2-dimension representation of categories. Barrera et al. [34] demonstrated that SOM can produce a 2-dimensional, discretized representation of permissions requested by different categories of Android applications. Categories in which applications request similar permissions are clustered together. Therefore we use the x and y coordinates in the map to represent categories. Figure 5.2 depicts the coordinates of 13 categories as an example. It is clear to see that some categories bear underlying similarities, such as "Entertainment", "Media and Video" and "Music and Audio" in the center of the figure [2] .

Clearly an unscrupulous developer can claim an irrelevant category to disguise an application's intended core functionalities. However, a user can easily notice the inconsistencies and remove such applications. In addition, falsifying an application's meta information violates the terms of application market's developer policies and may lead to immediate takedown.

[2]For more details on SOM, please refer to [34].

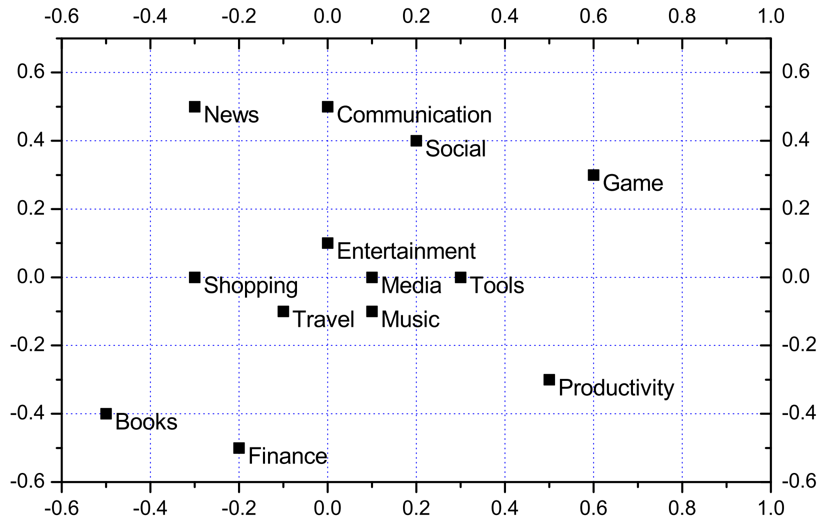


Figure 5.2: SOM Representation of 13 Categories

Finally, based on the scheme defined by these features, the application intelligence aggregator generates a dataset consisted of feature vectors extracted from API traces and meta information of each installed application.

5.2.2 Baseline Learner

The baseline learner is the core module of RiskMon. It takes two types of inputs, which are a user's expectations and feature vectors extracted by the application intelligence aggregator. Then the baseline learner generates a risk assessment baseline which is represented as a predictive model.

Acquiring Security Requirements

It is challenging for most users to express their security requirements accurately. We aim to find an approach that could be mostly acceptable by users. Krosnick and Alwin's dual path model [84] demonstrated that a *satisficing* user would rely on salient cues to make a decision. Based on this model we develop a simple heuristic:

For a specific application, accesses on resources that are more irrelevant of a user's expected core functionalities incur more risks.

This heuristic captures a user's expectations as security requirements by risk aversion, which implies the reluctance of a user to use a functionality with an unknown marginal utility [101]. For example, a user may consider that, microphone is necessary to a VoIP application such as Skype. But location seems not because she does not understand the underlying correlation between disclosing her location and making a phone call. Thus, microphone is more relevant and less risky than location in her perception.

Base on this, the risk learner asks a user to specify a relevancy level for each permission group requested by her trusted applications. We choose permission groups to represent resources because it is much easier for general users to learn 20+ permission groups than 140+ permissions. And recent usability studies demonstrated the ineffectiveness of permissions due to limited comprehension [43, 62]. Although users tend to overestimate the scope and risk of permission groups, they are more intuitive and reduce warning fatigue [62].

The process for users to communicate their security requirements with RiskMon is similar to a short questionnaire. Each permission group requested by a user's trusted applications corresponds to a five-point Likert item. The user specifies the level of relevancy on a symmetric bipolar scale, namely *relevant*, *probably relevant*, *neutral*, *probably irrelevant* or *irrelevant*. Figure 5.3 shows an example of relevancy of permission groups for Facebook and Skype. Permission groups are represented by self-descriptive icons, which are identical to those shown in Android Settings. `CAMERA` preceding `LOCATION` for Facebook is possibly due to the user's preference to photo sharing compared to check-ins.

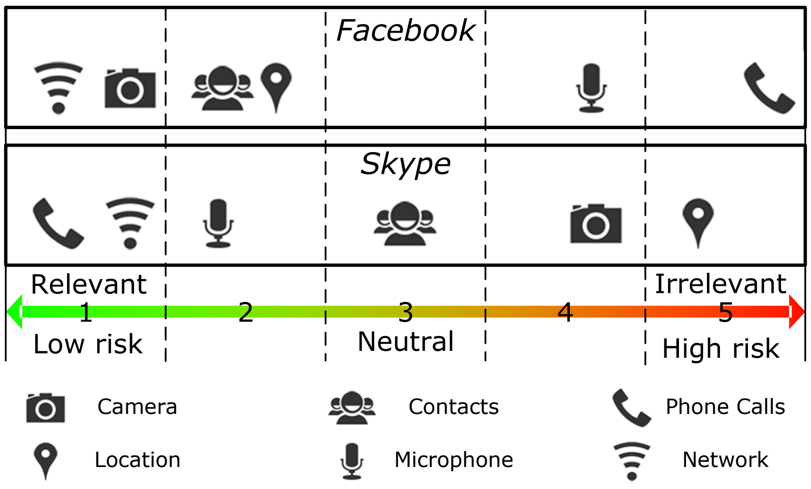


Figure 5.3: An Example of Specifying Relevancy for Permission Groups

Note that the relevancy levels specified by users are *subjective*. With that said, users' biased perception of applications and resources may affect their specified relevancy levels. From our user study, a user told us that `PHONE_CALLS` is relevant to Google Maps because he tapped a phone number shown in Google Map and then the dialer appeared. Although the dialer rather than Google Map has the capability to make phone calls, the baseline learner considers it as the security requirements for inter-application communication.

We next formalize the problem of acquiring security requirements. $PG = \{pg_1, pg_2, \cdots, pg_m\}$ is a set of permission groups available in a mobile operating system. $A = \{a_1, a_2, \cdots, a_n\}$ is a set of a user's installed applications. TA is a set of a user's trusted and installed applications and $TA \subseteq A$. $RequestedPG : A \rightarrow 2^{PG}$ is a function that maps an application to its requested permission groups. A user's security requirement Req is a mapping $Req : TA \times PG \rightarrow R$. $R = \{1, 2, 3, 4, 5\}$ is a set of relevancy levels, where a larger value indicates higher relevancy and less risk and vice versa.

Compiling Training Set

Next we describe how the baseline learner compiles a training set from the aggregated application intelligences and user-specified relevancy levels. For brevity, we apply the relevancy levels onto the feature vectors generated by the application intelligence aggregator to generate a set of vectors annotated with relevancy levels.

To bridge the gap between permission groups and feature vectors, we extract mappings of permission groups and permissions from the source code of Android. Meanwhile, existing work has provided mappings between permissions and APIs [29, 58]. Therefore, we can assign the relevancy level on feature vectors because each vector represents an API call or callback.

We formalize the problem of compiling a training set as follows. Algorithm 8 illustrates the process to compile the training set T.

- X is a space of features as defined by the scheme discussed in Section 5.2.1, $X = \{\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_l\}$, $X \in \mathbb{R}^i$, where i denotes the number of features;
- $DS = \{D_{a1}, D_{a2}, \cdots, D_{am}\}$ is a collection of sets of feature vectors, where $D_{aj} \subseteq X$ and D_{aj} corresponds to an application a_j;
- $Apd : A \times PG \to DS$ is a function that maps an application and one of its requested permission groups to a set of feature vectors; and
- $T = \{(\vec{x}_1, r_1), (\vec{x}_2, r_2), \cdots, (\vec{x}_n, r_n)\}$ is a training set consisted of annotated vectors, $r_k \in R$, $\vec{x}_k \in X$.

Generating Risk Assessment Baseline

Ranking Support Vector Machine (RSVM) [71, 81] is a pair-wise ranking method. Generally it utilizes a regular Support Vector Machine (SVM) solver to classify the order of *pairs of objects*. Next we explain how we apply RSVM to learn a risk assessment baseline.

Algorithm 1: Compiling Training Set

```
  Data: $DS$, $TA$, $Req$
  Result: $T$
1 $T \leftarrow \emptyset$;
2 for $a \in TA$ do
3     $pg \leftarrow RequestedPG(a)$; $r \leftarrow Req(a, pg)$; $D \leftarrow Apd(a, pg)$;
4     for $\vec{x} \in D$ do
5         add $(\vec{x}, r)$ to T;
6     end
7 end
8 return $T$
```

We assume that a set of ranking functions $f \in F$ exists and satisfies the following:

$$\vec{x}_i \prec \vec{x}_j \iff f(\vec{x}_i) < f(\vec{x}_j), \tag{5.1}$$

where $\prec$ denotes a preferential relationship of risks.

In the simplest form of RSVM, we assume that f is a linear function:

$$f_{\vec{w}}(\vec{x}) = \langle \vec{w}, \vec{x} \rangle, \tag{5.2}$$

where $\vec{w}$ is a weight vector, and $\langle \cdot, \cdot \rangle$ denotes inner product.

Combing (5.1) and (5.2), we have the following:

$$\vec{x}_i \prec \vec{x}_j \iff \langle \vec{w}, \vec{x}_i - \vec{x}_j \rangle < 0, \tag{5.3}$$

Note that $\vec{x}_i - \vec{x}_j$ is a new vector that expresses the relation $\vec{x}_i \prec \vec{x}_j$ between $\vec{x}_i$ and $\vec{x}_j$. Given the training set T, we create a new training set T' by assigning either a positive label $z = +1$ or a negative label $z = -1$ to each pair $(\vec{x}_i, \vec{x}_j)$.

$$(\vec{x}_i, \vec{x}_j) : z_{i,j} = \begin{cases} +1 & \text{if } r_i > r_j \\ -1 & \text{if } r_i < r_j \end{cases} \tag{5.4}$$

$$\forall (\vec{x}_i, r_i), (\vec{x}_j, r_j) \in T$$

In order to select a ranking function f that fits the training set T', we construct the SVM model to solve the following quadratic optimization problem:

$$\begin{aligned} \underset{\vec{w}}{\text{minimize}} \quad & \frac{1}{2}\vec{w} \cdot \vec{w} + C \sum \xi_{i,j} \\ \text{subject to} \quad & \forall (\vec{x}_i, \vec{x}_j) \in T' : z_{i,j} \langle \vec{w}, \vec{x}_i - \vec{x}_j \rangle \geq 1 - \xi_{i,j} \\ & \forall i \forall j : \xi_{i,j} > 0 \end{aligned} \tag{5.5}$$

Denoting $\vec{w}^*$ as the weight vector generated by solving (5.5), we define the risk scoring function $f_{\vec{w}^*}$, for assigning risk scores to the feature vectors in the application intelligence dataset:

$$f_{\vec{w}^*} = \langle \vec{w}^*, \vec{x} \rangle \tag{5.6}$$

For any $\vec{x} \in X$, the risk scoring function measures its projection onto $\vec{w}^*$, or the distance to a hyperplane whose normal vector is $\vec{w}^*$. Thus, the hyperplane is indeed the risk assessment baseline.

5.2.3 Risk Meter

Risk meter measures the risks incurred by each installed application including those are trusted by the user. Note that (5.6) gives a signed distance. We use the absolute value to represent the deviation and risk. The risks incurred by an application a_i are the cumulative risks of its runtime behaviors:

$$\sum_{\vec{x} \in D_{ai}} |f_{\vec{w}^*}(\vec{x})| \tag{5.7}$$

Another goal of the risk meter is to provide supporting evidences to end-users. To this end, it presents the measured risks at three levels of granularities.

Application: In the simplest form, the risk meter presents a ranking of installed applications by their risks as a bar chart. The X axis indicates the applications and the Y axis indicates the risks. A user can trust an application if it is less risky than her trusted ones. In contrast, an application that is significantly risky can also draw a user's attention. Note that the risk meter does not provide any technical explanation at this level.

Permission group: The ranking of applications may seem unconvincing sometimes for users. In such a case, the risk meter can provide risk composition by permission groups which is represented as a pie chart. The pie chart intuitively reveals the proportion of the risks incurred by the core functionalities of an application. As users have basic knowledge of permission groups when they specify security requirements, they should be able to interpret the risk composition correctly.

API calls and callbacks: The evidences presented at this level are intended for experienced security analysts who are familiar with the security mechanisms under the hood of Android. This is the raw data generated by the risk scoring function. An analyst can inspect values of features to reconstruct the semantic view of runtime behaviors.

Moreover, RiskMon allows a user to establish and revise her security requirements iteratively. RiskMon may generate biased or unconvincing evidences as a user may not have clear and accurate security requirements at the very beginning of using RiskMon. Thus, a user can provide her feedback by adjusting her security requirements and/or adding more trusted applications. RiskMon also periodically updates the security assessment baseline for observed new runtime behaviors. All of these enable RiskMon to approximate an optimum risk assessment baseline to help users make better decisions.

5.3 Automated Risk Mitigation

Based on the proposed risk assessment framework, we move one step further to address risk mitigation. Specifically, we propose an automated decision process that assists users to conveniently identify and revoke risky permissions from installed applications.

A typical permission framework, just like common access control systems, involves decision processes that grant and revoke permissions. While permission granting has been widely adopted in modern mobile platforms, permission revocation has not received a commensurate popularity. For example, iOS users could not deny accesses to their personal information until iOS 6. Google introduced App Ops as an experimental privacy control framework in Android 4.3, but later disabled its management interface in Android 4.4.2 [49].

Permission revocation is necessary because it enables complete and flexible control over granted capabilities. To this end, recent work has proposed enhanced middleware mandatory access control (MMAC) frameworks to support rule-driven permission revocation on Android [35, 38, 91, 109, 129]. An obvious limitation of such frameworks lies in the definition and maintenance of the rules [54], which place non-negligible burden on general users. To say the least, it remains an open question whether users can accurately cherry-pick the risky permissions that should indeed be revoked.

Intuitively, RiskMon could provide the necessary evidences to support a permission revocation decision process. However, Android by default only allows users to mitigate unnecessary risks is removing risky applications. Such an arbitrary approach may disrupt user experiences. For example, grey applications (*e.g.*, ad-supported games) are likely to request excessive permissions for harvesting user information. Revoking all the granted permissions (*i.e.*, removing application) seems unnecessary because some permissions are not major sources of risks and they may support functionalities that a user needs. Our goal is to selectively revoke risky permissions and mitigate future risks to a user's expected level. Therefore, those grey applications might still retain necessary functionalities and users could stay protected from privacy-infringing code.

We identify three key challenges in bridging the gap between risk assessment and risk mitigation: (1) selecting reference applications; (2) estimating risk budgets; and (3) enforcing decisions with minimal user intervention. Reference applications implicitly provide a user's expected runtime behaviors and upper bounds of acceptable risks. Risk budgets quantitatively determine decision thresholds that line up with the user's risk mitigation strategies. Moreover, we need to minimize user intervention in decision enforcement, because general users would be incapable and reluctant to create and manage security policies. We next describe how we address these challenges.

5.3.1 Selecting Reference Applications

As we previously assumed, a user's trusted applications define her expected appropriate behaviors for similar applications. To select a set of reference applications for a target application, we prefer trusted applications that are under the *same* or *close* categories because their core functionalities tend to be similar. Therefore, we assign coordinates to all the installed applications according to their categories in the category SOM. Then, we select the reference applications by computing a set of k-nearest trusted applications based on their Euclidean distances.

The best choice of k depends on the category SOM and the number of the trusted applications. Here we adopt a conservative approach to avoid over-generalization that could lead to over-estimation of risk budgets. First, we start from $k \leq \lfloor log_2|A_T| \rfloor$. Meanwhile, we need to filter this set by removing applications that are not close

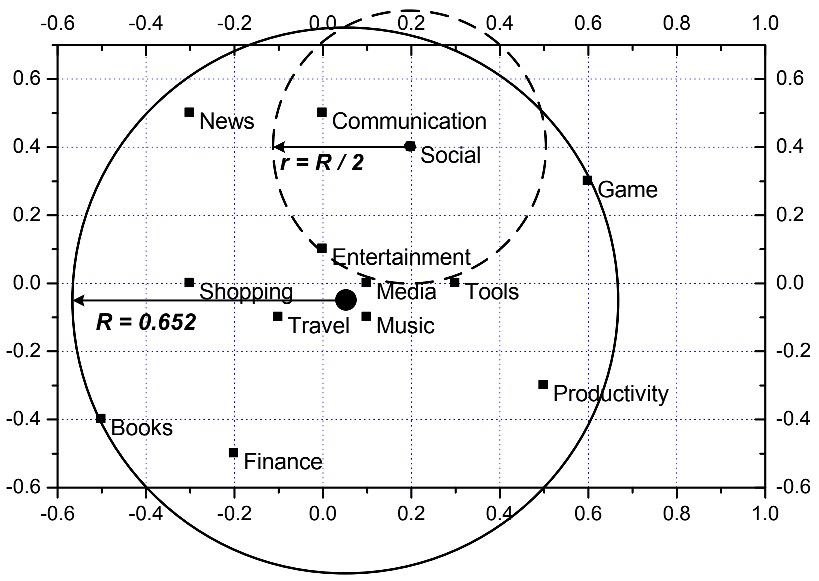


Figure 5.4: An Example of Selecting Reference Applications from Close Categories

enough to the target application. To quantitatively define "close", we compute the smallest enclosing circle of the category SOM and its radius R, and choose $R/2$ as the threshold of close categories. In summary, a target application a's reference application set A_{Ra} is the intersection of the following sets:

1. $\lfloor log_2|A_T|\rfloor$-nearest trusted applications; and
2. the trust applications whose Euclidean distance from a is no larger than r, where $r = R/2$.

Figure 5.4 demonstrates an example of selecting reference applications for a social application. The result is no more than $\lfloor log_2|A_T|\rfloor$ applications under the "Social", "Communication", and/or "Entertainment" categories.

Automated risk mitigation is also limited by the same problem of insufficient trusted applications as automated risk assessment. A_{Ra} could be empty because A_T does not cover sufficient categories. In such a case, reselecting reference applications is scheduled after a user adds trusted applications and improves coverage.

5.3.2 Estimating Risk Budgets

Risk budgets define decision thresholds used in our automated decision process. Our goal is to derive a risk budget for each permission of a target application from its reference applications.

We next formalize the problem of estimating risk budgets for a target application a as follows:

- $P = \{p_1, p_2, \cdots, p_n\}$ is a set of permissions available in a mobile operating system;
- $UsedP : A \rightarrow 2^P$ is a function that maps an application to a set of permissions whose usage patterns have been observed by RiskMon;
- $PAR : P \times A \rightarrow \mathbb{R}$ is a function that maps a granted permission of an application to its measured risk score; and
- $BI_a = \bigcup_{ta \in A_{Ra}} UsedP(ta)$ is a set of permissions that are the budget items for an application a.

We then introduce the following *budget estimation functions* to support different risk mitigation strategies, where $p \in UsedP(\text{a})$, $a \in A$, $a \notin A_T$, $A_{Ra} \subset A_T$:

$$\begin{aligned}
Strict_a(p) \quad &= \begin{cases} \min\limits_{ta \in A_{Ra}} PAR(p, ta) & \text{if } p \in BI_a \\ 0 & \text{if } p \notin BI_a \end{cases} \\
Average_a(p) &= \begin{cases} \mathop{\mathbf{avg}}\limits_{ta \in A_{Ra}} PAR(p, ta) & \text{if } p \in BI_a \\ 0 & \text{if } p \notin BI_a \end{cases} \\
Relaxed_a(p) &= \begin{cases} \max\limits_{ta \in A_{Ra}} PAR(p, ta) & \text{if } p \in BI_a \\ \mathop{\mathbf{avg}}\limits_{ta \in A_T} PAR(p, ta) & \text{if } p \notin BI_a \end{cases}
\end{aligned} \tag{5.8}$$

The `strict` function prefers the most privacy-preserving practices of the reference applications. The `average` function attempts to reduce the risks below the average practices. For the permissions not among the budget items, the `strict` and `average` functions both opt for a zero tolerance strategy. In contrast, the `relaxed` function allows such permissions but their incurred risks should not exceed the average of all the trusted applications.

5.3.3 Generating and Enforcing Decisions

To generate a decision for a permission p of an application a, we compute its cumulative risks as $Risk_a(p)$ and apply a user-specified budget estimation function, for example:

$$Decision(a,p) = \begin{cases} \text{Keep} & \text{if } Risk_a(p) \leq Strict_a(p) \\ \text{Revoke} & \text{if } Risk_a(p) > Strict_a(p) \end{cases} \tag{5.9}$$

Note that an important criterion of our decision process is *revoking by observed behaviors* [3].

Managing security policies for complex information systems has been a challenging task. It is even harder for dynamic systems such as the Android middleware, whose security policies have to confine various applications that rapidly update themselves. Enforcing security decisions for such systems would be unrealistic for general users because it consumes much user attention and leads to habituation [61]. This partially implies why Android community has been careful with integrating user-oriented and generic permission revocation [49].

We introduce automated policy generation to address this challenge. Specifically, automated permission revocation and policy generation are activated after (1) a user

[3]Intuitively, dormant permissions do not incur any risks so we choose not to revoke them because we have no observed evidence to prove that such permissions will be abused.

installs or updates a new application; (2) a user updates her risk assessment baseline; or (3) a pre-defined time period. Note that we do not attempt to implement our own policy enforcement mechanisms. Instead, our framework could be easily adapted to support new middleware MAC frameworks with an intuitive policy translation module.

5.4 Implementation and Evaluation

In this section we first discuss a proof-of-concept implementation of RiskMon. Then, we present the results of our online user study followed by two case studies. We conclude our evaluation with the usability and performance of our system.

5.4.1 Implementation and Experimental Setup

We implemented a proof-of-concept prototype of RiskMon on the Android mobile platform. In terms of continuous monitoring, we implemented a reference monitor for Binder IPC by placing hooks inside the Binder userspace library. The hooks tap into Binder transactions and log the parcels with zlog [4] which is a high-performance logging library. In addition, we implemented automated risk assessment based on SVMLight [5] and its built-in Gaussian radial basis function kernel.

We designed and conducted a user study to evaluate the practicality and usability of RiskMon. We hand-picked 10 applications (Table 5.2) that were mostly downloaded from Google Play in their respective categories. We assumed that all the participants trust them. Then we used participants' security requirements for the 10 applications and their application intelligences to generate the baselines. We also randomly selected 4 target applications from the Top Charts of Google Play to calculate

[4]`https://github.com/HardySimpson/zlog`

[5]`http://svmlight.joachims.org/`

their risks based on the generated baselines, including: *a*) CNN App for Android Phones (abbreviated as CNN); *b*) MXPlayer; *c*) Pandora Internet Radio (abbreviated as Pandora); and *d*) Walmart. For both trusted (10) and target (4) applications, we collected their one-day runtime behaviors on a Samsung Galaxy Nexus phone. In addition, we developed a web-based system that acquires a participant's security requirements, feeds them to RiskMon and presents the results calculated by RiskMon to the participant. A participant was first presented with a tutorial page that explains how to specify relevancy levels as her security requirements. Then she was required to set relevance levels for each permission group requested by each trusted application after reading the application's descriptions on Google Play. Afterwards, RiskMon generated a risk assessment baseline for the participant based on her inputs and runtime behaviors of the 10 trusted applications. Then RiskMon applied the baseline on each of the 14 applications, and displayed a bar chart that illustrates a ranking of 14 applications by their measured cumulative risks. Finally, an exit survey was presented to collect the participant's perceived usability of RiskMon. Our study protocol was reviewed by our institution's IRB. And we recruited participants through university mailing lists and Amazon MTurk. 33 users participated in the study and Table 5.1 lists the demographics of them.

5.4.2 Empirical Results

Security Requirements

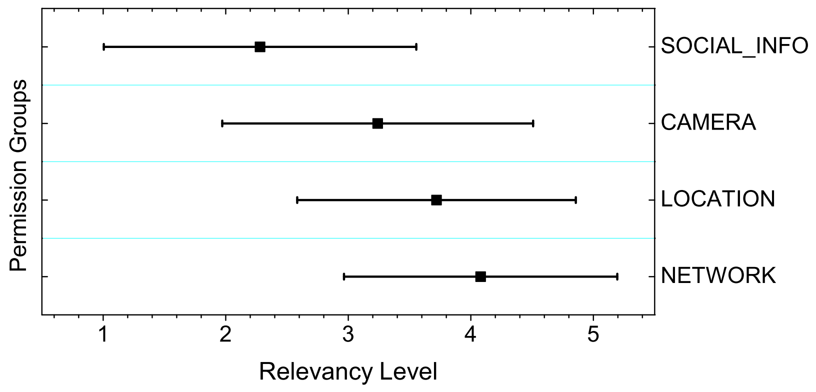
From our user study shown in Table 5.2, we highlight the results of Chase Mobile and Dropbox because they both request some ambiguous permission groups that are hard to justify for users. Figure 5.5 demonstrates the average relevancy levels set by

Table 5.1: Demographics of the Participants

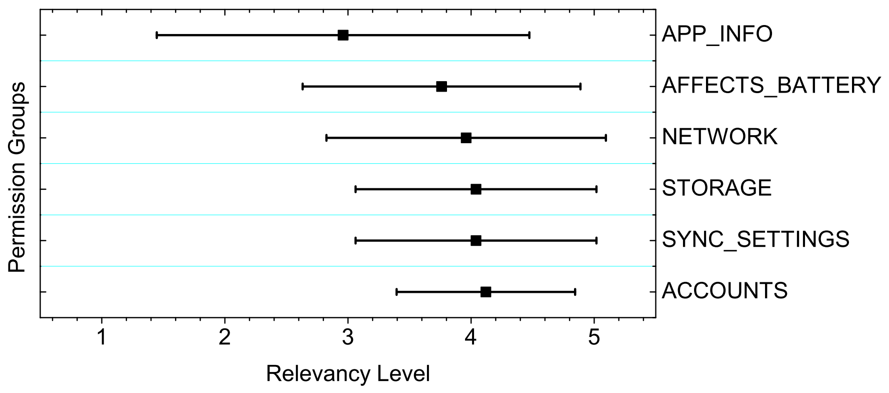
Category		# of users
Gender	Male	29 (87.9%)
	Female	4 (12.1%)
Age	18-24	15 (45.5%)
	25-34	16 (48.5%)
	35-54	2 (6.1%)
Education	Graduated high school or equivalent	3 (9.1%)
	Some college, no degree	6 (18.2%)
	Associate degree	1 (3.0%)
	Bachelor's degree	11 (33.3%)
	Post-graduate degree	12 (36.4%)

Table 5.2: Applications Assumed to be Trusted in the User Study

Application	Category
AmazonMobile	Shopping
BejeweledBlitz	Game
ChaseMobile	Finance
Dictionary.com	Books & Reference
Dropbox	Productivity
Google+	Social
GooglePlayMovies&TV	Media & Video
Hangouts(replacesTalk)	Communication
MoviesbyFlixster	Entertainment
Yelp	Travel & Local



(a) Chase Mobile



(b) Dropbox

Figure 5.5: Average Relevancy Levels Specified by the Participants for Chase Mobile and Dropbox

the participants for each permission group requested by Chase Mobile and Dropbox. The error bars indicate the standard deviation.

Chase Mobile is a banking application with functionalities like depositing a check by taking a picture and locating nearest branches. Apparently `NETWORK` is more relevant than others as participants agree that Chase Mobile needs to access the Internet. Even though Chase Mobile uses `LOCATION` to find nearby bank branches and `CAMERA` to deposit checks, both `LOCATION` and `CAMERA` have lower relevancy levels than `NETWORK`. We believe it is because some participants do not have the experiences of using such functionalities, but the averages are still higher than neutral. We can also observe that SOCIAL_INFO falls below “neutral”, showing participants’ concerns of why Chase Mobile uses such information.

Dropbox is an online file storage and synchronization service. From its results, we identified an interesting permission group, APP_INFO, whose description in Android’s official document is: *group of permissions that are related to the other applications installed on the system*. This authoritative description does not provide any cue of negative impacts, which leads to user confusion as we can see that APP_INFO has the largest standard deviation. STORAGE, SYNC_SETTINGS and ACCOUNTS are all above “probably relevant” possibly due to their self-descriptive names that are semantically close to Dropbox’s core functionalities.

Moreover, we noticed that the participants tend to set higher relevancy levels for self-descriptive permission groups, while they tend to be conservative for other permission groups. We note that this does not affect RiskMon in acquiring a user’s security requirements, because RiskMon captures the precedence of one permission group over another. Thus, the least relevant permission group (e.g. SOCIAL_INFO of Chase Mobile) always gets the highest risk scores for both trusted and distrusted applications.

Application Risk Ranking

Figure 5.6 illustrates the ranking of 14 applications by their average cumulative risk scores as measured by 33 risk assessment baselines generated for the participants. We can see that MXPlayer (2.55) and Walmart (12.72) fall within the trusted applications, while CNN (54.15) and Pandora (69.22) are ranked with highest risk scores.

Note that both Pandora and CNN are renowned applications developed by well-trained developers. Seemingly, they should use sensitive information appropriately. Hence, we verified them by manually dissecting their API traces. We found that

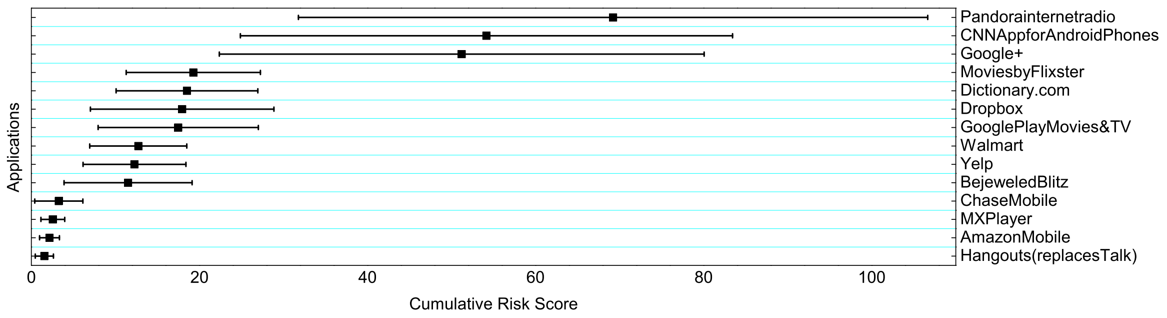


Figure 5.6: Average Cumulative Risk Scores Measured by the Participants' Risk Assessment Baselines

they both stayed in the background and attempted to keep connected to remote servers. To this end, they kept polling ConnectivityManager for a fine-grained state of the current network connection. This is an unexpected practice for both privacy and performance perspectives and the official Android documents suggest developers register `CONNECTIVITY_CHANGE` broadcasts [6] to get connectivity updates accordingly instead of polling. On the contrary, Hangouts incurred almost imperceptible amount of risks, although it has similar requirements for connectivity. Therefore, RiskMon showed that even popular applications might use sensitive information in a way that incurs potential risks for users.

5.4.3 Case Studies

In this section we evaluate the effectiveness of our approach. Note that there is no ground truth of user's expected appropriate behaviors. Thus, we opt for two case studies on two applications, SogouInput and PPS.TV. We specified the relevancy levels for 10 trusted applications and generated a risk assessment baseline. Then, we verified their identified risk composition with manual analysis.

SogouInput is an input method based on the pinyin method of romanization, and PPS.TV is a video streaming application similar to its counterparts such as

[6]`http://developer.android.com/training/monitoring-device-state/connectivity-monitoring.html`

Hulu and Netflix. Both of them are feature-rich, free and have accumulated over 5,000,000 installs on Google Play. We note that PPS.TV and SogouInput request 22 and 29 permissions, respectively. The numbers of requested permissions make them suspicious over-privileged or privacy-infringing applications.

The measured cumulative risk scores are 179.0 for SogouInput and 366.9 for PPS.TV. Table 5.3 demonstrates the risk composition of SogouInput and PPS.TV by their requested permission groups. First, the unusually large portion of `PHONE_CALLS` indicates significant use of capabilities related to making phone calls and reading unique identifiers. We verified the corresponding API traces and revealed that it attempted to read a user's subscriber ID and device ID. Second and more notably, `SOCIAL_INFO` contributed 4.02% of the total risks incurred by SogouInput. We verified the corresponding API traces and found that SogouInput accessed the Contacts app and received a parcel of 384 bytes. Usually an Android application queries the contact application and receives only the entries a user picks, which is several bytes long. On the contrary, SogouInput attempted to dump the whole contacts data repository. Similar to SogouInput, PPS.TV utilized permissions related to `PHONE_CALL`. In addition to reading a user's device ID and subscriber ID, it also registered a callback to receive events of call states. We note that this allows PPS.TV to read the number of incoming calls.

The results leave much room for imagination: how come an input method and a video streaming application need capabilities related to `PHONE_CALLS`, `LOCATION` and `SOCIAL_INFO`? Possibly users get personalized services by disclosing these information. However it comes with a price of privacy. RiskMon highlights the risks so that users can weigh the benefit and relevant cost by themselves.

Table 5.3: Risk Composition by Permission Groups of Applications in Case Studies

Application	Permission Group	Risk Score
SogouInput	LOCATION	5.6 (3.13%)
	NETWORK	104.4 (58.29%)
	PHONE_CALLS	61.8 (34.56%)
	SOCIAL_INFO	7.2 (4.02%)
	Total:	179.0 (100%)
PPS.TV	LOCATION	26.0 (7.09%)
	NETWORK	108.3 (29.52%)
	PHONE_CALLS	232.6 (63.40%)
	Total:	366.9 (100%)

Automated Risk Mitigation

Based on the measured risks of the 6 applications, we further applied our automated risk mitigation approach. In particular, we used Figure 5.2 to guide our selection of reference applications out of 10 trusted applications. Therefore, $r = R/2 = 0.326$ as shown in Figure 5.4 and k was no more than 3. Afterwards, we chose the **`average`** budget estimation function to reduce the incurred risks of the applications that are below the average level of their respective reference applications. Table 5.4 shows the revoked permissions and risk reduction of the assessed applications. In this table, we have denoted the specific reason for each revoked permission. "(O)" indicates that the revoked permission was used by one or more reference applications but exceeded the threshold set by the budget estimation function. "(N)" means that the permission was not used by any of the reference applications. Such permissions were also revoked in our case studies due to the **`average`** function's zero tolerance strategy.

The revoked permissions are lined up with the results as shown in Table 5.3. In particular, `READ_CONTACTS` and `VIBRATE` were revoked from SogouInput because they were used but not among the risk budget items. In contrast, none of permissions related to `LOCATION` was revoked, implying that SogouInput used `LOCATION` in a reasonable and conservative manner. 4 out of 5 revoked permissions of PPS.TV were mitigated due to over-budget, demonstrating its notable tendency of abusing a user's information. Overall, these applications were confined to behave like their respective reference trusted applications.

We enforced the generated decisions through AppOps, and the revoked permissions did not break the core functionalities. However, we can not guarantee that permission revocation does not significantly impair an application's usability, for two reasons. First, our framework does not directly enforce decisions. Graceful enforcement of decisions by access control frameworks is still an open question that is beyond the scope of this chapter. Second, risky permissions are not always excessive. Obviously, core functionalities would break if their abused permissions are revoked. applications instead of using our granular permission revocation mechanism.

The results of the case studies leave room for further analysis. How come an input method and a video streaming application need capabilities related to `PHONE_CALLS`, `LOCATION` and `SOCIAL_INFO`? Why does Walmart need to continuously access users' location? Possibly users could get personalized services through disclosing private information. However, it comes with a price. RiskMon is a necessary step towards highlighting and mitigating the excessive risks.

5.4.4 System Usability

The criteria for usability were split into three areas: *likeability*, *simplicity* and *risk perception*. Likeability is a measure of a user's basic opinion towards automated risk assessment. This identifies whether users would like to accept the proposed mechanism. Simplicity is a measure of how intuitive the concepts and procedures

Table 5.4: Revoked Permissions of Applications in Case Studies

Application	**Revoked Permissions** (O): Over budget (N): Not in budget	**Risk Reduction**
SogouInput	ACCESS_NETWORK_STATE (O) READ_PHONE_STATE (O) READ_CONTACTS (N) VIBRATE (N)	169.5 (94.7%)
PPS.TV	ACCESS_LOCATION (O) ACCESS_NETWORK_STATE (O) ACCESS_WIFI_STATE (O) CHANGE_WIFI_STATE (N) READ_PHONE_STATE (O)	367.0 (99.8%)
Pandora	ACCESS_NETWORK_STATE (O)	130.3 (98.5%)
CNN	ACCESS_LOCATION (N) ACCESS_NETWORK_STATE (O) WAKE_LOCK (N)	128.5 (100.0%)
Walmart	ACCESS_LOCATION (O) ACCESS_NETWORK_STATE (O)	56.6 (100.0%)
MXPlayer		0.0 (0.0%)

are, which is useful in evaluating the burden placed on users. Risk perception is a measure of a user's perceived awareness of risks through risk assessment, which evaluates how users interpret the risks as presented by RiskMon.

After using RiskMon, an exit survey was presented to collect users' perceived usability of RiskMon. In the survey we asked users questions on *likeability* (e.g. "indicate how much you like using your trusted apps to set a baseline"), *simplicity* (e.g. "do you agree that RiskMon requires less mental efforts in risk assessment"), and *risk perception* (e.g. "do you feel the increased awareness of the risks of your installed applications"). Questions were measured with a five-point Likert scale. A

Table 5.5: Usability Evaluation Results

Metric	Average	Lower bound on 95% confidence interval
Likeability	0.811	0.797
Simplicity	0.674	0.645
Risk perception	0.758	0.751

Table 5.6: Microbenchmark Results

Benchmark	Average (s)	Standard Deviation (s)
Feature extraction	8.27	0.07
Baseline generation (10 apps)	289.56	235.88
Risk measurement (per app)	0.55	0.17

higher score indicates a positive opinion or agreement, while a lower score indicates a negative one or disagreement. Then scores were adjusted to [0,1] for numerical analysis.

We analyzed a 95% confidence interval for users' answers. Specifically we are interested in determining the average user's minimum positive opinions. Hence, we looked at the lower bound of the confidence interval. Table 5.5 shows that an average user asserts 79.7% positively on likeability, 64.5% on simplicity and 75.1% on risk perception. The results show usability of RiskMon with the above-average feedback.

5.4.5 System Overhead

To understand the performance overhead of RiskMon, we performed several microbenchmarks. The experiments were performed on a Samsung Galaxy Nexus phone with a 1.2GHz dual-core ARM CPU. The phone runs Android v4.2.2 and RiskMon built on the same version. Table 5.6 shows the average results.

Feature extraction: The application intelligence aggregator extracted feature vectors from the raw API traces of 33,368,458 IPC transactions generated by 14 applications in one day. We measured the CPU-time used by parsing the API traces and generating the feature vectors. The average time is 8.27 seconds, which is acceptable on a resource-constrained mobile device.

Baseline generation: We ran baseline generation based on the input acquired in the online user study. The processing time varies for different participants, while the average time is approximately 289.56 seconds due to the computation complexity of the radial basis function kernel of SVMLight.

Risk measurement: Applying the risk assessment baseline is much faster than baseline generation. We measured the time taken to apply a risk assessment baseline on 14 applications. The average time per application is 0.55 seconds, which is imperceptible and demonstrates the feasibility of repeated risk assessment.

Finally, we anecdotally observed that it took 5-10 minutes for the participants to set relevancy levels for 10 applications. This usability overhead is acceptable compared to the lifetime of a risk assessment baseline.

5.5 Discussion

To capture actual risks incurred by applications used by a user, RiskMon fundamentally requires running them on the user's device. We note that 48.5% of the respondents in our user study claimed that they often test drive applications on their devices. RiskMon itself does not detect or prevent sensitive data from leaving users' devices. We would recommend users use on-device isolation mechanisms (e.g. Samsung KNOX [7]) or data shadowing (e.g. [76]). However, it is far from perfect for running untrusted applications on trusted operating systems.

RiskMon requires users to specify security requirements through permission groups. While most of the frequently requested permission groups are self-descriptive

[7]`http://www.samsung.com/global/business/mobile/solution/security/samsung-knox`

(e.g. `LOCATION` and `CAMERA`), some are ambiguous (e.g. `APP_INFO`) and contain low-level APIs only known to developers. Although we identify permission groups as an appropriate trade-off between granularity and usability, we admit that permission groups are still a partial artifact in representing sensitive resources for users. Note that we choose permission groups only to demonstrate the feasibility of our approach of security requirement communication. As our future work, we plan to develop a systematic and intuitive taxonomy of sensitive resources on mobile devices to facilitate more effective requirement communication. Moreover, generating a risk assessment baseline is a compute-intensive task that does not quite fit resource-constrained mobile devices. Thus, we plan to offload such a task to trusted third-parties or users' public or private clouds in the future.

Regarding our current implementation of RiskMon, it does not address: (1) interactions between third-party applications; and (2) interactions that do not utilize Binder. This indeed illustrates potential attack vectors that can bypass RiskMon. Unauthorized accesses on resources of third-party applications [42] might be possible because such resources are not protected by system permissions. Also, two or more malicious applications can collude via local sockets or covert channels and evade the Binder-centric reference monitor in RiskMon. For our future work, we will extend our framework to maximize the coverage of attack vectors in our approach.

5.6 Related Work

Analysis of meta information: Meta information available on application markets provides general descriptions of applications. Recent work has proposed techniques to distill risk signals from them. Kirin [56] provides a conservative certification technique that enforces policies to mitigate applications with risky permission combinations at install time. Sarma et al. [106] propose to analyze permissions along-side with application categories in two large application datasets. Peng et al. [99] use probabilistic generative models to generate risk scoring schemes that assign comparative risk scores on applications based on their requested permissions. In addition to analysis on permissions, Chia et al. [41] and Chen et al. [40] performed large-scale studies on application popularity, user ratings and external community ratings. In particular, Pandita et al. proposed WHYPER [97] which automatically infers an application's necessary permissions from its description in natural languages. However, meta information does not accurately describe the actual behaviors of applications. RiskMon uses meta information to provide contextual information so as to complement the analysis on the runtime behaviors for risk assessment.

Static and dynamic analysis: Analysis on execution semantics of applications, such as static analysis of code and dynamic analysis of runtime behaviors, can reveal how applications use sensitive information. Stowaway [58] extracts API calls from a compiled Android application and reveals its least privilege set of permissions. Enck et al. [55] developed a decompiler to uncover usage of phone identifiers and locations. Pegasus [39] checks temporal properties of API calls and detects API calls made without explicit user consent. TaintDroid [52] uses dynamic information flow tracking to detect sensitive data leaking to the network. Regarding malware analysis, DroidRanger [128] and RiskRanker [67] are systematic and comprehensive approaches that combine both static and dynamic analysis to detect dangerous behaviors. DroidScope [121] reconstructs semantic views to collect detailed execution traces of applications. These work focuses on fundamental challenges for assessing actual risks incurred by applications. However, they do not provide a baseline to capture the appropriate behaviors under diverse contexts of different applications. Thus, their approaches are more intended for security analysts rather than end users.

Mandatory access control frameworks: RiskMon includes a lightweight reference monitor for Binder IPC. While it monitors IPC transactions for risk assessment, several frameworks mediate IPC channels as part of their approaches to support enhanced mandatory access control (MAC). SEAndroid [109] brings SELinux kernel-level MAC to Android. It adds new hooks in the Binder device driver to address Binder IPC. Quire [48] provides IPC provenance by propagating verifiable signatures along IPC chains so as to mitigate confused deputy attacks. Aurasium [119] uses libc interposition to efficiently monitor IPC transactions without modifying the Android platform. FlaskDroid [38] provides flexible MAC on multiple layers, which is tailored the peculiarity of the Android system. Along these lines, RiskMon captures Binder transactions with a fine-grained scheme to facilitate risk assessment on applications' runtime behaviors.

5.7 Summary

In this chapter, we have presented RiskMon that continuously and automatically measures risks incurred by a user's installed applications. RiskMon has leveraged machine-learned ranking to generate a risk assessment baseline from a user's coarse expectations and runtime behaviors of her trusted applications. Also we have described a proof-of-concept implementation of RiskMon, along with the extensive evaluation results of our approach.

Chapter 6

FLOW-DRIVEN ASSESSMENT

As we have discussed our approach to assess individual apps, we argue that it is equally important to assess the inter-application information flows. In this section, we propose an approach to systematically check intent-based inter-application communication among installed Android apps.

6.1 Problem Statement

Modern mobile operating systems have shifted into a security architecture that is fundamentally different from those of traditional desktop OSs. Mobile applications (commonly referred to as apps) run as unique security principles; they are isolated in their respective sandboxes and receive few privileges. In addition, the mobile OSs support inter-application communication that enables interoperability among apps so that multiple apps can collaborate to accomplish complex tasks. For example, an email client exports a picture file to a photo editor; the photo editor modifies the picture and posts it online through a social network client. Inter-application communication respects the Unix philosophy of “do one thing and do it well” and promotes modular design in apps.

A type of messaging objects called *intents* build a major and sophisticated inter-application communication mechanism in Android [42]. Intents are flexible as they can carry simple data and even inter-process communication primitives (*e.g.* Binder [23] and file descriptors [24]). Moreover, the intent attributes are rich with Android middleware semantics, which naturally facilitate access control decisions [37, 95]. Intent-based inter-application communication has received much research attention.

In general, two aspects are covered: previously unknown security limitations of intents [35, 42, 48, 60, 87] and generic policy-driven security extensions that remedy the limitations [30, 35–37, 73, 90, 91, 93, 95, 109, 129]. However, there is an overlooked gap between configuring generic security extensions and securing a specific Android device. Every app, every device, and every user are different. A policy analyst needs insights into the policies before she can accurately define how the apps in her device communicate through intents in her intended ways. To bridge the gap, we seek a systematic approach for a policy analyst to conveniently acquire such insights.

Defining and verifying the policy for each individual security extension that controls intent-based communication is a complex task for a policy analyst. The recent emerging security requirements, such as “bring your own device” (BYOD), call for fine-grained and precise policies. For example, a single mobile device may host a doctor’s personal apps and the apps of several clinics. The doctor and the clinics would require that the deployed security policies accurately enforce the boundaries between the apps of the respective stakeholders. Meanwhile, mitigating existing threats related to intents such as communication hijacking [42], confused deputy attacks [35, 60], and accidental data disclosure [90] requires the that policies are tailored to the peculiarities of each threat and each vulnerable app.

However, unlike the other inter-application communication mechanisms in Android, which are usually controlled by a single security extension, intent-based communication is mediated by multiple security extensions. While multiple security extensions promote the flexibility of controlling intent-based communication, they also introduce new challenges in definition and verification of their policies.

C-1: *Incompatible policies.* The security extensions define their own incompatible schemas and semantics. For example, FlaskDroid [37] inherits SELinux’s policy semantics of type enforcement. Saint [95] uses an XACML-like schema customized by the authors. IntentFirewall's policy is unique and unlike the other security extensions, however it specifies a critical set of tests on intent attributes. As far as we know, no existing policy checker can work with every extension's policy. Therefore, checking such incompatible policies remain a manual process that requires a policy analyst to master the details of every security extension.

C-2: *Distributed policies.* The security extensions store policies in distributed locations. For example, IntentFirewall stores its policy in an XML file, and intent filters are stored in internal data structures inside AMS and PMS. In addition, each extension tends to make the policy exclusively accessible to itself. In other words, each security extension makes its decision by itself and is not aware of the other security extensions. Consequently, no security extension possesses a holistic view of the reachability among installed apps as controlled by all the security extensions.

C-3: *Dynamic policies.* The security extensions may allow apps to specify and modify policies at run-time. For example, app-defined intent filters and permissions are prevalent in Android. Recent security extensions that implement decentralized information flow control (DIFC) also encourage apps to participate in policy management [90]. Thus, the policies continuously change as a user installs, removes, or upgrades the apps on her device. This requires proactive verification to guarantee that the dynamically changing policies comply with the security requirements.

To address the challenges in checking intent-base communication, we seek to build: *a*) a *general* policy checker that easily adapts to the policy schema of any security extension that controls intents; *b*) a *holistic* policy checker that aggregates the policies into a holistic and verifiable view; and *c*) a *proactive* policy checker that automatically acquires the live states of security extensions as snapshots of dynamic policies. With the policy checker, we attempt to systematically answer the following two questions regardless of specific security extensions, apps, or devices: *a*) what intents can an app send to a specific app; and *b*) what intents can an app receive from a specific app. Meanwhile, we expect the checker to be mostly automated so as to reduce the burden on policy analysts.

Suppose we have a mission-critical app that signs sensitive treatment plans. Finding out which apps can send intents to this signer app assists a policy analyst to determine the domain of authorized apps and to rule out untrusted apps that may exploit the signer's potential vulnerabilities (*e.g.* capability leaks [66]). Similarly, finding out the signer's reachable apps is necessary for preventing accidental disclosure [90] or deliberate data theft where a user shares the treatment plans with untrusted apps such as a cloud storage client. In addition, knowing exactly what intents an app can send and receive enables fine-grained policies and furthers the notion of domain isolation. For example, a domain can be defined as a set of apps, a set of incoming intents, and a set of outgoing intents. According to Chin *et al.* [42], two domains that do not share incoming intents are safe from intent eavesdropping attacks; two domains that do not share outgoing intents are safe from spoofing attacks. Pushing the level of domain granularity from apps to intents also pushes the level of detail beyond the comprehension of a human analyst. Our policy checker implements automated analysis to verify that the system-wide intent-based communication is configured as intended.

6.2 Intent Space Analysis: Model

We believe that creating the right abstraction model is the first step toward checking intent-based communication. In this section, we elaborate the intent space model that lays the foundation for intent space analysis.

6.2.1 Overview

We observe a few common characteristics after analyzing the existing security extensions that mediate intent-based communication. First, all the security extensions implement policy-driven mandatory access control. Second, these security extensions allow or deny an intent if the values of the intent's attributes match their policies. Third, they are cascaded in a chain; one security extension allows an intent by passing it to the next security extension. Based on these observations, we find that intent-based communication is analogous to a computer network: each app is an endpoint, an intent is a packet, and the security extensions behave like a chain of routers whose rules specify how they forward intents based on their "header" attributes. Inspired by packet header space analysis [83], we propose to model each security extension's inputs or outputs as a geometric *intent space* over intent attributes; and further we model each security extension's intent forwarding functionalities as a *transfer function.*

Figure 6.1 demonstrates a motivating example where App A sends intents to App B. For simplicity of the example, we consider only actions and categories, and we represent the actions on the x-axis and the categories on the y-axis. The initial space of App A is full in both dimensions because an app can create arbitrary intents before the intents are processed by any security extension. And because the security extensions only forward the intents that match certain actions or categories specified in their policies, the space gradually shrinks as the transformations T_1, T_2, and T_3 are applied to the initial space (Figure 6.1 (a)). The remaining space at App B indicates the intents that App A can send to reach App B. And if no space remains, App A cannot communicate with App B through intents. One step further, we combine the transfer functions into a composite transfer function that describes app-to-app space

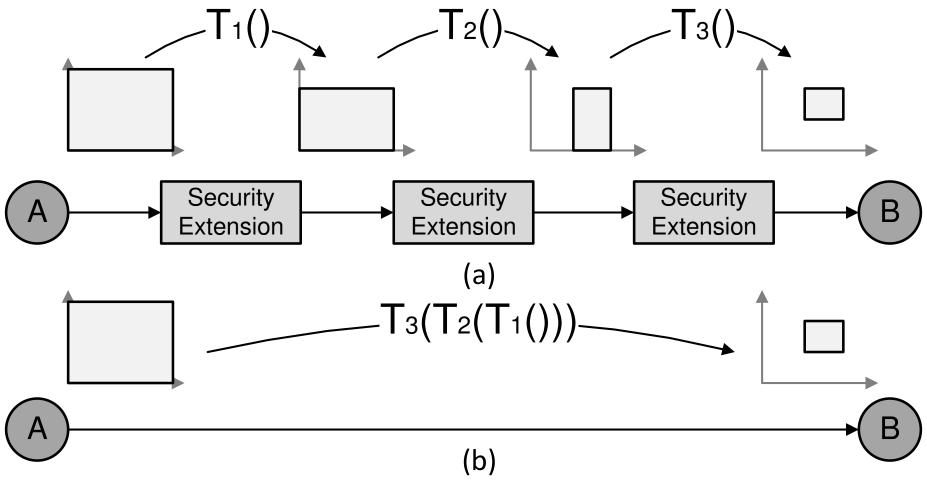


Figure 6.1: (a) The intent space shrinks as it passes security extensions, modeled here by the T_1, T_2, T_3. (b) Composing transfer functions to model app-to-app transformation.

transformation as illustrated in Figure 6.1 (b). This composite function captures all the security extensions. Thus, it describes the holistic intent forwarding state that we need for checking intent-based communication.

6.2.2 Intent Space

Formally, an intent space is a K-dimensional space of regular languages defined as $\mathcal{I} = \{.*\}^K$, where "`.*`" is the regular language that describes all words. The K dimensions correspond to K intent attributes, which are selected by the policy analyst based on her requirements. A policy analyst can set a smaller K if the security extensions to be analyzed do not inspect every intent attribute. An intent i maps to a point in the space, such as: {`action`: `SEND`,`category`: `DEFAULT`} [1] for $K = 2$. Multiple intents map to a subspace defined as a hypercube or a union of multiple hypercubes. A hypercube is represented with *exactly* K regular languages at K dimensions, such as {`action`: `SEND|SEND_MULTI`, `category`: ε (the empty string

[1]For clarity in this example we annotate the dimensions with the attributes.

language)}. Any hypercube with fewer than K dimensions or undefined dimensions is invalid and considered as an empty space $\varnothing$ in the subsequent computations.

6.2.3 Intent Space Algebra

Algorithms that check intent-based communication between two apps must determine whether an app's allowed outgoing intents overlap with the other apps' allowed incoming intents. To this end, we define the basic set operations on $\mathcal{I}$: *intersection*, *union*, *complementation*, and *difference*. Note that a point in $\mathcal{I}$ can be considered as a special hypercube whose regular languages contain only one word; and a subspace is a union of multiple hypercubes. We therefore define set operations for hypercubes and carry over the operations to other intent space objects. Throughout the rest of this chapter, we overload the term *intent space* to refer to all types of intent space objects including points, hypercubes, subspaces, as well as the entire intent space.

Intersection. The intersection of two intent spaces is computed by intersecting the regular languages at each dimension. Formally, given two intent spaces $i, j \subset \mathcal{I}$ and their dimension set $D = \{d_1, d_2, \ldots, d_k\}$, their intersection $i \cap j$ is $\{d_1 : regex_1^i \cap regex_1^j, \ldots, d_k : regex_k^i \cap regex_k^j\}$. For example, {`A[12]`, `C1`} $\cap$ {ε, `C1`} is equivalent to {`A[12]`, `C1`} and {`A[12]`, `C1`} $\cap$ {`A3`, `C1`} is equivalent to {$\varnothing$, `C1`}. Note that {$\varnothing$, `C1`} is missing a dimension and thus is considered as an empty space $\varnothing$.

Union. A union of intent spaces may not be simplified to a single intent space. For example, the union of two intent spaces {`A1|A2`, `C1`} and {`A3`, `.*`} cannot be represented by any single hypercube and we simply represent the union as {`A1|A2`, `C1`} $\cup$ {`A3`, `.*`}. We can simplify the result if the intent spaces are on the same hyperplane. For example, {`A1|A2`, `C1`} $\cup$ {`A3`, `C1`} is equivalent to {`A[1-3]`, `C1`}.

Complementation. The complement of an intent space i is the union of all the intent spaces that do *not* intersect with i. Recall that the intersection of two intent spaces is an empty space if the intersection is missing any of the K dimensions. We compute i's complement $\overline{i}$ with Algorithm 8, which finds all non-intersecting intent spaces by replacing the regular language at one dimension with its complement if the language is not `.*` and setting `.*` at the other dimensions. For example, the complement of $\{\varepsilon\}$ is $\{.*\}$ and the complement of $\{\texttt{A1}, \texttt{C1}\}$ is $\{\overline{\texttt{A1}}, \texttt{.*}\} \cup \{\texttt{.*}, \overline{\texttt{C1}}\}$.

Algorithm 2: Computing an intent space's complement

```
   Data: $i$
   Result: $\overline{i}$
1  $i' \leftarrow \varnothing$;
2  for dimension $d_i \in D$ do
3      $L \leftarrow$ regular language at $d_i$;
4      if $L \neq .*$ then
5          $i' \leftarrow i' \cup \{d_1 : .*, \ldots, d_i : \overline{L}, \ldots, d_k : .*\}$;
6      end
7  end
8  return $i'$
```

Difference: The difference (or subtraction) is computed with intersection and complementation, *i.e.*, $i - j = i \cap \overline{j}$. For example, $\{\texttt{A1|A2}, \texttt{.*}\}$ - $\{\texttt{A2}, \texttt{.*}\}$ is equivalent to $\{\texttt{A1|A2}, \texttt{.*}\} \cap \{\overline{\texttt{A2}}, \texttt{.*}\}$, which is $\{\texttt{A1}, \texttt{.*}\}$. A slightly more complicated example which reuses the complement of $\{\texttt{A1}, \texttt{C1}\}$ is shown below:

$$\begin{aligned}
&\{\texttt{A1|A2,C1|C2}\} - \{\texttt{A1,C1}\} \\
&= \{\texttt{A1|A2,C1|C2}\} \cap \overline{\{\texttt{A1,C1}\}} \\
&= \{\texttt{A1|A2,C1|C2}\} \cap (\{\overline{\texttt{A1}}, \texttt{.*}\} \cup \{\texttt{.*}, \overline{\texttt{C1}}\}) \\
&= \{\texttt{A2,C1|C2}\} \cup \{\texttt{A1|A2,C2}\}
\end{aligned}$$

6.2.4 Transfer Function

For convenience of analysis, we assume that all security extensions deny by default. For those security extensions that accept by default, it is trivial to reduce them into deny-by-default extensions with a least-priority rule that accepts everything. Therefore, apps cannot communicate if the security extensions specify no rule. Conversely, the rules of a security extension that allow/deny some intents from one app to another app essentially specify how the security extension *forwards* or *drops* intents from the source app to the destination app. As we represent intents as an intent space, we model a security extension's intent forwarding and dropping functionality as intent space transformation and represent a security extension with a *transfer function.* Given that the space of all apps is $\mathcal{A}$, a transfer function T is formally defined as:

$$T : (a, i) \to 2^{\mathcal{A}\times\mathcal{I}}, a \in \mathcal{A}, i \subset \mathcal{I}$$

To aggregate multiple transfer functions into a holistic view, we iteratively apply each (a, i) tuple of the output of a transfer function to the input of the next transfer function and build a composite transfer function.

A transfer function captures the transformation that a security extension performs on $\mathcal{A}$, $\mathcal{I}$, or both. Suppose we are to model a simple security extension that works like a Layer-2 network switch: it only supports coarse-grained control over which app can send intents to another app regardless of intent attributes. Such an extension can be modeled as a transfer function that transforms only on $\mathcal{A}$. IntentFirewall denies an app from sending a specific intent regardless of the intent's destination apps. It therefore can be modeled as a transfer function that only transforms on $\mathcal{I}$.

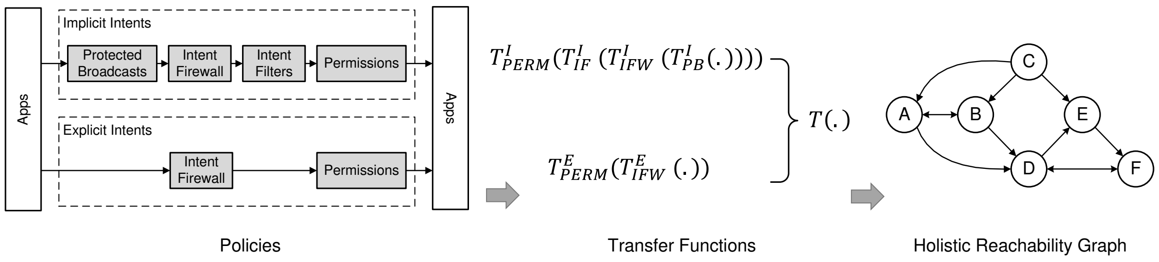


Figure 6.2: IntentScope System Workflow.

We elaborate more details about how we model security extensions for intent space analysis in the subsequent section.

6.3 Intent Space Analysis: System

In this section, we describe our policy analysis framework IntentScope which supports intent space analysis. To demonstrate its generality, we also discuss how IntentScope works with the AOSP security extensions and their policies. We emphasize that IntentScope is not limited to only the discussed security extensions in this chapter.

6.3.1 System Workflows

Figure 6.2 depicts the workflow of IntentScope. In general, IntentScope starts from acquiring the policies of security extensions, then creates transfer functions, and converts the composite transfer function into a holistic reachability graph for subsequent analysis.

Acquiring Policies

The policy of a security extension is often referred to as a dedicated file stored in the filesystem. In this work, we opt for a more general definition of policy and propose to acquire all the states and configurations of security extensions so long as they specify how the intents are forwarded. To this end, we create a privileged watchdog app for IntentScope that proactively observes policy changes and automatically takes a snapshot of the policies. The implementation of the watchdog app is largely specific to the analyzed security extensions. For example, intent filters are registered by apps and maintained by AMS and PMS. The watchdog app may acquire the registered intents filters on an Android device by dumping the internal states of AMS and PMS after an app registers/unregisters any intent filter.

Creating Transfer Functions

Next, we map the acquired policies onto transfer functions. Given that a security extension makes decisions based on its loaded *policy* and implemented policy interpretation *logic*, a transfer function that models the intent forwarding state must capture both. While the policy can be automatically retrieved by IntentScope's watchdog app, the policy interpretation logic still requires manual effort to model. IntentScope requires a policy analyst or the security extension's authors to define a transfer function for its policy interpretation logic and to create a policy parser that instantiates the corresponding transfer function. Note that this logic construction overhead is only performed once as the defined transfer functions can be reused and the parsers can automatically instantiate transfer functions. We elaborate our transfer functions for the AOSP security extensions in Section 6.3.2.

Building a Holistic Reachability Graph

To facilitate analysis and visualization, we propose to convert the composite transfer function into a directed graph that represents inter-application reachability. Formally, a holistic reachability graph is denoted as $G = (V, E)$, where V is a set of vertices that correspond to the installed apps and E is a set of edges that correspond to the intent spaces that an app can send to reach another app. Constructing such a reachability graph is straight forward. Each app maps to a vertex in the graph. For each app, we apply the composite transfer function on its initial intent space (*e.g.*, $\{.*\}^K$) and add a directed edge if any non-empty intent space remains at the destination app. We assign the remaining intent spaces on the edges as their weights, which allows IntentScope to support flexible queries and graph pruning as a policy analyst adds constraints on the graph.

6.3.2 Transfer Functions for AOSP Security Extensions

Intent filters, IntentFirewall, protected broadcasts, and permissions are the integral parts of AOSP and therefore widely deployed in COTS Android devices. They also serve as reference implementations for other security extensions. For example, Apex [91] and CRePe [44] extend the permissions; and SEAndroid controls intents with a slightly modified IntentFirewall [25]. Based on these observations, we believe that the AOSP security extensions are a good starting point to demonstrate that IntentScope is general, because it can effectively work with their policies. In the remainder of this section, we share our experiences of modeling these security extensions for intent space analysis. Although we are not the first to formally model them, we provide the most accurate models by covering a complete set of intent attributes and undocumented logic in the security extensions. Unless stated otherwise, the contents in this section are based on our manual analysis of the `kitkat-release` branch in AOSP.

As shown on the left side of Figure 6.2, two chains of security extensions control implicit and explicit intents. We define two intent spaces: (1) $\mathcal{I}_I$ as a six-dimensional implicit intent space over five intent attributes `action`, `category`, `scheme`, `authority`, `type` and one additional attribute `permission`; and (2) $\mathcal{I}_E$ as a two-dimensional explicit intent space over `component name` and `permission`. Note that the `permission` of an intent is inherited from the app that created the intent. The chain for implicit intents consists of four security extensions: protected broadcasts, IntentFirewall, intent filters, and permissions; and we define their transfer functions over $\mathcal{I}_I$ as T^I_{PB}, T^I_{IFW}, T^I_{IF}, and T^I_{PERM}. The chain for explicit intents includes two security extensions: IntentFirewall and permissions; and we define their transfer functions over $\mathcal{I}_E$ as T^E_{IFW} and T^E_{PERM}.

Intent Filters: T^I_{IF}

An intent filter specifies the implicit intents that it allows to be forwarded to the next security extension. Therefore, an intent filter's output is the intersection of the input intent space and the intent filter's corresponding intent space. Suppose a component $dst.c$ in an app dst has an intent filter $filter$ that describes an intent space $i^{dst.c}_{filter}$. Then, an intent filter transforms (src, i) to $(dst, i \cap i^{dst.c}_{filter})$. Note that the transformation is performed on both $\mathcal{A}$ and $\mathcal{I}$. Given the installed apps on a device as a set A, we combine their registered intent filters and define T_{IF} as follows:

$$\begin{aligned} T^I_{IF}(m, i) = \{(n, i \cap i^{n.c}_{filter}) | i \cap i^{n.c}_{filter} \neq \varnothing, \\ \forall c \text{ is a component of } n, \forall n \in A, n \neq m, \\ i, i^{n.c}_{filter} \subset \mathcal{I}_I\} \end{aligned}$$

Next we explain how we map an intent filter to its intent space i_{filter}. In general, an intent filter accepts an intent if the intent's attributes pass a series of tests on the intent filter's attributes. Therefore, we reduce the problem of modeling an intent filter to constructing a set of regular languages which consists of the words that pass each respective test.

Action Test: An intent passes the action test if the intent's action matches any action in the intent filter. Therefore, we map the one or more actions of an intent filter onto a regular expression that concatenates the *escaped* action strings and separates them with the vertical bar character `|`, such as `VIEW|EDIT`. There are two corner cases in this test. First, zero action in a filter fails the test. Second, zero action in an implicit intent also fails the test. We capture both cases with a regular expression `[]`, which denotes an empty language whose intersection with any language is empty. Note that the Android documentation is incorrect with respect to the second corner case: "*if an intent does not specify an action, it will pass the test as long as the filter contains at least one action*". The reason is that `queryIntent()` in the `IntentResolver` class eventually denies such intents even though `matchAction()` in the `IntentFilter` class allows. Our experiments also confirm this behavior. Interested readers are referred to the source code [2] for more details.

Scheme Test: An intent passes the scheme test if the intent's scheme matches any scheme in the filter. Therefore, the regular expression here is constructed in the same way as the action test, *e.g.*, `http|gopher`. This test also has unique cases. First, an intent filter without any scheme still matches three schemes: `content`, `file`, or an empty string. We represent them with a regular expression `file|content|`, where the last `|` matches the empty string. Second, an intent without any scheme passes the scheme test only if the intent filter does not specify any scheme. We consider such intents as intent spaces whose scheme is an empty string.

Authority Test: This test is dependent on the scheme test. If the intent filter does not specify any scheme, this test automatically passes regardless of the authority. This test also passes if the filter does not specify any authority. Thus, we use `.*` to match any authority in these two cases. An intent without any authority passes the test only if the filter has no authority. We represent such intents with an empty

[2]`https://goo.gl/A1auU5` and `https://goo.gl/cdzxg8`

string at the authority dimension. Otherwise, an intent passes the authority test if its authority matches any authority in the filter.

Type Test: An intent passes the type test if the intent's MIME type matches any type in the filter. The challenge here is the wildcard character `*` in MIME type strings. For example, `*` and `*/*` match any type; and `audio/*` matches any subtype of `audio`. To maintain the semantics of the wildcard character, we convert `*` and `*/*` to `.*`. The slash character `/` is a special character in regular expressions so we escape it as `\/`. For example, `audio\/.*|video\/mp4` represents every audio subtype and a single video type. Moreover, an intent filter that has no type accepts only the intents that have no type. Therefore, zero type in either the intent or the filter maps to an empty string.

Category Test: Unlike the other attributes, an intent can include more than one category. An intent passes the category test if *every* category in the intent matches a category in the filter, *i.e.*, the intent's category set is the subset of the filter's category set. To capture this logic, we construct a regular language for an intent filter's categories with three steps: (1) escape the category strings; (2) concatenate the escaped strings and separate them with `|`; and (3) surround the concatenated string with `(` and `)*`. For example, the subsets of an intent filter's category set {`DEFAULT`, `LAUNCHER`, `BROWSABLE`} are represented with a single regular expression `(DEFAULT|LAUNCHER|BROWSABLE)*`. This expression also matches zero category and duplicate categories specified in an intent. The other corner cases are similar to those of the type test. No specification of category in an intent or a filter maps to an empty string. An intent filter with no category accepts only the intents with no category.

Intent filters do not transform on the permission dimension. The regular language at the permission dimension of all i_{filter} intent spaces is `.*`.

IntentFirewall: T^I_{IFW} and T^E_{IFW}

IntentFirewall is a policy-driven MAC framework that block apps from *sending* specific intents. The policy files, located at `/data/system/ifw/*.xml`, specify a list of *firewall filters* (fwfilters for short) that describe the implicit or explicit intents to be blocked for a specific sender app. We model IntentFirewall as a transformation over $\mathcal{I}_I$ or $\mathcal{I}_E$ that subtracts the intent space of each fwfilter from the input intent space. Suppose a fwfilter that blocks an app *src* is represented with an intent space $i^{src}_{fwfilter}$. T^I_{IFW} and T^E_{IFW} are defined in the same way as follows:

$$\begin{aligned} T^I_{IFW}(a,i) = \{(a, i - \bigcup i^a_{fwfilter}) | i - \bigcup i^a_{fwfilter} \neq \varnothing, \\ \forall fwfilter \text{ that blocks the sender app } a, \\ i, i^a_{fwfilter} \subset \mathcal{I}_I\} \end{aligned}$$

$$\begin{aligned} T^E_{IFW}(a,i) = \{(a, i - \bigcup i^a_{fwfilter}) | i - \bigcup i^a_{fwfilter} \neq \varnothing, \\ \forall fwfilter \text{ that blocks the sender app } a, \\ i, i^a_{fwfilter} \subset \mathcal{I}_E\} \end{aligned}$$

Next we explain how we construct the intent space $i_{fwfilter}$ for a fwfilter over the implicit intent space $\mathcal{I}_I$ and the explicit intent space $\mathcal{I}_E$, respectively. In general, we construct $i_{fwfilter}$ according to IntentFirewall's two-phase intent attribute matching process.

If a fwfilter is for implicit intents, IntentFirewall first considers the fwfilter as an intent filter and tests the intent attributes with the same tests as we discussed in Section 6.3.2. We skip modeling this phase for brevity. In the second phase, IntentFirewall tests the intent attributes with common string tests, such as `isEqual`, `isStartsWith`, `isContained`, and `matchRegex`. Therefore, we model these tests with their equivalent regular expressions. For example, `isStartsWith=abc` maps to a regular expression `abc.*`; `isContained=def` maps to a regular expression `.*def.*`. The tests can be aggregated by computing the intersection of the regular expressions. For example, two tests `isEqual=abc` and `isStartsWith=ab` map to `abc`.

For a fwfilter that filters explicit intents, we also construct its intent space in two phases. In the first phase, IntentFirewall checks if an explicit intent's component name matches the one specified in the fwfilter. Thus, we simply copy the fwfilter's escaped component name to the corresponding dimension in $i_{fwfilter}$. There are two corner cases to be handled. An explicit intent with no component name is dropped immediately because it resolves to nowhere. A fwfilter with no component name does not block any explicit intent. We model the former case with a regular expression `[]` and model the latter case with a regular expression `.*`. In the second phase, Intent Firewall tests the intent's component name with the identical string tests so we do not rephrase how we model them. Finally, both T^{I}_{IFW} and T^{E}_{IFW} do not transform an intent space at the permission dimension because IntentFirewall does not inspect permissions.

Note that IntentFirewall is a relatively new security extension in AOSP with no official documentation and limited comments in the code. At first we referred to the unofficial documentation maintained by Yagemann [120] to define the transfer functions. However, we found unexplained behaviors of IntentFirewall when we tested IntentFirewall's sample policies, which led us to the discovery of the overlooked second matching phase. In order to obtain an accurate and comprehensive model, we manually derived the transfer functions presented in this section from IntentFirewall's source code [3] .

[3] `https://goo.gl/e4zzxL`

Permissions: T^I_{PERM} and T^E_{PERM}

Permissions constrain an app's capability to *receive* intents from other apps. Suppose an app has a sensitive component that only accepts the intents from authorized apps. Then, the app can define a permission and assign it to the component, which requires the component's callers to hold the exact same permission. If we treat intents as if they inherit the permissions of their creator/sender apps, a permission's role is to forward only the intents that have matching permissions. Therefore, a permission's output is the intersection of the input intent space and the permission's own intent space. Note that permissions do not transform on $\mathcal{A}$ because the other security extensions have already resolved the destination app/component. Suppose a component $dst.c$ is protected by a permission p described by an intent space $i^{dst.c}_{c.p}$. The transformation is defined as $(dst.c, i) \to (dst.c, i \cap i^{dst.c}_{c.p})$.

We define T^I_{PERM} and T^E_{PERM} as follows:

$$\begin{aligned} T^I_{PERM}(a, i) = \{(a.c, i \cap i^{a.c}_{c.p}) | i \cap i^{a.c}_{c.p} \neq \varnothing, \\ \forall c \text{ is a component of } a, \\ c \text{ is protected by } c.p, \\ i, i^{a.c}_{c.p} \subset \mathcal{I}_I\} \\ T^E_{PERM}(a, i) = \{(a.c, i \cap i^{a.c}_{c.p}) | i \cap i^{a.c}_{c.p} \neq \varnothing, \\ \forall c \text{ is a component of } a, \\ c \text{ is protected by } c.p \\ i, i^{a.c}_{c.p} \subset \mathcal{I}_E\} \end{aligned}$$

Mapping a permission to an intent space i_p is straight-forward. The regular language at the permission dimension of i_p is the escaped permission string. A special case is that a content provider may have separate permissions for reading and writing. Similar to the action test in intent filters, we model this case with a regular expression perm_r|perm_w, based on the fact that an app with either the read or write permission can access the content provider. The regular languages at the other dimensions are .*, leaving the intent space unchanged at these dimensions.

Protected Broadcasts: T^I_{PB}

Protected broadcasts are a set of implicit intents with special actions that only the apps whose UIDs are SYSTEM, BLUETOOTH, PHONE, or SHELL can send. The other apps are prevented from sending such intents. Similar to IntentFirewall, we model protected broadcasts as a space transformation that subtracts the intent spaces of protected broadcasts from the input intent space if the input app is not a system-app. Suppose each protected broadcast maps to an intent space $i_{protected}$. Then, we define the transfer function for protected broadcasts as follows:

$$T^I_{PB}(a,i) = \begin{cases} (a,i) & \text{if } a \text{ is an allowed app} \\ (a, i - \bigcup i_{protected}) & \text{otherwise} \end{cases}$$

$$i, i_{protected} \subset \mathcal{I}_I$$

A list of actions used by protected broadcasts is available in the Android SDK [4] . Thus, we build an intent space $i_{protected}$ for each action by assigning the escaped action string into the action dimension of the space. The other dimensions do not involve space transformation and remain with a regular expression .*.

Composite Transfer Function

As we have defined the transfer function for each individual security extension, we combine them together to build the composite transfer function. The composite function covers two chains of transfer functions for the implicit and explicit intent

[4] ANDROID_SDK_ROOT/platforms/android-19/data/broadcast_actions.txt

space, respectively. To build each chain of transfer functions, we start from integrating the transfer functions of those security extensions that restrict an app from *sending* intents. Then, the transfer functions of the security extensions that restrict an app from *receiving* intents follow. For the transfer functions defined in this section, their composite transfer function T is defined as:

$$T(a,i) = \begin{cases} T^I_{PERM}(T^I_{IF}(T^I_{IFW}(T^I_{PB}(a,i)))) & \text{if } i \subset \mathcal{I}_I \\ \\ T^E_{PERM}(T^E_{IFW}(a,i)) & \text{if } i \subset \mathcal{I}_E \end{cases}$$

6.4 Evaluation

In this section, we first discuss a prototype implementation of IntentScope. We then present the experiments in which we apply IntentScope to check intent-based communication mediated by the AOSP security extensions installed in commodity Android devices and customized Android OSs. We conclude with an evaluation of the throughput of our system.

6.4.1 Implementation

IntentScope includes an implementation of the intent space model, a watchdog app that acquires the policies of the AOSP security extensions that control intents, a set of policy parsers that build and compose transfer functions, and a graph builder that converts the composite transfer function into the holistic reachability graph.

The intent space model is built on Augeas Libfa [18], a native library that supports accurate and fast operations on regular expressions. In particular, we opt for Hopcroft's DFA minimization algorithm [75] to minimize regular expressions. This algorithm runs in $O(nlogn)$ time in the worst case, where n is the number of states of a regular expression's equivalent DFA. The watchdog app runs as a privileged system

Table 6.1: Evaluated Android Devices/OSs and Generated Reachability Graphs

	Device	**OS**	**|V|**	**$\|E_I\|$** / **$\|E_E\|$**	**Global Clustering Coefficient**	**Standard Deviation**
1	Samsung Galaxy Note II	Customized Android	311	880,456	0.986	0.007
				979,993	0.994	0.006
2	LGE Nexus 4	Stock Android	108	155,369	0.971	0.014
				138,651	0.990	0.009
3		MIUI v5	104	99,170	0.979	0.013
				118,707	0.991	0.009
4		CyanogenMod 11 M12	85	38,606	0.974	0.015
				47,458	0.989	0.011

Table 6.2: Apps Ranked by PageRank

	Highest in G_I	**Lowest in G_I**	**Highest in G_E**	**Lowest in G_E**
1	**com.viber.voip**	com.android.proxyhandler	com.android.contacts	com.sec.enterprise.permissions
	com.android.contacts	com.monotype.android.font.cooljazz	com.android.phone	com.samsung.android.mdm
	com.android.settings	com.sec.android.provider.badge	com.android.settings	com.samung.android.sdk.spenv10
2	com.google.android.apps.plus	com.android.dreams.basic	**com.google.android.setupwizard**	com.android.dreams.basic
	com.android.settings	com.android.providers.userdictionary	com.google.android.apps.plus	com.android.wallpaper
	com.google.android.apps.gms	com.android.vpndialogs	com.android.settings	com.google.android.apps.docs.editors.slides
3	com.android.mms	com.android.pacprocessor	com.android.email	cm.android.printspooler
	com.android.contacts	com.android.sharedstoragebackup	com.android.mms	com.android.nfc
	com.android.settings	com.miui.providers.weather	com.android.settings	com.android.noisefield
4	com.android.gallery3d	com.android.nfc	com.android.contacts	com.android.nfc
	com.android.email	com.android.backupconfirm	com.android.email	com.android.incallui
	com.android.contacts	com.android.sharedstoragebackup	com.android.settings	com.android.printspooler

app. It dumps the internal states of PMS and AMS to acquire a comprehensive list of intent filters and permissions, regardless of whether they are statically declared in apps' manifest or dynamically registered in app's code. The watchdog app also fetches the relevant files where IntentFirewall and protected broadcasts store their policies. While IntentScope does not seek to be a realtime checker, policy re-acquisition occurs periodically and after app installs/uninstalls. As the operations over intent spaces are both computation and memory intensive, the parsers and graph builder run on a PC rather than on the mobile device where the watchdog app runs.

6.4.2 Experimental Setup

We evaluated IntentScope on two Android devices and four Android-based OSs, as shown in Table 6.1. The Galaxy Note ran Samsung's deeply customized Android (4.4.2), which pre-installed a large number of Samsung's apps. In addition, we loaded it with the top 50 apps from a list [22] of most downloaded Android apps in the Google Play marketplace. The Nexus 4 ran three OSs, including stock Android (5.0), MIUI (4.4.2), and CyanogenMod (4.4.4). We kept them as they were and did not install additional apps. In particular, the first two OSs pre-installed a few proprietary Google-branded apps. MIUI and CyanogenMod did not include these apps due to licensing restrictions.

For each OS, we started each installed app and kept it in the foreground for at least 30 seconds. We assume that the apps had requested AMS and PMS to dynamically register any intent filters or permissions. Then we applied IntentScope to generate a reachability graph G and two subgraphs G_I and G_E that respectively represent the holistic forwarding state of implicit and explicit intents. Each vertex represents an app identified by its package name rather than UID [5] . Parallel edges are allowed and prevalent in the graphs to capture the multiple entry points of an app.

6.4.3 Graph Overview

Table 6.1 lists the number of vertices, the number of edges (including parallel edges), and the global clustering coefficient (measured without parallel edges) of each G_I and G_E. A global clustering coefficient is a measure of the degree to which vertices in a graph tend to cluster together, defined as:

[5]Apps with the same UID are considered as separate apps but share the permissions of one another [33].

$$C_{global} = \frac{3 \times number\ of\ triangles}{number\ of\ connected\ triples\ of\ vertices}$$

We opted for this measure to get a general idea about how freely the installed apps on a mobile OS are allowed to communicate with one another. As the clustering coefficient of a clique is 1, the measured values of C_G indicate that the vertices in all the graphs are densely connected, which is in line with our observation that most apps have at least one component (the main activity) exposed to other apps. The large number of edges also imply the complexities of managing fine-grained policies for intent-based communication.

Given the large number of apps/vertices and edges, prioritizing the apps that expose larger attack surfaces is critical for efficiency in policy management. Therefore, we propose to identify such apps with PageRank [96]. The underlying intuition is that such apps are more likely to be accessed by other apps and thus have more incoming edges, and the apps that have direct incoming edges from such apps are also likely to be attacked. Table 6.2 lists the apps in the four mobile OSs with the highest and lowest rankings. Most of the listed apps are in line with intuition, such as `com.android.settings` and `com.android.email`. Here we discuss two apps which are displayed in bold in Table 6.2. The app `com.google.android.setupwizard` is highly ranked because it exports 69 components that can be accessed with explicit intents. The app `com.viber.voip` is highly ranked because of its 94 intent filters that expose the components to implicit intents.

6.4.4 Experiments

IntentScope answers the questions: *what intents can an app send and receive?* Given the holistic reachability graph generated by IntentScope, checking what intents an app can send is equivalent to checking the vertex's outgoing edges as well as the intent spaces assigned on them. Conversely, checking what intents an app can receive is equivalent to checking the incoming edges. In addition, IntentScope supports flexible queries backed by regular expressions. Next we elaborate four experiments in which we leverage the insights provided by IntentScope to identify potential vulnerabilities due to errors in security policies of the AOSP security extensions.

Zero Permission $\neq$ Zero Privilege

Enforcing least privilege is a common practice in mobile security. While recent work [44, 91, 115] attempts to control and minimize the set of an app's granted permissions, we are interested in another question: *what can an app do if it has no permissions*. In this experiment, we created and installed such a *zero-permission app*. We then checked what components this app can reach with its allowed intents. This experiment helps a policy analyst reveal the exposed components that could possibly be exploited by even a zero-permission app. If any sensitive components are exposed, the details of the allowed intents that reach these components provide the necessary knowledge for a policy analyst to create precise policies that protect them.

We find that zero permission does not necessarily mean zero privilege as users might expect. Table 6.3 shows the number of the zero-permission app's reachable apps (*i.e.* out-neighbors) and its local clustering coefficient. A local clustering coefficient measures the degree to which a vertex and its neighbors tend to cluster:

$$C_{local}(v) = \frac{number\ of\ edges\ among\ v's\ neighbors}{number\ of\ possible\ edges\ among\ v's\ neighbors}$$

The flexible queries supported by IntentScope also allow a policy analyst to pinpoint the intents that have interesting semantics. In the Galaxy Note, we found that this zero-permission app can send implicit intents that contain an interesting

Table 6.3: Reachability of a Zero-Permission App

	1		2		3		4	
#Outgoing edges	2,767	3,072	1,443	1,280	955	1,142	454	557
#Reachable apps	241	263	77	92	79	90	62	72
Loal clustering coefficient	0.943	0.968	0.905	0.960	0.927	0.968	0.914	0.961

scheme called `android_secret_code`. For example, one of the reachable apps is `com.sec.android.app.wlantest`, which accepts intents with an action `android.provider.Telephony.SECRET_CODE`, an authority of `526`, and a scheme of `android_secret_code`. Another reachable app `com.wssyncmldm` is a sensitive app that can silently download and install apps. Therefore, an app with no permissions could exploit a vulnerability in this app in order to download and install apps, thus escalating the privilege of the zero-permission app without exploiting the underlying OS. Due to time constraints, we did not discover any exploitable vulnerabilities. Yet a recent attack [98] demonstrates the feasibility of taking over an app with a malformed intent.

Fine-grained Domain Isolation

Chin *et al.* [42] presents a limitation of intent-based communication. Suppose a malicious app Mallory attempts to attack a legitimate and sensitive app Alice and existing policies prevent their direct communication. The limitation allows Mallory to eavesdrop the intents from Alice to Bob and allows Mallory to send spoofed intents to Alice. This situation calls for a fine-grained domain isolation model that not only considers apps but also includes intents. IntentScope is useful in this scenario because it provides insights about intents.

Specifically, two apps are not isolated with respect to eavesdropping attacks if they share in-neighbors and incoming intents in the reachability graph. They are not isolated with respect to spoofing attacks if they share out-neighbors and outgoing intents. Based on this observation, IntentScope guarantees intent isolation between two apps if: (1) the apps are not neighbors of each other; *and* (2) the intent spaces of their incoming edges from common in-neighbors do not intersect; *and* (3) the intent spaces of their outgoing edges to common out-neighbors do not intersect.

As a case study, we checked the intent isolation between two apps in the Galaxy Note: `com.android.externalstorage` and `com.fmm.dm`. The former is an Android system app. The latter is believed to be bloatware as reported on several online forums. Figure 6.3(a) depicts their 8 common in-neighbors. IntentScope reported that the intent spaces do not intersect, which implies that no app steals any intent from the other. However, these two apps share 242 common out-neighbors and the intersection of the intent spaces is not empty. Therefore, these apps are still susceptible to spoofing attacks.

Enumerating Multi-app Workflows

In modern mobile operating systems, it is common for a user to orchestrate multiple apps for a large and user-defined task. For example, a user may streamline a workflow of downloading, viewing, editing, and sending a picture with a chain of apps. Under the hood of Android, a multi-app workflow is implemented as a calling sequence of intents. While controlling such workflows has been well covered by Nadkarni and Enck [90], enumerating possible workflows would facilitate defining appropriate policy for Aquifer [90] and similar access control systems.

In this experiment, we applied IntentScope to enumerate the workflows in MIUI that match the aforementioned example. Specifically, we started from an app `com.android.providers.downloads`, which manages downloaded files. We then performed a breath-first search on the reachability graph for a sequence of implicit intents as follows:

1. action=`android.intent.action.VIEW`, scheme=`content`,

   category=`android.intent.category.BROWSABLE`;

2. action=`android.intent.action.EDIT`, type=`image/*`;

3. action=`android.intent.action.SEND`, type=`image/*`.

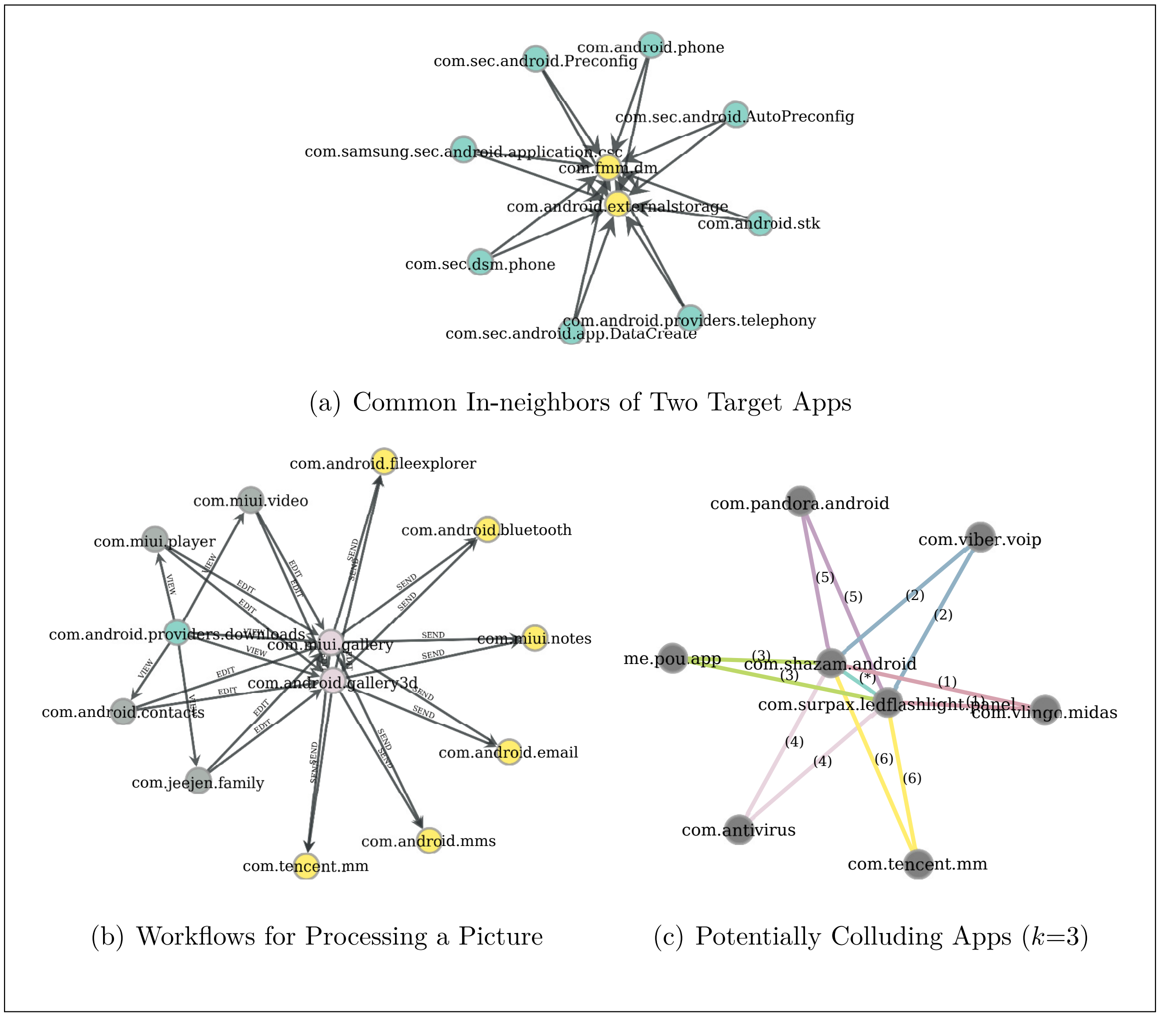
Figure 6.3(b) shows the matching workflows that start from the cyan node. The grey nodes are the first hop; the purple nodes in the middle are the second hop. Note that the purple nodes also serve as the first hop because the photo editors can also handle the `VIEW` action. The yellow nodes represent the last hop where data may leave a mobile device via emails, Bluetooth, or MMS messages.

Table 6.4: System Throughput

	$\lvert E_I \rvert$	**Avg. Time (s)**	**StdDev (s)**	**# edges/sec**	$\lvert E_E \rvert$	**Avg. Time (s)**	**StdDev (s)**	**# edges/sec**
1	800,456	302.05	5.73	2,915	979,993	115.57	2.02	8,454
2	155,369	70.08	3.02	2,217	138,651	21.59	0.74	6,422
3	99,170	38.69	0.92	2,563	118,707	16.92	1.02	7,014
4	38,606	15.63	1.00	2,469	47,458	6.77	0.45	7,013
Average				2,541				7,225

Discovering Permission Re-Delegation Paths

An unprivileged app without permissions can delegate a privileged app with the permissions to perform sensitive tasks [60]. Existing research [35, 60] detects and mitigates permission re-delegation between two apps at runtime when they communicate. While mitigation at runtime is one solution, we expect to enable a policy analyst to be aware of potential permission re-delegation paths *before* apps may execute. Meanwhile, the intents used along re-delegation paths provide semantics for the policy analyst to make informed decisions and take precise actions against the privileged apps that could be abused.



(a) Common In-neighbors of Two Target Apps

(b) Workflows for Processing a Picture

(c) Potentially Colluding Apps (k=3)

Figure 6.3: Experimental Results

We propose to use *connected subgraphs* to represent permission re-delegation paths in a reachability graph. A subgraph is connected if every pair of its vertices has a path that consists of *only* the vertices in the subgraph. This is analogous to the situation where multiple apps collude but cannot relay their communication via other apps. We define the problem of discovering re-delegation paths as follows: given a set of critical permissions denoted as CP, find all the connected subgraphs of k vertices that satisfy:

- Each app (vertex) holds at least one permission but not all the permissions in CP.
- The union of the apps' permissions is a superset of CP.

The best algorithm we found to generate connected subgraphs of k vertices is $ConSubG(G, k)$ [82], whose worst-case time complexity is exponential in k. The performance of this algorithm is generally acceptable because we rarely encounter cases where more than five apps collude.

We targeted the 50 third-party apps installed on the Galaxy Note and set $k = 3$. The critical permission set included three permissions: `BLUETOOTH_ADMIN`, `NFC`, and `FLASHLIGHT`, all of which are for accessing hardware devices that may significantly affect battery life. Figure 6.3(c) demonstrates 6 groups of apps (triangles) that can possibly collude to cover the critical permissions. In particular, the two apps in the center respectively hold `FLASHLIGHT` and `NFC`, while the surrounding six apps hold `BLUETOOTH_ADMIN`.

Even though the discovered eight apps are mostly downloaded and seem to be trusted by general users, they may carry third-party libraries or vulnerable components that are exploitable by other apps. In other words, they may not deliberately collude, but could be exploited by other apps to acquire privileges. The analysis discussed in this experiment can be combined with the other analyses (*e.g.* zero-permission apps) to further generate knowledge for a policy analyst to take precautions before real exploits occur.

6.4.5 System Throughput

To understand the performance of IntentScope, we performed a microbenchmark to evaluate the number of edges that IntentScope can check in a second. Given that checking an edge is done by testing whether the intersection of the edge's intent space and a given intent space is empty, this benchmark also implies the throughput of IntentScope in terms of processing intent spaces. In the benchmark, we used the following two intent spaces to evaluate the throughput of implicit intents and explicit intents, respectively. Note that the intersection of an implicit intent space and an explicit intent space is always empty and thus not evaluated.

- i_I: action=`android\.intent\.action\.EDIT`,
  category=`android\.intent\.category\.DEFAULT`,
  scheme=`http`, authority=`\d+`, type=`mpeg`,
  permission=`.*`;
- i_E: component=`com\.sec\..*`, permission=`.*`.

We performed the benchmark in a Xen VM running Ubuntu 14.04 with Intel Xeon E5620 2.4GHz and 8GB of RAM. Only one core was used during the benchmark. Table 6.4 shows the average results of 10 runs. It took approximately 5 minutes to check the customized Android OS of the Galaxy Note loaded with 311 apps, and less than 1 minute to check the others. In general, the processing time is proportional to the number of edges. As shown in Table 6.4, IntentScope processed 2,541 implicit intent spaces and 7,225 explicit intent spaces in a second. While explicit intent spaces were almost three times faster than implicit intent spaces, we note that an explicit intent spaces has only two dimensions and an implicit intent space has six dimensions.

6.5 Discussion

Policy analysis and app analysis. In terms of providing insights for configuring security extensions, our intent space based policy analysis complements existing static and dynamic app analysis. We make this argument based on the fact that an app's runtime behaviors on a specific mobile device are shaped by (1) the app whose code specifies its executional semantics; and (2) the security extensions whose policies specify how the app's specific behaviors are restricted. While we admit that app analysis is indispensable, we also note the alarming trend of malware thwarting app analysis. For example, code obfuscation and encryption hide an app's true semantics from static analysis. "Split personalities" in apps [32, 80] make malware appear innocent by detecting and evading dynamic analysis tools. To get an upper hand against adversaries, we would need policy analysis to orchestrate security extensions for an additional line of defense.

Generality of intent space analysis. While we presented intent space analysis for checking intent-based communication, the underlying methodology is beyond the scope of intents and generally applicable to other security extensions. A promising target is SE Android [109], which controls almost every inter-application communication mechanism other than intent-based communication. Specifically, it checks an attribute called *security context* when an app requests to access files, sockets and so on. Given that security contexts and intent attributes are essentially *access control labels* [50], we foresee that our intent space analysis can be extended to a "context space analysis" for SE Android. For our future work, we will extend our framework to reason about SE Android policies and further maximize the coverage of inter-application communication.

Usability of the holistic reachability graph. As we focused on developing the intent space model and implementing a prototype of IntentScope, usability of the reachability graph was not the primary goal. While the current graph already supports network analysis and flexible queries as shown in the evaluation, we believe that the usability of the graph has a lot of space to improve and indeed this is an exciting area to explore. For example, the proper visualization can assist a security analyst in understanding the inter-application communication and in ultimately developing a robust security policy. Parallel processing on the graph can be introduced to further speed up queries.

6.6 Related Work

Static and dynamic app analysis. App-oriented analysis provides insights for a policy analyst to create appropriate security policies. ComDroid [42] and CHEX [87] statically vet apps for the components that are vulnerable to intent-based attacks. Woodpecker [66] employs an inter-procedural static analysis to discover similar vulnerabilities but specific to stock apps created by device vendors. Epicc [92], AmanDroid [114], FlowDroid [27], and DroidSafe [65] statically discover information flows that potentially leak sensitive data. Beyond static analysis, dynamic runtime solutions reveal how apps communicate through intents in real time. IPC Inspection [60] automatically reduces an intent sender's effective permissions to mitigate unauthorized privilege escalations. QUIRE [48] provides provenance of intents so that a callee can track down the original caller. XManDroid [35] maintains a system-centric call graph for the intents that have been sent and received. TaintDroid [53] and VetDroid [123] track sensitive data shared among apps, regardless of how the data is shared though intents or other inter-application communication mechanisms. Along these lines, our intent space analysis assists policy analysts by systematically analyzing how security extensions confine apps' behaviors.

Experimental security extensions for Android: Besides intent filters, permissions, and IntentFirewall covered in this work, previous research has proposed a series of experimental security extensions for Android. Saint [95] and TISSA [129] support policy-driven access control for intents. CRePe [44] and APEX [91] enable context-aware and fine-grained permissions. FlaskDroid [37] and SE Android [109] are generic and flexible MAC systems that provide comprehensive protection on both

Android's middleware and kernel layers. Aquifer [90] enforces distributed information flow control over intent-based UI workflows. Boxify [31] and DeepDroid [113] enforce security policies on unmodified stock Android. Android Security Module (ASM) [73] and Android Security Framework (ASF) [30] provide programmable interfaces that promote the creation of customized security extensions. IntentScope facilitates defining and verifying security policies for these security extensions. It is especially useful for ASM and ASF that may host security extensions from multiple stakeholders.

6.7 Summary

In this chapter, we have presented intent space analysis for intent-based communication. Intent space analysis is based on an intent space model and a systematic policy checking framework called IntentScope. The intent space model maps a security extension's functionality of forwarding intents as transformation on a geometric space. Based on the intent space model, IntentScope acquires the live states of multiple security extensions and further derives a holistic view that supports formal verification. Also we have described a prototype implementation, along with extensive evaluation results of our approach.

Chapter 7

CONCLUDING REMARKS

7.1 Contributions

Mobile apps may pose security and privacy threats to users. However, existing defensive approaches largely rely on users to evaluate the security implications of apps, but they are too complex for users to take actions. To remedy this situation, we propose to empower users with insights and mechanisms that help them monitor, assess, and confine apps. Toward this direction, we have demonstrated an automated framework for systematically discovering heuristics that enable Android malware to detect the presence of Android emulators. The results imply that some sensitive information assets are overlooked by existing security mechanisms. As a solution, we have proposed a multi-layer security framework that comprehensively and flexibly monitors and confines apps. Based on this framework, we propose a risk management framework that enables automated risk assessment and mitigation. We also propose a holistic assessment framework to check intent-based inter-application communication to discover potential data leakage and app collusion.

REFERENCES

[1] "CVE-2009-1185: Exploid exploit for Android", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185` (2009).

[2] "CVE-2009-2692: Asroot exploit for Android", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2692` (2009).

[3] "Droid2", `http://c-skills.blogspot.com/2010/08/droid2.html` (2009).

[4] "Zimperlich sources", `http://c-skills.blogspot.com/2011/02/zimperlich-sources.html` (2009).

[5] "Defcon 18: These are not the permissions you're looking for", `http://goo.gl/sxHyV` (2010).

[6] "CVE-2011-1149: KillingInTheNameOf exploit for Android", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1149` (2011).

[7] "CVE-2011-1717: Skype stores sensitive user data in files that have weak permissions", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1717` (2011).

[8] "CVE-2011-1823: GingerBreak exploit for Android", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1823` (2011).

[9] "CVE-2011-3874: ZergRush exploit for Android", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3874` (2011).

[10] "Droidbox: An android application sandbox for dynamic analysis", `https://code.google.com/p/droidbox/`, accessed: May 2014 (2011).

[11] "Andrubis: A tool for analyzing unknown android applications", `http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/`, accessed: May 2014 (2012).

[12] "CVE-2012-0056: Mempodipper exploit for Android", `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0056` (2012).

[13] "Security alert: New android malware – dkfbootkit – moves towards the first android bootkit", `http://www.csc.ncsu.edu/faculty/jiang/DKFBootKit/` (2012).

[14] "Security alert: New rootsmart android malware utilizes the gingerbreak root exploit", `http://www.csc.ncsu.edu/faculty/jiang/RootSmart/` (2012).

[15] "CVE-2013-2094: Libperf_event exploit for Android", `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2094` (2013).

[16] "Sophos security threat report 2014 - smarter, shadier, stealthier malware", https://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-security-threat-report-2014.pdf (2013).

[17] "Android developers - using the emulator", http://developer.android.com/tools/devices/emulator.html, accessed: May 2014 (2014).

[18] "Finite automata", http://augeas.net/libfa/, accessed: 06/2015 (2014).

[19] "Genymotion, the fastest android emulator for app testing and presentation", http://genymotion.com, accessed: May 2014 (2014).

[20] "Sanddroid - an apk analysis sandbox", http://sanddroid.xjtu.edu.cn/, accessed: May 2014 (2014).

[21] "Tracedroid - dynamic android app analysis (by vu amsterdam)", http://tracedroid.few.vu.nl/, accessed: May 2014 (2014).

[22] "Wikipedia: List of most downloaded android applications", http://en.wikipedia.org/wiki/List_of_most_downloaded_Android_applications, accessed: 06/2015 (2014).

[23] "Bound services - android developers", http://developer.android.com/guide/components/bound-services.html, accessed: 06/2015 (2015).

[24] "Requesting a shared file - android developers", http://developer.android.com/training/secure-file-sharing/request-file.html, accessed: 06/2015 (2015).

[25] "Selinux wiki", http://selinuxproject.org/page/NB_SEforAndroid_1, accessed: 06/2015 (2015).

[26] Alberts, C., A. Dorofee, J. Stevens and C. Woody, "Introduction to the octave approach", Pittsburgh, PA, CMU (2003).

[27] Arzt, S., S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps", in "ACM SIGPLAN Notices", vol. 49, pp. 259–269 (2014).

[28] Association, G. *et al.*, "Imei allocation and approval guidelines", vol. 10 (2010).

[29] Au, K. W. Y., Y. F. Zhou, Z. Huang and D. Lie, "Pscout: analyzing the android permission specification", in "Proceedings of the ACM conference on Computer and communications security", pp. 217–228 (ACM, 2012).

[30] Backes, M., S. Bugiel, S. Gerling and P. von Styp-Rekowsky, "Android Security Framework: Extensible multi-layered access control on Android", in "Proceedings of the Annual Computer Security Applications Conference", (ACM, 2014).

[31] Backes, M., S. Bugiel, C. Hammer, O. Schranz and P. von Styp-Rekowsky, “Boxify: Full-fledged app sandboxing for stock android”, in “Proceedings of the USENIX Security Symposium”, (2015).

[32] Balzarotti, D., M. Cova, C. Karlberger, C. Kruegel, E. Kirda and G. Vigna, “Efficient detection of split personalities in malware”, in “Proceedings of Network and Distributed System Security Symposium”, (2010).

[33] Barrera, D., J. Clark, D. McCarney and P. van Oorschot, “Understanding and improving app installation security mechanisms through empirical analysis of android”, (2012).

[34] Barrera, D., H. G. Kayacik, P. C. van Oorschot and A. Somayaji, “A methodology for empirical analysis of permission-based security models and its application to android”, in “Proceedings of the 17th ACM conference on Computer and communications security”, pp. 73–84 (ACM, 2010).

[35] Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi and B. Shastry, “Towards taming privilege-escalation attacks on android”, in “Proceedings of the Symposium on Network and Distributed System Security”, (2012).

[36] Bugiel, S., L. Davi, A. Dmitrienko, S. Heuser, A. Sadeghi and B. Shastry, “Practical and lightweight domain isolation on android”, in “Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices”, pp. 51–62 (ACM, 2011).

[37] Bugiel, S., S. Heuser and A.-R. Sadeghi, “Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies”, in “Proceedings of the USENIX Security Symposium”, (USENIX, 2013).

[38] Bugiel, S., S. Heuser and A.-R. Sadeghi, “Flexible and fine-grained mandatory access control on android for diverse security and privacy policies”, in “Proceedings of the USENIX Security Symposium. USENIX”, (2013).

[39] Chen, K. Z., N. Johnson, V. D’Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard and D. Song, “Contextual policy enforcement in android applications with permission event graphs”, (2013).

[40] Chen, Y., H. Xu, Y. Zhou and S. Zhu, “Is this app safe for children?: a comparison study of maturity ratings on android and ios applications”, in “Proceedings of the international conference on World Wide Web”, pp. 201–212 (International World Wide Web Conferences Steering Committee, 2013).

[41] Chia, P. H., Y. Yamamoto and N. Asokan, “Is this app safe?: a large scale study on application permissions and risk signals”, in “Proceedings of the international conference on World Wide Web”, pp. 311–320 (ACM, 2012).

[42] Chin, E., A. Felt, K. Greenwood and D. Wagner, “Analyzing inter-application communication in android”, in “Proceedings of the 9th international conference on Mobile systems, applications, and services”, pp. 239–252 (ACM, 2011).

[43] Chin, E., A. P. Felt, V. Sekar and D. Wagner, “Measuring user confidence in smartphone security and privacy”, in “Proceedings of the Eighth Symposium on Usable Privacy and Security”, (ACM, 2012).

[44] Conti, M., V. Nguyen and B. Crispo, “Crepe: Context-related policy enforcement for android”, Information Security pp. 331–345 (2011).

[45] Davis, B. and H. Chen, “Retroskeleton: Retrofitting android apps”, in “Proceeding of the 11th annual international conference on Mobile systems, applications, and services”, pp. 181–192 (ACM, 2013).

[46] Davis, B., B. Sanders, A. Khodaverdian and H. Chen, “I-arm-droid: A rewriting framework for in-app reference monitors for android applications”, Mobile Security Technologies (2012).

[47] Dharmdasani, H., “Android.hehe: Malware now disconnects phone calls”, `http://www.fireeye.com/blog/technical/2014/01/android-hehe-malware-now-disconnects-phone-calls.html`, accessed: May 2014 (2014).

[48] Dietz, M., S. Shekhar, Y. Pisetsky, A. Shu and D. Wallach, “Quire: Lightweight provenance for smart phone operating systems”, in “Proceedings of the USENIX Security Symposium”, (2011).

[49] Eckersley, P., “Google removes vital privacy feature from android, claiming its release was accidental”, `https://www.eff.org/deeplinks/2013/12/google-removes-vital-privacy-features-android-shortly-after-adding-them`, accessed: 02/2014 (2013).

[50] Efstathopoulos, P., M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek and R. Morris, “Labels and event processes in the asbestos operating system”, in “ACM SIGOPS Operating Systems Review”, vol. 39, pp. 17–30 (ACM, 2005).

[51] Enck, W., “Defending users against smartphone apps: Techniques and future directions”, Information Systems Security pp. 49–70 (2011).

[52] Enck, W., P. Gilbert, 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”, in “Proceedings of the USENIX conference on Operating systems design and implementation”, pp. 1–6 (USENIX, 2010).

[53] Enck, W., 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 Transactions on Computer Systems (TOCS) **32**, 2, 5 (2014).

[54] Enck, W., D. Octeau, P. McDaniel and S. Chaudhuri, “A study of android application security”, in “Proceedings of the USENIX conference on Security”, (USENIX Association, 2011).

[55] Enck, W., D. Octeau, P. McDaniel and S. Chaudhuri, “A study of android application security”, in “Proceedings of the 20th USENIX conference on Security”, SEC’11, pp. 21–21 (USENIX Association, Berkeley, CA, USA, 2011), URL `http://dl.acm.org/citation.cfm?id=2028067.2028088`.

[56] Enck, W., M. Ongtang and P. McDaniel, “On lightweight mobile phone application certification”, in “Proceedings of the ACM Conference on Computer and Communications Security”, pp. 235–245 (ACM, 2009).

[57] F-Secure, “Trojan:android/pincer.a”, `http://www.f-secure.com/weblog/archives/00002538.html`, accessed: May 2014 (2013).

[58] Felt, A., E. Chin, S. Hanna, D. Song and D. Wagner, “Android permissions demystified”, in “Proceedings of the ACM Conference on Computer and Communications Security”, pp. 627–638 (ACM, 2011).

[59] Felt, A., M. Finifter, E. Chin, S. Hanna and D. Wagner, “A survey of mobile malware in the wild”, in “Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices”, pp. 3–14 (ACM, 2011).

[60] Felt, A., H. Wang, A. Moshchuk, S. Hanna and E. Chin, “Permission re-delegation: Attacks and defenses”, (2011).

[61] Felt, A. P., S. Egelman, M. Finifter, D. Akhawe, D. Wagner *et al.*, “How to ask for permission”, in “Proceddings of the USENIX Workshop on Hot Topics in Security”, (2012).

[62] Felt, A. P., E. Ha, S. Egelman, A. Haney, E. Chin and D. Wagner, “Android permissions: User attention, comprehension, and behavior”, in “Proceedings of the Eighth Symposium on Usable Privacy and Security”, p. 3 (ACM, 2012).

[63] Garfinkel, T., K. Adams, A. Warfield and J. Franklin, “Compatibility is not transparency: Vmm detection myths and realities”, in “Proceedings of USENIX Workshop on Hot Topics in Operating Systems”, (2007).

[64] Gilbert, P., B. Chun, L. Cox and J. Jung, “Automating privacy testing of smartphone applications”, (2011).

[65] Gordon, M. I., D. Kim, J. Perkins, L. Gilham, N. Nguyen and M. Rinard, “Information-flow analysis of android applications in droidsafe”, in “Proceedings of the Symposium on Network and Distributed System Security”, (2015).

[66] Grace, M., Y. Zhou, Z. Wang and X. Jiang, “Systematic detection of capability leaks in stock android smartphones”, in “Proceedings of the 19th Annual Symposium on Network and Distributed System Security”, (2012).

[67] Grace, M., Y. Zhou, Q. Zhang, S. Zou and X. Jiang, “Riskranker: scalable and accurate zero-day android malware detection”, in “Proceedings of the international conference on Mobile systems, applications, and services”, pp. 281–294 (ACM, 2012).

[68] Group, L. S., "Zero-permission android applications", `http://www.leviathansecurity.com/blog/zero-permission-android-applications/` (2012).

[69] Hao, H., V. Singh and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android", in "Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security", pp. 25–36 (ACM, 2013).

[70] Harada, T., T. Horie and K. Tanaka, "Task oriented management obviates your onus on linux", in "Linux Conference", (2004).

[71] Herbrich, R., T. Graepel and K. Obermayer, "Large margin rank boundaries for ordinal regression", Advances in Neural Information Processing Systems pp. 115–132 (1999).

[72] Heuser, S., A. Nadkarni, W. Enck and A.-R. Sadeghi, "Asm: A programmable interface for extending android security", in "Proceedings of the USENIX Security Symposium", (2014).

[73] Heuser, S., A. Nadkarni, W. Enck and A.-R. Sadeghi, "Asm: A programmable interface for extending android security", in "Proc. 23rd USENIX Security Symposium (SEC'14)", (2014).

[74] Ho, T. K., "The random subspace method for constructing decision forests", IEEE Transactions on Pattern Analysis and Machine Intelligence **20**, 8, 832–844 (1998).

[75] Hopcroft, J. E., *Introduction to automata theory, languages, and computation* (Pearson Education, 1979).

[76] Hornyack, P., S. Han, J. Jung, S. Schechter and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications", in "Proceedings of the ACM conference on Computer and communications security", pp. 639–652 (ACM, 2011).

[77] Jeon, J., K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster and T. Millstein, "Dr. android and mr. hide: fine-grained permissions in android applications", in "Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices", pp. 3–14 (ACM, 2012).

[78] Jing, Y., G.-J. Ahn, Z. Zhao and H. Hu, "Riskmon: continuous and automated risk assessment of mobile applications", in "Proceedings of the 4th ACM conference on Data and application security and privacy", pp. 99–110 (ACM, 2014).

[79] Jing, Y., G.-J. Ahn, Z. Zhao and H. Hu, "Towards automated risk assessment and mitigation of mobile application", IEEE Transactions on Dependable and Secure Computing **PP** (2014).

[80] Jing, Y., Z. Zhao, G.-J. Ahn and H. Hu, "Morpheus: automatically generating heuristics to detect android emulators", in "Proceedings of the 30th Annual Computer Security Applications Conference", pp. 216–225 (ACM, 2014).

[81] Joachims, T., "Optimizing search engines using clickthrough data", in "Proceedings of the ACM international conference on Knowledge discovery and data mining", pp. 133–142 (ACM, 2002).

[82] Karakashian, S., "An Implementation of An Algorithm for Generating All Connected Subgraphs of a Fixed Size", Software (Version Oct2010), Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, NE (2010).

[83] Kazemian, P., G. Varghese and N. McKeown, "Header space analysis: Static checking for networks.", in "NSDI", pp. 113–126 (2012).

[84] Krosnick, J. A. and D. F. Alwin, "An evaluation of a cognitive theory of response-order effects in survey measurement", Public Opinion Quarterly **51**, 2, 201–219 (1987).

[85] Li, N., Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo and D. Lin, "Access control policy combining: theory meets practice", in "Proceedings of the 14th ACM symposium on Access control models and technologies", pp. 135–144 (ACM, 2009).

[86] Li, Z., M. Sanghi, Y. Chen, M.-Y. Kao and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience", in "Proceedings of the IEEE Symposium on Security and Privacy", pp. 15–pp (IEEE, 2006).

[87] Lu, L., Z. Li, Z. Wu, W. Lee and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities", in "Proceedings of the 2012 ACM conference on Computer and communications security", pp. 229–240 (ACM, 2012).

[88] Matenaar, F. and P. Schulz, "Detecting android sandboxes", `http://dexlabs.org/blog/btdetect`, accessed: May 2014 (2012).

[89] Muthukumaran, D., T. Jaeger and V. Ganapathy, "Leveraing 'choice' in authorization hook placement", in "19th ACM Conference on Computer and Commumications Security", (2012).

[90] Nadkarni, A. and W. Enck, "Preventing accidental data disclosure in modern operating systems", in "Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security", pp. 1029–1042 (ACM, 2013).

[91] Nauman, M., S. Khan and X. Zhang, "Apex: Extending android permission model and enforcement with user-defined runtime constraints", in "Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security", pp. 328–332 (ACM, 2010).

[92] Octeau, D., P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis", in "Proceedings of the 22nd USENIX Security Symposium", (Citeseer, 2013).

[93] Ongtang, M., K. Butler and P. McDaniel, "Porscha: Policy oriented secure content handling in android", in "Proceedings of the Annual Computer Security Applications Conference", pp. 221–230 (ACM, 2010).

[94] Ongtang, M., S. McLaughlin, W. Enck and P. McDaniel, "Semantically rich application-centric security in android", in "Computer Security Applications Conference, 2009. ACSAC'09. Annual", pp. 340–349 (Ieee, 2009).

[95] Ongtang, M., S. McLaughlin, W. Enck and P. McDaniel, "Semantically rich application-centric security in android", Security and Communication Networks **5**, 6, 658–673 (2012).

[96] Page, L., S. Brin, R. Motwani and T. Winograd, "The pagerank citation ranking: Bringing order to the web.", (1999).

[97] Pandita, R., X. Xiao, W. Yang, W. Enck and T. Xie, "Whyper: Towards automating risk assessment of mobile applications", in "Proceedings of the USENIX conference on Security symposium", (USENIX Association, 2013).

[98] Peles, O. and R. Hay, "One class to rule them all: 0-day deserialization vulnerabilities in android", in "9th USENIX Workshop on Offensive Technologies (WOOT 15)", (2015).

[99] Peng, H., C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru and I. Molloy, "Using probabilistic generative models for ranking risks of android apps", in "Proceedings of the ACM conference on Computer and communications security", pp. 241–252 (ACM, 2012).

[100] Petsas, T., G. Voyatzis, E. Athanasopoulos, M. Polychronakis and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware", in "Proceedings of the European Workshop on System Security", p. 5 (ACM, 2014).

[101] Rabin, M., "Risk aversion and expected-utility theory: A calibration theorem", Econometrica **68**, 5, 1281–1292 (2000).

[102] Rasthofer, S., S. Arzt and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks", in "Proceedings of the Network and Distributed System Security Symposium", (2014).

[103] Rastogi, V., Y. Chen and W. Enck, "Appsplayground: automatic security analysis of smartphone applications", in "Proceedings of the ACM conference on Data and application security and privacy", pp. 209–220 (ACM, 2013).

[104] Reddy, N., J. Jeon, J. Vaughan, T. Millstein and J. Foster, "Application-centric security policies on unmodified android", UCLA Computer Science Department, Tech. Rep **110017** (2011).

[105] Reina, A., A. Fattori and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors", in "Proceedings of the European Workshop on System Security", (2013).

[106] Sarma, B. P., N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru and I. Molloy, "Android permissions: a perspective combining risks and benefits", in "Proceedings of the 17th ACM symposium on Access Control Models and Technologies", pp. 13–22 (ACM, 2012).

[107] Schulz, P., "Android emulator detection by observing low-level caching behavior", `https://bluebox.com/technical/android-emulator-detection-by-observing-low-level-caching-behavior/`, accessed: May 2014 (2013).

[108] Sheehan, P. J., B. Davis and H. Chen, "Rewriting an android app using retroskeleton", in "Proceeding of the 11th annual international conference on Mobile systems, applications, and services", pp. 483–484 (ACM, 2013).

[109] Smalley, S. and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android.", in "Proceedings of the Symposium on Network and Distributed System Security", (2013).

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

[111] Stoneburner, G., A. Goguen and A. Feringa, "Risk management guide for information technology systems", Nist special publication **800**, 30, 800–30 (2002).

[112] Vidas, T. and N. Christin, "Evading android runtime analysis via sandbox detection", in "Proceedings of the ACM Symposium on Information, Computer and Communications Security", (ACM, 2014).

[113] Wang, X., K. Sun, Y. Wang and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices", in "Proceedings of the Symposium on Network and Distributed System Security", (2015).

[114] Wei, F., S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps", in "Proceedings of the ACM Conference on Computer and Communications Security", pp. 1329–1341 (ACM, 2014).

[115] Wijesekera, P., A. Baokar, A. Hosseini, S. Egelman, D. Wagner and K. Beznosov, "Android permissions remystified: A field study on contextual integrity", in "Proceedings of the USENIX Security Symposium", (USENIX Association, 2015).

[116] Wilkinson, D. M., “Strong regularities in online peer production”, in “Proceedings of the 9th ACM conference on Electronic commerce”, pp. 302–309 (ACM, 2008).

[117] Wright, C., C. Cowan, S. Smalley, J. Morris and G. Kroah-Hartman, “Linux security modules: General security support for the linux kernel”, in “Proceedings of the 11th USENIX Security Symposium”, vol. 5 (San Francisco, CA, 2002).

[118] Wu, C., Y. Zhou, K. Patel, Z. Liang and X. Jiang, “Airbag: Boosting smartphone resistance to malware infection”, in “Proceedings of the Network and Distributed System Security Symposium”, (2014).

[119] Xu, R., H. Saïdi and R. Anderson, “Aurasium: Practical policy enforcement for android applications”, in “Proceedings of the USENIX Security Symposium”, (2012).

[120] Yagemann, C., “Intent firewall”, `http://www.cis.syr.edu/~wedu/android/IntentFirewall/index.html`, accessed: 06/2015 (2014).

[121] Yan, L. K. and H. Yin, “Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis”, in “Proceedings of the 21st USENIX Security Symposium”, (2012).

[122] Yang, Z., M. Yang, Y. Zhang, G. Gu, P. Ning and X. S. Wang, “Appintent: Analyzing sensitive data transmission in android for privacy leakage detection”, in “Proceedings of the ACM conference on Computer and communications security”, pp. 1043–1054 (ACM, 2013).

[123] Zhang, Y., M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang and B. Zang, “Vetting undesirable behaviors in android apps with permission use analysis”, in “Proceedings of the ACM conference on Computer and communications security”, pp. 611–622 (ACM, 2013).

[124] Zhou, X., S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter and K. Nahrstedt, “Identity, location, disease and more: Inferring your secrets from android public resources”, in “Proceedings of the ACM Conference on Computer and Communications Security”, (2013).

[125] Zhou, X., Y. Lee, N. Zhang, M. Naveed and X. Wang, “The peril of fragmentation: Security hazards in android device driver customizations”, in “Proceedings of the IEEE Symposium on Security and Privacy”, (IEEE, 2014).

[126] Zhou, Y. and X. Jiang, “Dissecting android malware: Characterization and evolution”, in “Proceedings of the IEEE Symposium on Security and Privacy”, pp. 95–109 (IEEE, 2012).

[127] Zhou, Y. and X. Jiang, “Dissecting android malware: Characterization and evolution”, in “Proceedings of the 2012 IEEE Symposium on Security and Privacy”, pp. 95–109 (IEEE, 2012).

[128] Zhou, Y., Z. Wang, W. Zhou and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets”, in “Proceedings of the Network and Distributed System Security Symposium”, pp. 5–8 (2012).

[129] Zhou, Y., X. Zhang, X. Jiang and V. Freeh, “Taming information-stealing smartphone applications (on android)”, Trust and Trustworthy Computing pp. 93–107 (2011).