

SYSTEM CALL DEPENDENCE GRAPH BASED BEHAVIOR DECOMPOSITION OF ANDROID APPLICATIONS

Bin Zhao

JD.com Silicon Valley R&D Center, Mountain View, California, USA

ABSTRACT

Millions of developers and third-party organizations have flooded into the Android ecosystem due to Android's open-source feature and low barriers to entry for developers. However, that also attracts many attackers. Over 90 percent of mobile malware is found targeted on Android. Though Android provides multiple security features and layers to protect user data and system resources, there are still some over-privileged applications in Google Play Store or third-party Android app stores at wild. In this paper, we proposed an approach to map system level behavior and Android APIs, based on the observation that system level behaviors cannot be avoided but sensitive Android APIs could be evaded. To the best of our knowledge, our approach provides the first work to decompose Android application behaviors based on system-level behaviors. We then map system level behaviors and Android APIs through System Call Dependence Graphs. The study also shows that our approach can effectively identify potential permission abusing, with an almost negligible performance impact.

KEYWORDS

Behavior Representation, System Call Dependence Graph, Privilege Escalation, Android APIs

1. INTRODUCTION

In the past ten years, there is a huge booming the global market for mobile devices, especially smartphones. In the fourth quarter of 2011, smartphone sales outpaced PC sales for the first time ever, and it was not even close, reported by Canalys[1]. Another milestone for smartphones is that smartphone sales have surpassed those of feature phones in early 2013 for the first time and will continue the dominance over feature phone sales [2]. Of all smartphone sales, Google's Android accounted for 86.8% in the third quarter of 2018 (3Q18), reported by IDC [3]. In the near seen future, Android will continue to dominate the smartphone market [4].

Due to Android's open-source feature and low barriers to entry for developers, millions of developers and third-party organizations have been attracted to the Android ecosystem. With more than 2 million apps and 20 billion downloads in Q3 2018, Google Play now acts as the engine of the application ecosystem[5]. Meanwhile, there are many public application markets for installing Android applications thanks to its leniency policy on customizations and extensions. However, it has drawn the numerous attentions of attackers and malicious apps. White House estimated that malicious cyber activity cost the U.S. economy between \$57 billion and \$109 billion in 2016 [6], while mobile malware plays an important role in these attacks. Not surprisingly, over 90% are found targeted on Android [7]. The total global number of malicious Android applications has risen steadily in the last 5 years. In 2013, just over a half-million samples were malicious. By 2015 it had risen to just under 2.5 million. For 2017, the number is up to nearly 3.5 million, as reported by SophosLabs in 2018 [8].

A malicious application can harm a user's mobile device in various ways. A Trojan malware can steal user's data like contact list and email addresses, hijack a user's device resources, and prevent user from performing some actions, etc. Spyware can stealthily collect data regarding user's behaviors and send those data to a remote server. Ransomware can deny access to your mobile phone or data until a ransom is paid, which could be devastating to an individual [9].

Android provides multiple security features to achieve the goal of protecting user data and system resources and providing application isolation. Essentially, those features can be divided into two layers: system and kernel-level security, and application-level security [10, 11]. System and kernel-level security is provided and ensured primarily by the Linux kernel. Specifically, a new inter-process communication (IPC) mechanism is provided by kernel to ensure the security when different applications communicate with each other [11]. These features contribute to the process isolation and application sandboxing. The application level security includes the android permission model, application signing and verification, etc. Android permission model provides "additional finer-grained security features through a 'permission' mechanism that enforces restrictions on specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data" [12]. However, there are still security problems with this finer-grained android permission model. Norton's Joe Keehnast said "very few people actually look through an app's permissions before installing it" [13]. For common users, it could be very unclear about the permissions listed. For example, an application may request for the permission of data connection; however, little is known about what it is using the connection for and when the connection is used. Besides confusion, what is worse is the inherent vulnerabilities with this permission model. Android user-based permission model is per-process based. Android is implemented based on Linux kernel which adopts the principle of the privilege-based security scheme. The purpose is to isolate user resources one another. Once a user has permission (privilege) granted during installation, checks against the user will not be done during running [12]. There is no context-awareness checking for the information flow during run time.

Android APIs and permissions play an important role in the Android security system. The Permission system is achieved via calling Android API and declaring permissions in the application manifest file, though permission check is done in the `system_server` process. There are several prior studies on correlating the connection between Android APIs and permissions. For example, a tool called Stowaway built by Adrienne P. Felt *et al.* can determine the set of API calls that an application uses and map those API calls to permissions [14]. PScout, developed by Kathy Au *et al.* [15], could identify Android APIs that could be called by an application and the permissions this API call might need. PScout considers both documented and undocumented APIs. These approaches can find whether an Android application is overprivileged or abusing the permissions, provided that APIs can be effectively tracked. However, Android applications not only use more and more undocumented APIs, but also tend to use their own libraries to evade the Android APIs. A primary reason is that those attacks via calling Android APIs will be detected by tools such as Stowaway or PScout. For example, Z. Zhang *et al.* proposed a transplantation attack which could "spy on users without the Android API auditing being aware of it" [16]. Essentially, this transplantation attack takes out the code from the `system_server` or `mediaserver` process and builds its library to avoid directly calling Android APIs.

Based on this observation, we proposed an approach to map system level behavior and Android APIs. Though Android APIs can be evaded, system-level behavior cannot be avoided given that Linux kernel resources are required such as Camera driver or Binder driver. In general, our approach has three steps. Step 1, obtain an application's behavior via a combination of system-level tracking and symbolic execution. This step will result in some System Call Dependence Graphs (SCDGs). SCDGs are a clear representation of behaviors for an application. Step 2, concurrently with Step 1, obtain all Android APIs called by this application. Step 3, map SCDGs

with Android APIs based on system call entries and timestamps. These steps are done on both benign and malicious applications.

Though this work is not the first to apply system level behavior analysis on Android applications [17-21], this is by far the first attempt to map system-level behaviors with Android APIs. Our goal in this paper is not to thoroughly detect malicious applications which evaded Android APIs. Instead, we intend to build a connection and finally a mapping between permissions an application declares and the invoked system-level behaviors. This is a basis for further study on detecting possible malicious applications. Overall, this work makes the following contributions:

- We systematically employ system-level behavior tracking on Android applications. Their behaviors are dynamically represented by SCDGs.
- To the best of our knowledge, this is the first attempt to map system-level behavior of Android application with Android APIs.
- The mapping between system calls and Android APIs can be used to detect malicious applications which try to evade Android APIs to conduct malicious actions.

The rest of this paper is organized as follows. We first shed light on some related work in Section 2. An introduction to Android system and the Android applications are presented in Section 3, followed by the current issues associated with Android applications in Section 4. The Problem statement is also presented in this section. Our approach of mapping system level behaviors with Android APIs is proposed in Section 5. Section 6 presents a case study to evaluate this approach. Current limitations and some future work are discussed in Section 7. Finally, we conclude this paper in Section 8.

2. RELATED WORK

Many efforts have been employed to address the security of Android applications using Android APIs or system calls. In this paper, we shed light on some work that our study stands upon.

K. Z. Chen *et al.* proposed an approach of contextual policy enforcement to check the interactions between an application and the Android event system [38]. It could detect sensitive operations being performed without the user's awareness. They proposed a new abstraction of Permission Event Graph (PEG) to detect malicious behaviors. This contextual policy is an important complement to the current Android system. It can detect some malicious behaviors in Android applications. However, there are several issues with this approach. Precision and efficiency still need to be improved. There is a scalability issue in identifying applications PEGs in another context [38].

A. Reina *et al.* proposed a tool CopperDroid of system call-centric analysis to reconstruct Android malware behaviors [17]. CopperDroid tracks system calls to characterize low-level and high-level Android-specific behaviors. CopperDroid also considers the path coverage issue. However, CopperDroid is dedicated to reconstructing application behaviors including malware, not identifying malicious Android applications. K. Singh proposed context-sensitive permission model MobileIFC to for hybrid mobile applications [39]. The model "allows applications accessing sensitive user data while preventing them from leaking such data to external entities" [39]. MobileIFC can also achieve the information flow control over user contents. Although it is not intended to identify malicious Android applications, it provides a framework for the context-awareness permission model.

H. Cai *et al.* presented DriodCatus using a diverse set of dynamic features based on method calls and ICC Intents. Those features were obtained from a dynamic characterization study that revealed

behavioral differences between benign and malicious apps in terms of method calls and ICCs [41]. DroidCat achieved relatively high accuracy with an acceptable performance overhead. However, this proposal did not discuss the execution paths that could lead to some potential malicious behaviors. The code coverage issue also could undermine the approach in terms of capturing app behaviors. The defined “diverse” set of features could also have to be consistently maintained and updated regularly.

Y. Aaferet *et al.* proposed an approach *ARCADE* to generate Android API specification, an Access-control Flow Graph (AFG) [42]. It then processes the constructed AFG to extract the access control conditions enforced by the API and their correlations. This mapping was then leveraged to address the over-privilege and component hijacking issues. This approach addresses some path-sensitive issues. However, it has one inherent limitation. The mapping requires “understanding of certain contextual factors an app holds at a specific API invocation” [42]. If these conditions are not inferred correctly, it will result in inaccurate detection results.

A system call tracking based automatic classification tool is presented by Dimjašević M. *et al.* They implemented a tool *MALINE* and performed an empirical evaluation on the applications to detect malicious ones [43]. They have a 93% detection accuracy with a 5% benign application classification error, which is not very bad [43]. However, there are some limitations of this tool. This approach is based on random testing. It is very likely that some malicious applications could evade this tool by hiding or triggering their behaviors under certain circumstances.

3. BACKGROUND: THE ANDROID SYSTEM

Android is an open-source platform implemented primarily based on Linux kernel, and designed mainly for mobile devices. It usually consists of an operating system based on Linux, middleware, application framework, and some essential applications [22], as shown in Fig.1. Linux kernel lies in the lowest layer in the Android system architecture. Besides some traditional features like memory management, security model, network stack and process management that the original Linux kernel supports, power management, and some specific mobile phone related drivers are added into this Linux kernel. Those drivers include binder (IPC) driver, USB gadget driver, and Low Memory Killer, etc. [23]. Android middleware includes native libraries and Android runtime. Native libraries are used to support screen display, multi-media services, and web browser, etc. Android runtime contains a Dalvik virtual machine (VM) and core Java application libraries [22]. Every application runs in its own Dalvik VM. Unlike conventional Java VMs, which are stack machines, Dalvik is register-based VM [22, 24]. In Android, applications are usually written in Java and compiled to Java bytecode (.class files). A dx tool then convert the .class files into *.dex* (Dalvik Executable) format. The reason converting Java code into *.dex* format is that optimization of code is required due to limited resources, such as memory and processor speed [22,24]. The Android application framework is essentially a built-in toolkit and APIs providing services and interfaces to develop applications. It includes the Activity Manager, Package Manager, Windows Manager, Telephony Manager, and Content Providers, etc. All of them can simplify the reuse of components.

3.1. Android Applications

Android applications lie in the top layer of Android system. They are basically Android packages (APK), archive files containing all the content of apps [25]. In Android, each app runs in its own Linux process. Android app comprises four types of app components: activities, services, content providers, and broadcast receivers [25]. An activity is a single screen that users can interact with, such as an activity of composing an email in an email app. Unlike activities, a service has no user interface; rather, it runs in the background, such as playing music while a user composing text

messages. A content provider manages the data among applications. Different content providers manage different types of data, such as contact list, images, and videos. Any app with the proper permissions can interact with the corresponding content providers [22, 25]. A broadcast receiver is used to respond to system-wide broadcast announcements, such as battery is too low [25]. Every app should have an AndroidManifest.xml file [26]. The manifest commonly includes the following information: package name, components of the application, permissions, specific libraries the package need linked to, etc. [22, 26]. Permissions are one of the most important declarations in the manifest to fulfill the principle of least privilege and privilege separation. As aforementioned, permission is a mechanism restricting an application’s access to a part of the code or to data on the device. An Android app usually has to explicitly declare the permissions to access the protected part, such as GPS, Wi-Fi, Bluetooth service, broadcasting service, telephony service, camera, SMS/MMS messages, contacts, calendar, and network/data connections. That access are usually implemented via Android APIs. Permissions requested by an app are granted at app install time. The package installer first retrieves permissions from AndroidManifest.xml and check against the signatures declaring those permissions. If needed, users are then prompted to either allow or deny all permissions the installation package requests. If all granted, the app will be installed successfully. During running time, no checks with the user are done [12].

3.2. Android APIs

A framework API is provided on the Android platform that enables applications to interact with the underlying Android system, including resources and services. According to Android [27], the framework API consists of several sets of APIs: a set of packages and classes (e.g., android.bluetooth package and its BluetoothHeadset class), a set of XML elements and attributes

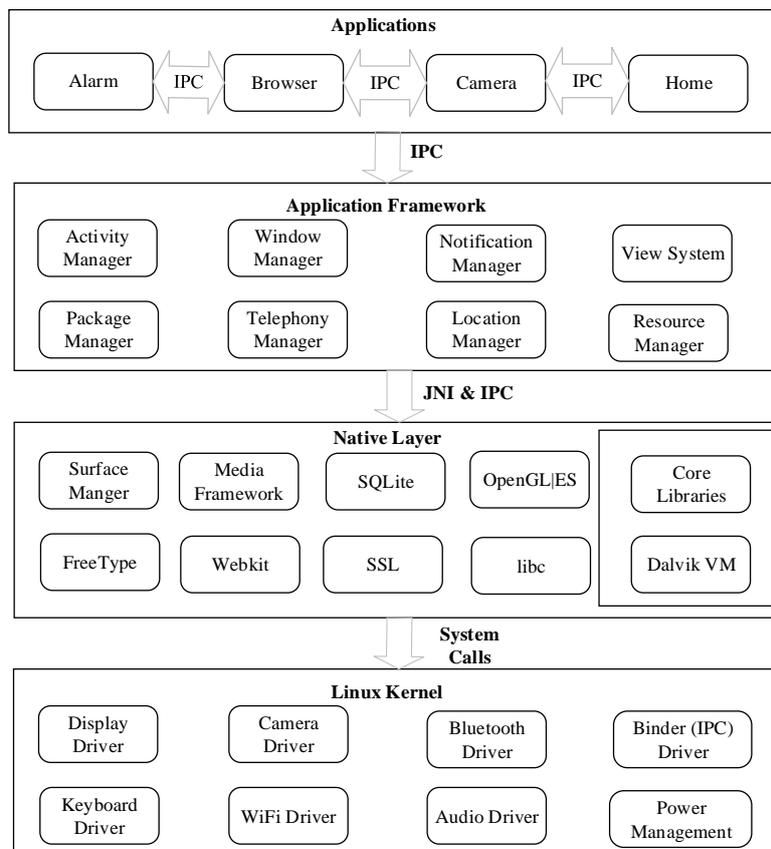


Figure 1. Architecture of Android System

for declaring a manifest file, a set of XML elements and attributes for declaring and accessing resources, a set of intents, and a set of permissions that Android applications can request (e.g., READ_EXTERNAL_STORAGE), as well as permission enforcements included in the Android system.

Android applications and system services are all implemented in Java [15]. The access to the lower level libraries via the framework API. The Android application framework implements and provides these Android APIs. They are responsible for the Core system services (e.g., Activity manager) and hardware services (e.g., Location manager). Fig. 2 shows the interaction between an example application process and the system_server process. For example, if an application X wants to use the GPS service. The activity in application X will first call the framework API implemented in framework.jar. In this case, Context.getSystemService (Context.LOCATION_SERVICE) is called to retrieve a link to android.location.LocationManager for controlling location updates. The Location Manager then get the reference to the Service Manager from the binder. The Location Manager then looks up “location” service in the Service Manager. The Service Manager register a binder context manager, and the references from the binder are obtained. The binder will invoke the LocationManagerService in the system_server process. Permission check is done in the system_server process. If permission of ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION is declared in the android.Manifest.permission, the SystemServer will register “location” service via the Service Manager. The Service Manager finally get the references of the LocationManagerService.

The APIs that an Android application uses can be roughly divided into two categories: documented APIs and undocumented APIs. Documented the Android APIs are released on Android platform by Google Android Developers. These documented Android APIs include Android specific libraries (usually start with android, com.android or dalvik), Java compatibility libraries (start with java or javax), and third party libraries (start with org, primarily Apache libraries). Undocumented APIs are also used by Android application developers since Android is open-source. Developers may use Java reflection or implement their own source code to obtain references to any Java method.

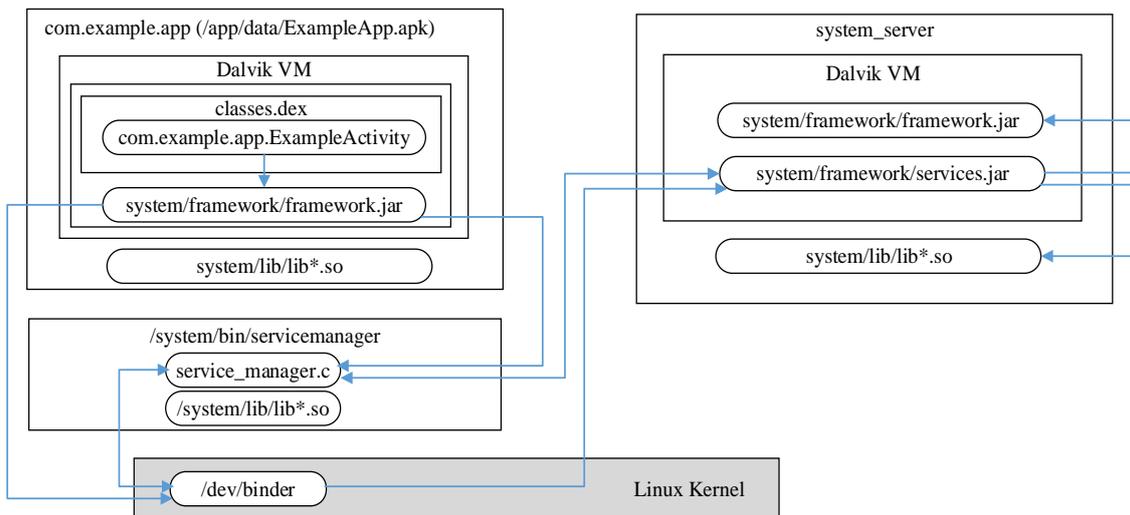


Figure 2. An Example of application calling APIs and binder in Android system. The upper left is an example application process and the upper right is the system_server process.

As mentioned in the introduction, to access sensitive resources, permissions should be granted for applications. However, this should work together with Android APIs. As shown in Fig. 2, the

permission check occurs in the `system_server` process. Kathy Au *et al.* found that there is a mapping between Android API calls and permissions [15]. They identified every Android API that could be called and the permissions this API call might need.

4. ISSUES WITH ANDROID API AUDITING

In this section, we first do a review of existing techniques for detecting malicious applications, including Android API auditing. We then focus on the issues with Android API auditing. Finally, we present the problem statement of this paper.

4.1. Review of Prior Techniques in Detecting Malicious Android Applications

To address permission abusing attacks in Android systems, many works have been presented. Generally, those works can be divided into two primary classes: (C1), static analysis of decompiled applications [17]; and (C2), dynamic analysis to monitor runtime behavior of applications [17, 28]. Static analysis such as DroidRanger [29] and RiskRanker [30] has several strengths. It is relatively easier to decompile Java in Android. Much useful information can be easily obtained in the application's manifest file. In addition, static analysis can properly address the issue of code coverage. However, static analysis is not resilient to obfuscation and compiler optimization techniques. It has a weakness to handle the dynamic code and native code in Android. Sometimes runtime actions cannot be determined unless executed.

Therefore, dynamic analysis such as TaintDroid [28] and DroidScope [31] are aiming to address those weaknesses. They either use instrumentation (TaintDroid) or virtual machine monitor (DroidScope) techniques to conduct the runtime analysis [17]. This technique is resilient to obfuscation and polymorphism. It also properly addresses the runtime action issue. However, code coverage is a big challenge in dynamic analysis. Overhead and scalability can also become a bottleneck for some approaches.

4.2. Issues with Current Android API Auditing

Many prior approaches use Android API auditing to detect malicious applications regardless of static or dynamic analysis [14, 15]. However, Android API auditing can be significantly undermined by some existing attacks, e.g. the transplantation attacks. An essential reason is that once an application evades (intentionally or unintentionally) Android APIs whether by, native libraries or developers' own implementation, Android API auditing will usually not work effectively. Some malicious attacks like in [16] could even evade the binder.

4.3. Problem Statement

Inspired by a transplantation attack on the Android platform and limitations of existing techniques, we present the problem statement of this paper. First, how to obtain the privileges (permissions) an application used? Second, how to represent behavior so that these behaviors can correctly reflect the applications even when these applications evade some Android APIs. This representation should also reflect the functionalities of those applications. Third, how to correlate the permissions of an application with the obtained behaviors? Fourth, can the correlation and representation be used to detect potential permission abuse or privilege escalation?

5. DESIGN AND IMPLEMENTATION

5.1. Approach Rationale

To address the issues, we propose the system-level behavior analysis on Android applications. There are several reasons that we introduce system-level behavior analysis to identify malicious behaviors. First, system calls are the only interface between Android OS and an application, providing the only way for an application to access the OS services. Second, almost every attack goal is bundled with Android OS resources. Hence, for malicious Android applications, it is usually not possible for them to conduct sensitive actions without triggering system calls, even if they use obfuscation or polymorphism techniques [32, 34]. Third, privilege escalation is commonly achieved through IPC mechanism in Android. System calls used by IPC mechanism in Android can be easily identified and tracked. Fourth, system calls can be practically tracked and analyzed, while giving little overhead to the Android OS, given that Android applications are process-based.

However, from a single system call trace, we know little information about the overall behavior of an application, as system calls are low-level reflection about the behavior characteristics of a program. How can we map the low-level system call traces with application level behavioral? We need an intermediate representation to connect them together. This is one of the primary reasons for using SCDGs, as SCDGs can appropriately reflect the dependencies between system calls. They are the abstraction of a sequential system call traces. They can clearly describe the interactions among all the private browsing data and disk operations.

5.2. Behavior Representation of Android Applications

As aforementioned, an application's behavior can be represented by using a particular graph called SCDG, specifically, a set of disconnected SCDGs. Each SCDG is a graph in which "system calls are denoted as vertices, and dependencies between them are denoted as edges" [32, 33, 40]. A SCDG essentially describes the interaction between a running program and the operating system. This interaction is an essential behavior characteristic of the program [32, 40]. In this paper, SCDG is defined as follows [32, 33, 35].

Definition 1. System Call Dependence Graph. Let p be a running application. Let I be the input to p . $f(p, I)$ is the generated system call traces. $f(p, I)$ can be represented by a set of System Call Dependence Graphs (SCDGs) $\cup_{i=0}^n G_i: G_i = \langle N, E, F, \alpha, \beta \rangle$, where

- N is a set of vertices, $n \in N$ is a system call
- E is a set of data dependence edges, $E \subseteq N \times N$
- F is the set of functions $\cup f: x_1, x_2, \dots, x_n \rightarrow y$, where each x_i is a returnvalue of system call, y is the dependence derived by x_i
- α assigns the function f to an argument $a_i \in A$ of a system call
- β is a function assigning attributes to node value

A system call trace contains a system call name, arguments, return value and time, etc. The arguments of a system call are often highly dependent on previous system calls. There are two types of data dependence between system calls. First, a data dependency exists if a system call's argument is derived from the return value(s) of previous system calls. Second, a system call can also be dependent on the arguments of previous system calls [35]. Fig. 3 shows an example of the possible dependencies among system calls of file management [35]. *write* is dependent on *read* as

the input argument of *write* is derived from the return value of *reading*. *read* is also dependent on *write* as the argument of *reading* is sometimes dependent on *write*.

In the definition of SCDG, we mention that α assigns function f to a_i to a system call. Here, f is a function to derive dependencies between system calls. Specifically, for an argument f_{a_i} , f_{a_i} is defined as $f_{a_i}: x_1, x_2, \dots, x_n \rightarrow y$, where x_i denotes the return value or arguments of a previous system call, y represents the dependence between a_i and these return values. If a_i of a system call depends on the return value or arguments of previous system call, an edge is built between these two system calls.

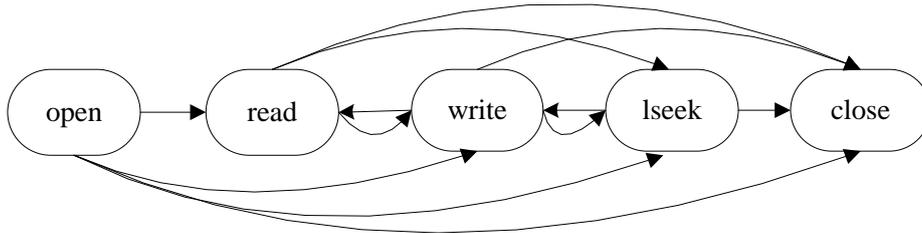


Figure 3. Edges (dependences) and nodes (system calls) in file management.

5.3. Approach Overview

Fig. 4 shows the architecture of our system. It primarily consists of Dynamic Tracer, SCDG Extractor, IPC Analyzer, Android API and Permission Retriever, and Mapping System.

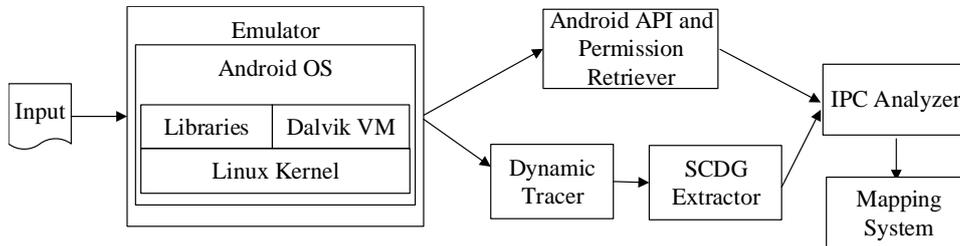


Figure 4. Architecture of mapping system level behavior with Android APIs.

5.3.1. Dynamic Tracer

The dynamic tracer tracks the behaviors of Android applications running on Dalvik VMs in the form of system calls. Android OS is implemented based on the Linux kernel. System call invoking is similar even Android OS is running ARM architecture. Hence, we adopt Dynamic Tracer. A challenge is that the dynamic tracer needs to trace multiple processes, as permission escalation usually involves several applications as described in Section 4. Noise filtering is required to trim system call traces. Another major challenge is the execution path coverage. *strace* is used to intercept system calls as it is a built-in tool in Android SDK [36].

Execution Path Resolving. A key challenge for dynamic tracing is known as the execution path coverage issue. Many dynamic tracing techniques would have to address this issue, or otherwise, the evaluation would be random and a big space would be left for attackers to evade, like in MALINE, proposed by Dimjašević M. *et al.*, using system level behaviors tracking to represent Android applications with random testing.

There are many factors and randomness affecting an application’s inputs. An input (value and event, e.g. finger touch, keyboard input, data read from memory, a network packet, etc.) cannot always be guaranteed to reoccur during a re-execution. As a result, an application will generate a set of different execution paths due to different inputs, while these execution paths cannot be guaranteed to be in the same union set during the dynamic tracing. Likely that certain malicious actions can only be triggered under specific inputs (i.e., conditional expressions are satisfied, or when a certain command is received). If these specific inputs are not included in the test input space, it is possible that malicious actions can be triggered in a particular execution path. Suppose e_i is used to represent the input and e_i as the corresponding execution path. As a result, $\bigcup_{i=0}^n e_i$ is the union set of the execution paths. The purpose is to obtain the largest union set here. In another way, we need to cover as many execution paths as possible.

However, almost none of the prior approaches related to Android applications have taken input space coverage issue into account [42, 43]. Even if this issues was considered, like in [43], they only mentioned this issue and never solved this problem. There is a need to automatically explore the input space of Android applications. Generally, the input space of an Android applications can be divided into two categories: the event space and the value space [35]. Many applications define some event handlers, which may execute in any order as a result of user actions such as clicking buttons or submitting forms. The value range of input includes user data such as form field and text areas, URL and HTTP channels.

An execution path resolver (EPR) is used based on dynamic symbolic execution in this paper. The EPR can be used to “hit” as many execution paths as possible for an application to obtain high coverage for flexible coverage metrics. In the EPR, symbolic variables are tracked instead of the actual values. Values of other variables that depend on symbolic inputs are represented by symbolic formulas over the symbolic inputs. When a symbolic value propagates to the condition of a branch, it can use a constraint solver to generate inputs to the program that would cause the branch to satisfy some new paths [35].

Concrete execution-only follows one program path when execution, while symbolic execution uses symbolic values as inputs and symbolic expressions[44]. Suppose that a list of symbols $\{\xi_1, \xi_2 \dots\}$ are supplied for a new input value of an application each time [35]. Symbolic execution maintains a symbolic state, mapping variables to symbolic expressions, a symbolic path condition pc , and a Boolean expression over the symbolic inputs $\{\xi_i\}$. pc accumulates constraints on the inputs that trigger the execution to follow the associated path.

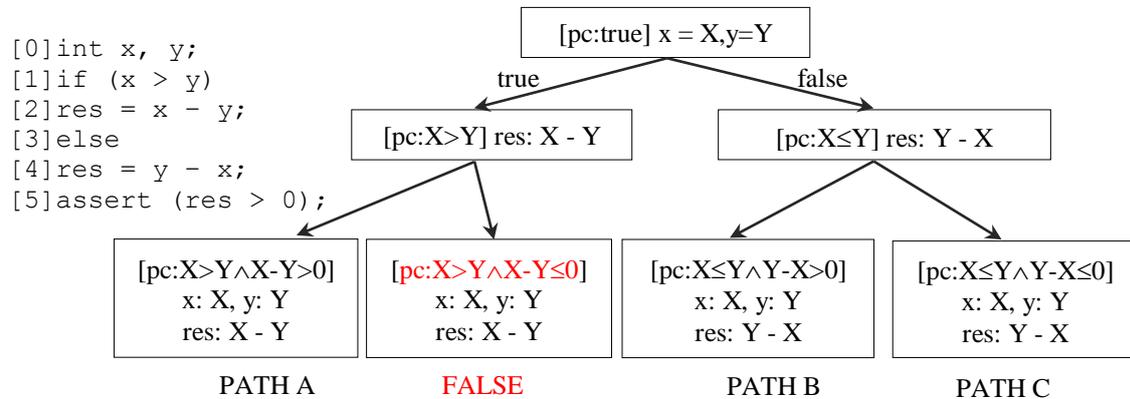


Figure 5. Symbolic execution tree of the corresponding program

Given a conditional *if* (e) S_1 *else* S_2 , pc is updated with assumptions on the inputs to choose between alternative paths. If the new control branch is chosen to be S_1 , pc is updated to $pc \wedge \mu(e) = 0$; otherwise, for S_2 , pc is then updated to $pc \wedge \mu(e) \neq 0$. $\mu(e)$ denotes the symbolic predicate obtained by evaluating e in the symbolic state μ . In symbolic state, both branches can be taken, resulting in two different execution paths. Symbolic execution terminates when pc is not satisfied. Satisfiability is checked with a constraint solver. For each execution path, every satisfying assignment to pc gives values to the input variables that guarantee the concrete execution proceeds along this path. For code containing loops or recursion, one needs to give a limit on the iteration, i.e., a timeout or a limit on the number of paths [35].

Fig. 5 shows a simple example of the symbolic execution tree of a program computing the absolute difference between two integers. Given $x=4$ and $y=3$, concrete execution will only execute the *if* statement branch. Symbolic execution will start with the symbolic value of $x=X$ and $y=Y$. The initial value of pc is true. The symbolic execution tree is shown in Figure 5. As mentioned above, pc is updated with constraints on the inputs in order to choose between alternative paths [35,44]. In line [1] of the *if* condition, unlike the concrete execution, symbolic execution will consider that both alternatives of the *if* statement are possible, and pc is updated accordingly. Symbolic execution terminates when pc is not satisfied like in the path with red "FALSE". As a result, in this example, symbolic execution explores three different feasible paths, with a fourth path not satisfied and terminate, while PATH C will finally result in an assertion violation. As of test case generation, all three feasible path conditions are solved and the solutions are used as test inputs that are guaranteed to execute all the paths through this code. While for concrete execution, it is possible that only one path is taken even with multiple test inputs. Symbolic execution usually guarantees that conditions are all considered.

Specifically, the EPR includes a dynamic symbolic interpreter that performs symbolic execution of Java, a path constraint extractor that builds queries based on the results of symbolic execution, a constraint solver that finds satisfying assignments to those queries, and an input feedback component that uses the results from the constraint solver as new program inputs [35]. They are used to generate values to "hit" as many paths as possible.

5.3.2. SCDG Extractor

The SCDG Extractor here takes the trimmed system call traces as the input and aims to generate SCDGs for applications. It primarily explores the dependencies between system calls. This component has two functions: API and the Permission Retriever. The permission retriever extracts permission list in the corresponding applications. It retrieves the permissions from `AndroidManifest.xml`. The output is a list of vectors $\langle pid, packname, pms_1, pms_2, \dots, pms_n \rangle$, where pid is the process id of the application, $packname$ is the name of the application, and pms_n is the permission. The API retriever can obtain every Android APIs that are called by an application. We only extract the official Android APIs in this part. This component runs concurrently with dynamic tracer and SCDG extractor.

5.3.3. IPC Analyzer

Permission enforcing usually requires the involvement of IPC. Of all the IPC mechanisms, binder acts as the core feature and component. Intents are conveyed via binder. In addition, interactions with the operating system also go through the binder [17]. The interactions with the system are primarily through the system call `ioctl()`. The IPC analyzer takes SCDGs as the input and extracts those with IPCs. Finally, based on the arguments, a mapping of caller/callee is presented. A caller is a component in an application that requests the permission. A callee is a component in another application that has the permission or is the resource that requested.

5.3.4. Mapping System

We use Android's *logcat* as a baseline for mapping system level behavior with Android APIs [37]. All the three use the same time clock and have an accurate timestamp associated with each log, system call entry and API calling.

5.4. Implementation

The trace generator is implemented based on *strace* [36, 40]. It can track the system calls and filter off the unnecessary system calls. We implemented the SCDG extractor under *Valgrind*. The SCDG extractor can construct SCDGs. We use *PScout* as the API and permission retriever [15]. For Java bytecode symbolic execution, Java PathFinder (JPF) is used to generate test suites and obtain high coverage for flexible coverage metrics [44,45]. JPF is an open-source system that executes normal java bytecode programs and generates test cases by means of symbolic execution. One limitation of JPF is that JPF cannot analyze Java native methods. We will discuss more on this issue in Section 7.

6. EVALUATION AND CASE STUDY

As mentioned in the introduction, our goal is not to detect malicious applications using this technique. Instead, we intend to build a mapping between the Android APIs and system-level behavior. In this section, we present a case study to demonstrate the process of this approach. We will also compare our approach with several other methods to demonstrate the advantages and effectiveness of our approach. Although we do not detect malicious applications, our approach can be further used to identify suspicious over-privilege applications.

6.1. Evaluation Environment

Our experiments were performed on a workstation with a 2.40 GHz Quad-core Intel(R) Xeon(R) CPU and 16GB memory, under Fedora 28. We use the emulator of Samsung Galaxy S9 on Android 8.0 as the test mobile phone platform.

6.2. A Case Study on Camera Services in Android

We use the transplantation attack in [16] as a case study to show the process of our approach. Overall, it has three steps, obtaining system level behavior represented in SCDGs, retrieving Android APIs, and mapping system-level behavior with Android APIs.

6.2.1. Camera Service in Android System

Fig. 6 shows the architecture of camera services called by a benign Android application [37]. The camera service is achieved via a native process called mediaserver. The mediaserver contains three layers, native library, HAL and kernel layer (camera driver). A transplantation attack is to transplant the code both in the native library layer and the HAL layer from the mediaserver's address space to the malicious application's address space directly [16]. In this case, after transplantation, this application named CameraTest can access to the Camera service and take photos without a user's consent or even awareness. It can also send those photos taken to a third-party server or its server through the network.

6.2.2. Obtaining System Level Behavior

As every Android application runs in its own process, there is no need to differentiate the system call traces with others. *strace* can simply obtain the system call traces for the transplanted application CameraTest process and the mediaserver process. The following Fig. 7 shows part of the SCDGs obtained from the CameraTest process.

6.2.3. Mapping System Level Behavior with Android APIs

To map the system-level behavior with Android APIs, Android's logcat is introduced in our experiments. Android's logcat collects logs for various applications and portions of the system, where the CameraTest and mediaserver are both included. Based on the timestamps in the system call traces, Android APIs and the logcat, we map the SCDGs to Android APIs. Fig. 8 shows the

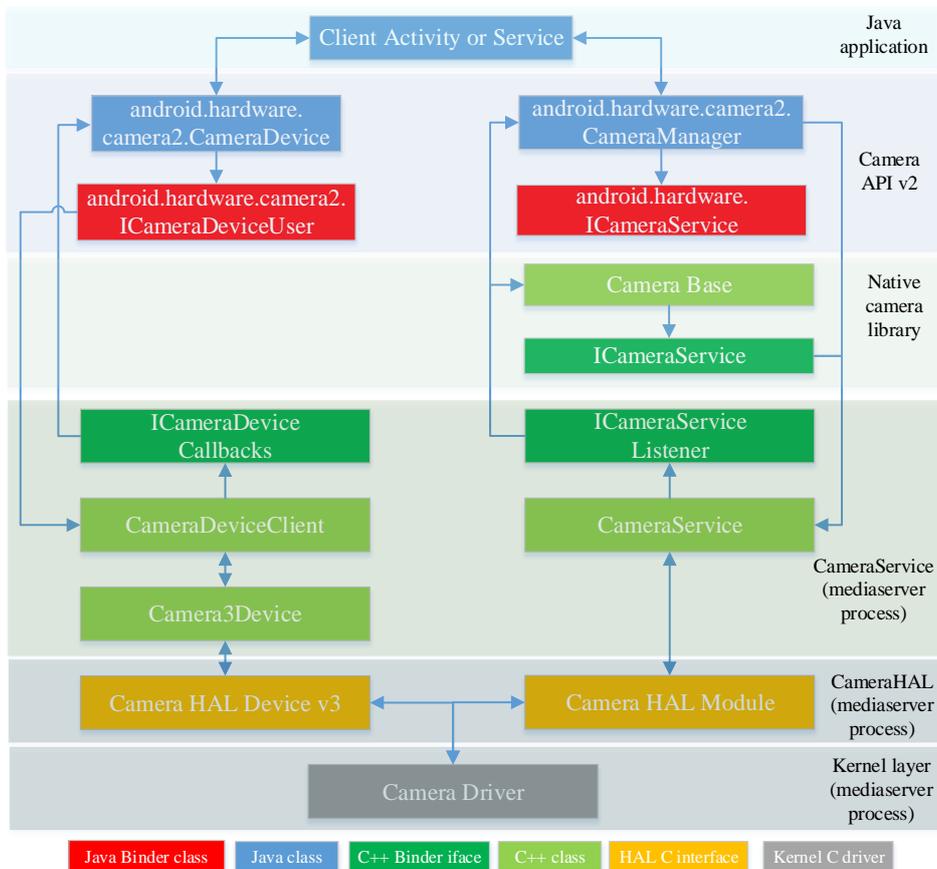


Figure 6. Architecture of camera services in Android system.

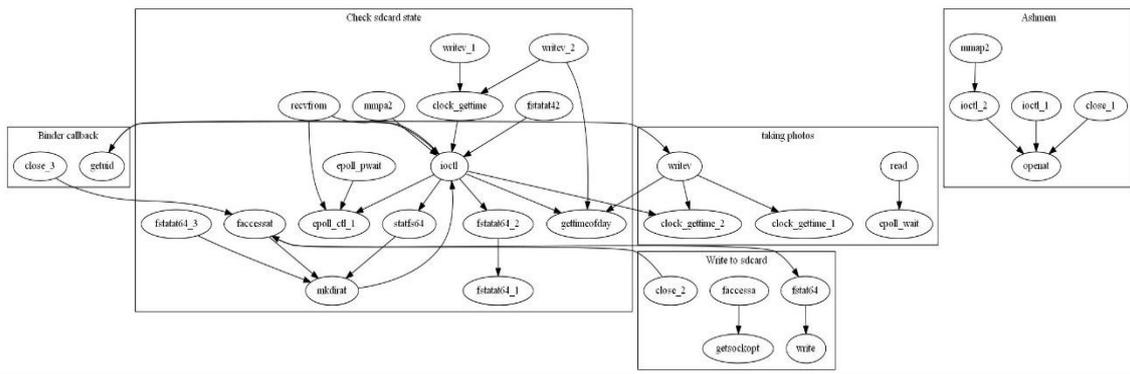


Figure 7. SCDGs obtained from CameraTest process.

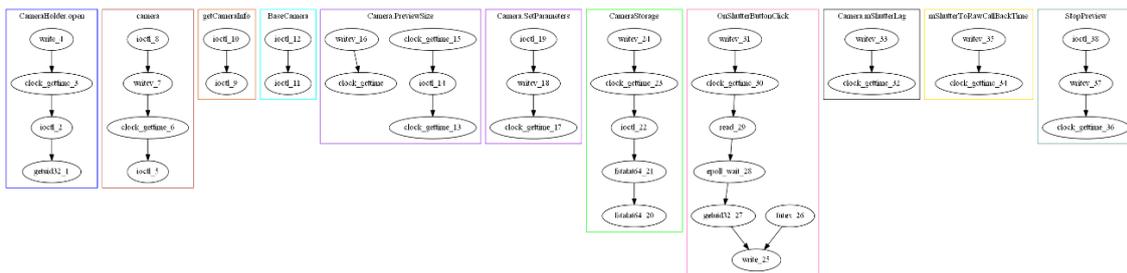


Figure 8. Mapping between SCDGs and Android APIs.

mapping of SCDGs to Android APIs. The mapping is very clean. For example, the CameraStorage APIs consists of just five system calls. It first checks the file status and then uses the binder to implement the IPC interaction. After obtaining the time, it writes to the previously checked folder.

6.2.4. Performance Evaluation

We discuss the performance of our system in two metrics: memory usage and startup time.

Memory Usage. We use Android’s own API to measure the memory usage by calling the *meminfo*. The average memory usage is 1.2% of total memory usage. This memory usage can be considered very small. CPU usage measurement is obtained in Android’s Monitoring. The average CPU usage is 0.9%, which is negligible.

Startup Time. We calculate the average startup time with 10 tests. The results show that there is almost no difference with or without our system.

Overall, our evaluation on memory usage and startup time demonstrate that the system has very little impact on Android’s performance. The overhead brought by the system should not be a concern for users, developers, and app stores.

6.2.5. Comparison with MALINE

MALINE proposed by Dimjašević M. *et al.* also uses system call representation to detect malicious Android applications. Through our experiment, their approach cannot detect CameraTest as a malicious application. MALINE uses random test inputs and did not consider

the execution path resolving issue. MALINE did not consider transplanted attacks either. Our approach can clearly show and identify CameraTest as a malicious application.

7. DISCUSSION

In Section 5, the dynamic tracer is implemented with the execution path resolver. The goal of execution path resolver is to obtain more coverage of execution paths. However, it is infeasible for any resolver to obtain all the coverage. As mentioned in Section 5.4, JPF cannot analyze Java native methods. Additional peer classes or listeners have to be provided or intercepted to test such methods. However, compared with other methods that do not have the execution path resolvers, our approach still outperforms them. On the other hand, behavior stimulation can be used to generate more paths together with symbolic execution [17]. This combined approach might generate more paths and will be the future study of this paper.

Most permission policies are enforced by a binder driver; however, some permission policies can be enforced by other measures. In this paper, currently, both binder and *ashmem* mechanisms are considered. The mapping between the SCDGs and Android APIs can be further used to detect if an Android application is suspicious or not. For example, the SCDGs for an Android API are obtained. If there is a suspicious or even malicious Android application evading the using of a sensitive Android API, through system-level behavior analysis, SCDGs of this application can be retrieved. If those SCDGs are an isomorphism, it can be claimed that this application implements the functionality of this Android API. This method makes it much harder for evading the Android API auditing tools.

8. CONCLUSION

Android provides multi-pronged security features to achieve the goal of protecting user data and system resources and providing application isolation. Android permission mechanism provides a finer-grained security feature to protect user data and system resources. However, permissions can be abused and escalated by malicious applications. We proposed an approach to map system-level behavior with permissions that an application declares. Currently, our study shows that the proposed approach can map system calls with permissions that an application declares, whether benign or malicious applications. It could also identify potential suspicious permission abuse or escalation. However, more experiments are needed to identify malicious applications via investigating the mapping between system calls and Android application permissions.

REFERENCES

- [1] Canalys (2012), "Smart phones overtake client PCs in 2011," <http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011>.
- [2] Gartner (2013), "Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time," <http://www.gartner.com/newsroom/id/2573415>.
- [3] IDC (2018), "Smartphone Market Share," <https://www.idc.com/promo/smartphone-market-share/os>.
- [4] Gartner (2013), "Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013," <http://www.gartner.com/newsroom/id/2408515>.
- [5] Blair, Ian (2019), "Mobile App Download and Usage Statistics," <https://buildfire.com/app-statistics/>.
- [6] The Council of Economic Advisers (2018), "The Cost of Malicious Cyber Activity to the U.S. Economy," <https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf>.

- [7] F-Secure Labs (2013), "Mobile Threat Report Q1 2013,"http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2013.pdf.
- [8] SophosLabs (2018), "SophosLabs 2018 Malware Forecast,"<https://www.sophos.com/en-us/en-us/medialibrary/PDFs/technical-papers/malware-forecast-2018.pdf?la=en>
- [9] CISA (2019), "Ransomware," <https://www.us-cert.gov/Ransomware>
- [10] Smalley, S. and R. Craig (2013) "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in NDSS '13 20th Annual Network and Distributed System Security Symposium.
- [11] Google Android Developers, "Android Security Overview,"<http://source.android.com/devices/tech/security/>.
- [12] Google Android Developers, "System Permissions,"<http://developer.android.com/guide/topics/security/permissions.html>.
- [13] Rodriguez, A. (2012) "Android's Permission Problems," PC World.
- [14] Felt, A. P., E. Chin, S. Hanna, D. Song, and D. Wagner (2011) "Android permissions demystified," In Proceedings of the 18th ACM conference on Computer and communications security (CCS '11). ACM, New York, NY, USA, 627-638. DOI: <https://doi.org/10.1145/2046707.2046779>
- [15] Au, K. W. Y., Y. F. Zhou, Z. Huang, and D. Lie (2012) "PScout: analyzing the Android permission specification," In Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12). ACM, New York, NY, USA, 217-228. DOI: <https://doi.org/10.1145/2382196.2382222>
- [16] Zhang, Z., P. Liu, J. Xiang, J. Jing, and L. Lei (2015) "How Your Phone Camera Can Be Used to Stealthily Spy on You: Transplantation Attacks Against Android Camera Service," in Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 99-110.
- [17] Reina, A., A. Fattori, and L. Cavallaro (2013) "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors," in Proceedings of the 6th European Workshop on System Security (EUROSEC 2013), Prague, Czech Republic.
- [18] Pathak, A., Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang (2011) "Fine-grained Power Modeling for Smartphones Using System Call Tracing," in Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, pp. 153-168.
- [19] Ham, Y. J. and H.-W. Lee (2014) "Detection of Malicious Android Mobile Applications Based on Aggregated System Call Events," International Journal of Computer and Communication Engineering, 3.
- [20] Karami, M., M. Elsabagh, P. Najafiborazjani, and A. Stavrou (2013) "Behavioral Analysis of Android Applications Using Automated Instrumentation," in Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion, SERE-C '13, IEEE Computer Society, Washington, DC, USA, pp. 182-187.
- [21] Liu, J., S. Liu, P. Liu, X. Zhou, and B. Zhao (2013) "A new decision support model in multi-criteria decision making with intuitionistic fuzzy sets based on risk preferences and criteria reduction," Journal of the Operational Research Society, 64(8), pp. 1205-1220.
- [22] Khan, S., S. H. K. Banuri, M. Nauman, S. Khan, and M. Alam (2010) Analysis report on Android Application Framework and existing Security Architecture, Tech. rep., Security Engineering Research Group, Institute of Management Sciences Peshawar, Pakistan.
- [23] Al-Rayes, H. T. (2012) "Studying Main Differences between Android & Linux Operating Systems," International Journal of Electrical & Computer Sciences, 12(5).
- [24] Shi, Y., K. Casey, M. A. Ertl, and D. Gregg (2008) "Virtual Machine Showdown: Stack Versus Registers," ACM Trans. Archit. Code Optim., 4(4), pp. 2:1-2:36.
- [25] Google Android Developers, "Application Fundamentals,"<http://developer.android.com/guide/components/fundamentals.html>.

- [26] Google Android Developers, “App Manifest,” <http://developer.android.com/guide/topics/manifest/manifest-intro.html>.
- [27] Android Developers, “What is API Level?” <http://developer.android.com/guide/topics/manifest/uses-sdkelement.html#ApiLevels>.
- [28] Enck, W., P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth (2010) “TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones,” in Proceedings of the 9th USENIX Conference on OSDI’10, pp. 1–6.
- [29] Zhou, Y., Z. Wang, W. Zhou, and X. Jiang (2012) “Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets,” In Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS).
- [30] Grace, M. C., Y. Zhou, Q. Zhang, S. Zou, and X. Jiang (2012) “RiskRanker: scalable and accurate zero-day android malware detection,” in MobiSys, pp. 281–294.
- [31] Yan, L. K. and H. Yin (2012) “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis,” in Proceedings of the 21st USENIX Conference on Security Symposium, Security’12, pp. 29–29.
- [32] Wang, X., Y.-C. Jhi, S. Zhu, and P. Liu (2009) “Behavior based software theft detection,” Proceedings of the 16th ACM conference on Computer and communications security (CCS ’09). ACM, New York, NY, USA, 280-290. DOI: <https://doi.org/10.1145/1653662.1653696>
- [33] Wang, X., Y.-C. Jhi, S. Zhu, and P. Liu (2009) “Detecting Software Theft via System Call Based Birthmarks,” in Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC), IEEE Computer Society, pp. 149–158.
- [34] Kolbitsch, C., P. M. Comparetti, C. Kruegel, E. Kirda, X. yongZhou, and X. Wang (2009) “Effective and Efficient Malware Detection at the End Host,” in USENIX Security Symposium, pp. 351–366.
- [35] Zhao B., Liu P. (2013) Behavior Decomposition: Aspect-Level Browser Extension Clustering and Its Security Implications. In: Stolfo S.J., Stavrou A., Wright C.V. (eds) Research in Attacks, Intrusions, and Defenses. RAID 2013. Lecture Notes in Computer Science, vol 8145. Springer, Berlin, Heidelberg
- [36] Linux Man Page, “strace,” <http://linux.die.net/man/1/strace>.
- [37] AOSP, “Camera,” <https://source.android.com/devices/camera/index.html>.
- [38] Chen, K. Z., N. M. Johnson, V. D’Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. X. Song (2013) “Contextual Policy Enforcement in Android Applications with Permission Event Graphs,” in Proceedings of the Annual Network and Distributed System Security Symposium (NDSS).
- [39] Singh, K. (2011) “Practical Context-Aware Permission Control for Hybrid Mobile Applications,” In: Stolfo S.J., Stavrou A., Wright C.V. (eds) Research in Attacks, Intrusions, and Defenses RAID, pp. 307–327.
- [40] Zhao, B. and P. Liu (2015) “Private browsing mode not really that private: Dealing with privacy breach caused by browser extensions,” in 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2015, Rio de Janeiro, pp. 184–195.
- [41] Cai, H., Meng, N., Ryder, B., and D. Yao. (2018). DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling. IEEE Transactions on Information Forensics and Security. PP. 14. 10.1109/TIFS.2018.2879302.
- [42] Aafer, Y., Tao, G., Huang, J., Zhang, X., and N. Li. (2018). Precise Android API Protection Mapping Derivation and Reasoning. In Proceedings of the ACM Conference on Computer and Communications Security (CCS).
- [43] Dimjašević, M., Atzeni, S., Ugrina, I., and Z. Rakamaric. (2016). Evaluation of Android Malware Detection Based on System Calls. In Proceedings of the 2016 ACM on International Workshop on

- [44] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. (2008). Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08). ACM, New York, NY, USA, 15-26. DOI: <https://doi.org/10.1145/1390630.1390635>
- [45] Symbolic Java PathFinder. <http://javapathfinder.sourceforge.net/>

AUTHOR

Dr. Bin Zhao graduated from the Cyber Security Lab at the Pennsylvania State University in 2015. He worked for Palo Alto Networks as a Senior Staff Engineer. Currently he is working in JD.com Silicon Valley R&D Center as a Security Architect. His primary interests are network security, software security, black market, mobile data leakage, and network protocol analysis.

