# Tainting-Assisted and Context-Migrated Symbolic Execution of Android Framework for Vulnerability Discovery and Exploit Generation

Lannan Luo, *Member, IEEE*, Qiang Zeng*, *Member, IEEE*, Chen Cao, Kai Chen, *Member, IEEE*, Jian Liu, *Member, IEEE*, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu, *Member, IEEE*

**Abstract**—Android Application Framework is an integral and foundational part of the Android system. Each of the two billion (as of 2017) Android devices relies on the system services of Android Framework to manage applications and system resources. Given its critical role, a vulnerability in the framework can be exploited to launch large-scale cyber attacks and cause severe harms to user security and privacy. Recently, many vulnerabilities in Android Framework were exposed, showing that it is indeed vulnerable and exploitable. While there is a large body of studies on *Android application* analysis, research on *Android Framework* analysis is very limited. In particular, to our knowledge, there is no prior work that investigates how to enable symbolic execution of the framework, an approach that has proven to be very powerful for vulnerability discovery and exploit generation. We design and build the first system, CENTAUR, that enables symbolic execution of Android Framework. Due to the middleware nature and technical peculiarities of the framework that impinge on the analysis, many unique challenges arise and are addressed in CENTAUR. The system has been applied to discovering new vulnerability instances, which can be exploited by recently uncovered attacks against the framework, and to generating PoC exploits.

**Index Terms**—Symbolic execution, concolic execution, vulnerability discovery, exploit generation, Android Framework.

## 1 INTRODUCTION

THE global smartphone market is booming and Android dominates the market with a share of 87.6% [36]. As of 2017, there were two billion active Android devices [45]. Each device relies on the Android Application Framework (*Android Framework*, for short) to make it useful. E.g., all the user interface designs and multi-tasking features would not work without WMS (Window Manager Service) and AMS (Activity Manager Service) of Android Framework; as another example, apps cannot obtain the GPS location without LMS (Location Manager Service) of the framework. Thus, Android Framework is an integral and foundational part of the Android system; it runs on each Android device for managing all applications and providing a generic abstraction for hardware access [29]. Recently, many vulnerabilities in Android Framework were identified [17], [18], [19], [20]. A vulnerability in the framework can lead to large-scale cyber attacks and cause serious harms to user security and privacy; e.g., malicious apps can exploit them to steal user passwords, take pictures in the background, launch UI spoofing attacks, and tamper with user data [54], [56], [57].

Despite the critical role of Android Framework and the concern of vulnerabilities hidden in its multi-million lines of code, most of the existing work has been focused on analyzing *Android applications* [2], [11], [12], [13], [14], [24], [25], [41], [42], [43], [48], [53], [62], [66], [67]. Very few systems are available for analyzing Android Framework; they either perform fuzzing [26] or simple static analysis, such as call graph generation and its reachability analysis [3], [5], [6].

As a result, the insecurity analysis of the framework has been largely imprecise and requires significant manual effort [54], [56]. For example, Shao et al. [56] uncovered a very interesting type of Android Framework vulnerabilities that are due to inconsistent permission checking (detailed in Section 8.3); however, due to the overwhelming amount of manual effort needed to validate their findings, the process of vulnerability discovery was tedious and error-prone; moreover, the reported vulnerabilities were hard to verify since no PoC exploits are generated. Plus, an increasing number of vulnerabilities have been revealed [10], [17], [18], [19], [20], [34], [57], [61], and many more yet discovered

* *Correponding author.*

- *L. Luo is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208.*
- *Q. Zeng is with the Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208.*
- *C. Chao is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *K. Chen is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China 100012, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China 100049.*
- *J. Liu is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China 100012, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China 100049.*
- *L. Liu is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China 100012, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China 100049.*
- *N. Gao is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China 100012, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China 100049.*
- *M. Yang is with the School of Computer science, Fudan University, Shanghai, China 200433.*
- *X. Xing is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *P. Liu is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*

This manuscript is an extension of the conference version published in the *Proceedings of the 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'17)* [44]. This manuscript presents a more detailed description of the system design, implementation, and evaluation. We extended CENTAUR to support symbolic execution of multiple processes of Android Framework and evaluated the system.

Fig. 1. The `getProviders` service interface method.

```
1  // Defined in the LocationManagerService class
2  List<String> getProviders(Criteria criteria,
       boolean enabledOnly) {
3    int uid = Binder.getCallingUid();
4    int level = getAllowedResolutionLevel(uid);
5    ArrayList<String> out = new ArrayList<String>();
6    for (LocationProviderInterface p : mProviders) {
7      if(level >= p.requriedLevel) {
8        ...
9        out.add(p.name);
10     }
11   }
12   return out;
13 }
14 int getAllowedResolutionLevel(int uid) {
15   // Inside checkPermission(), the function
       getUserIdLPr() is invoked
16   HashSet ps = mContext.checkPermission(uid);
17   if(ps.contains(ACCESS_FINE_LOCATION))
18     return 2;
19   else if (ps.contains(ACCESS_COARSE_LOCATION))
20     return 1;
21   else
22     return 0;
23 }
24
25 // Defined in the Settings class
26 static final int FIRST_APPLICATION_UID = 10000;
27 static final int PER_USER_RANGE = 100000;
28 ArrayList<Object> mUserIds;
29 Object getUserIdLPr(int uid) {
30   if (uid >= FIRST_APPLICATION_UID) {
31     uid %= PER_USER_RANGE;
32     int index = uid - FIRST_APPLICATION_UID;
33     return mUserIds.get(index);
34   }
35 }
```

from many *custom* system services and Google's standard ones. Thus, there is an urgent need for techniques and tools for *precise* and *automated* insecurity analysis of Android Framework. In particular, to our knowledge, there is no tool that is able to analyze the framework through *symbolic execution*, a precise and automated analysis approach that has proven to be very powerful for vulnerability discovery and exploit generation [4], [9], [28].

This work is to fill the gap, aiming to (1) study the unique design features and technical peculiarities of the framework that may impede symbolic execution analysis, (2) design and build a system that enables symbolic execution of the Android Framework, and (3) apply the system to precisely and automatically finding zero-day vulnerability instances and generating PoC exploits to validate the findings; such exploits can also be fed into defense systems for automated malware signature generation [4], [16].

While many symbolic execution systems have been proposed for analyzing *Windows programs* [7], [27], [28], *Unix programs* [4], [8], [9], and *Android apps* [1], [37], [46], [47], [64], none has explored how to symbolically analyze such complex middleware as Android Framework. Due to unique characteristics of the framework, many new challenges arise.

**Challenge 1.** Unlike an independent program, Android Framework is a large piece of middleware consisting of many system services, atop which Android applications are started and run. It is not surprising that Android Framework has a complex initialization phase, which parses system and application settings and then prepares all the system services. Symbolic execution that starts from the main entry of Android Framework, `SystemServer.main`, would quickly cause state explosion and hence cannot reach deep code paths. Meanwhile, Android Framework exports system services to apps in the form of *service interface methods* (also called *entrypoint methods*); e.g., in Android Framework 5.0, there are 3,079 entrypoint methods exported. Our insight is that, instead of analyzing Android Framework as a whole, *the capability of analyzing each entrypoint method separately is the key to the* scalability *of the symbolic execution analysis.*

Under-constrained symbolic execution can directly start from an arbitrary function within a program [51]. However, the context information, such as the *type* and *value* of variables, for executing the target function is missing and thus many problems may be caused. First, without the *type* information, it is hard to precisely determine the dispatch target of a virtual function call. For example, Figure 1 shows the code for the service interface method `getProviders`,[1] which returns the names of the GPS providers that the calling app is allowed to access. Line 16 contains a virtual function call to `checkPermission` through `mContext`, a reference variable of the `Context` type; `Context` is an abstract class extended by four classes, including `ContextWrapper`, `ContextImpl`, `BridgeContext`, and `MockContext`, each of which implements the function `checkPermission` and is further inherited by other classes. Without the concrete *type* information of the object pointed to by `mContext`, it is hard to precisely determine the dispatch target of the call. Such virtual function calls prevail in the framework code. Consider a call `s.iterator()` as another example, where `s` is a reference of the `Set` interface type; `Set` is implemented by over 40 subclasses in Android Framework code, which means that symbolic execution needs to try each possibility if the type information of the object pointed to by `s` is missing, causing many spurious paths to be explored.

Second, without the *value* information of variables, the state explosion problem can be exacerbated. For example, consider `mProviders` in Line 6 as an example, which is an ArrayList that stores the currently installed GPS providers; if the elements in the list are unknown, it is difficult to carry out a loop that iterates through the list. One workaround is to regard the list as a symbolic input and then handle it using lazy initialization [39]; this way, however, the loop becomes unbounded and elements of the list become symbolic, which exacerbates the state explosion problem.

Therefore, while it increases the scalability of analysis by directly starting symbolic execution from a system service entrypoint method (i.e., API), *how to deal with the situation of the missing context information is a challenging problem (**C1**)*. To resolve it, we employ an analysis scheme that *bridges* concrete execution and symbolic execution, allowing analysis to start from an arbitrary middleware API method (Section 3).

**Challenge 2.** *Another uniqueness is the form of exploits*. Existing techniques typically generate PoC exploits as some simple form of inputs of stand-alone executables, such as a command line argument, a format string, a network packet, etc. In contrast, the exploit we consider here is a malicious app,

1. The code snippet has been modified slightly from the original to ease the understanding.

which comprises the app's configuration (i.e., the manifest file) and code that issues system service calls.

From the perspective of finding vulnerabilities exploitable by a malicious app, *any* framework variables derived from the malicious app are under the control of attackers, and thus should be specified as symbolic inputs. However, Android Framework contains a large number of data structures. Given an app, the variables derived from it *scatter* in the form of object fields and array elements in the many data structures.

In Figure 1, for example, `mUserIds` at Line 28 is an ArrayList that stores the information of all the installed apps with one element for each app. The object pointed to by `ps` at Line 16 is an element of `mUserIds` (Line 33) derived from the calling app and thus should be handled as a symbolic input (as it is under the control of attackers), such that all the branches in the function `getAllowedResolutionLevel` can be explored by considering different values of the object. *How to automatically identify variables derived from the malicious app, among the large number of data structures in the framework, is an intriguing new problem as well as a challenge (**C2**).*

Our intuition is that, as the framework stores information for multiple clients (i.e., apps), there should exist certain *patterns* used to access the client-specific information when the framework services a client's call. This intuition is verified via our manual investigation. Based on the patterns how client-specific variables are accessed upon a system service call, a customized taint analysis approach, called *slim tainting*, is designed to precisely and automatically pinpoint client-specific variables (Section 4).

**Challenge 3.** A straightforward design is to place the symbolic execution engine inside the Android system, such that the analyzer can make use of the underlying system conveniently (e.g., to invoke native libraries). However, the Android system is designed for concrete execution, which is very different from symbolic execution in terms of thread management, garbage collection, object representation, instruction execution, etc. Thus, with the "inside-the-box" design, the symbolic executor implementation has to take care of the compatibility issues. This significantly complicates the symbolic executor implementation. Moreover, whenever the related Android components (e.g., the Android Runtime) are changed, the implementation of symbolic executor has to be updated and verified. To avoid the complicated implementation and endless maintenance, a decoupled architecture is desired. However, *how to design an architecture that can make use of the Android execution environment without leading to a complex coupled implementation is a challenge (**C3**).*

To solve the challenge, an innovative component of the system is designed to migrate information generated in Android, such as the classes and objects, to the external symbolic execution environment (Section 5).

We have overcome the challenges above and implemented a system named CENTAUR for symbolic execution of the *bytecode* of Android Framework (i.e., CENTAUR does *not* need the source code of the framework to perform analysis), and it is publicly available.[2] We concretely demonstrate how CENTAUR can be applied to vulnerability discovery. In contrast to recent research that relies on laborious and error-prone manual work for finding framework vulnerabilities [54], [56], we show that how it is automated and guarantees *zero-false positives*. Finally, we make use of CENTAUR to generate PoC exploits to validate the findings. We make the following contributions.

- Many unique design features and technical internals of the framework are studied and documented. We have revealed how application-specific information is retrieved (Section 4.3), categorized how calls among system services are handled (Section 6), and explored messaging and JNI calls (Section 7). As a result of the efforts, we have established an in-depth knowledge base that future analyses of the framework can leverage.
- Unlike previous symbolic execution techniques that either start analysis from the main function or analyze a non-main function without the context information, we propose techniques that allow service interface methods of middleware to be analyzed separately, for much improved scalability, without harming the soundness of the analysis results.
- A novel tainting analysis technique is proposed to precisely identify the framework variables derived from a given app. It is particularly suitable for vulnerability discovery as it considers all possible values under the control of a malicious app.
- An innovative architecture that builds the symbolic executor out of the Android system is proposed. An enabling algorithm that migrates the execution context information from the Android system to the symbolic executor is designed.
- We have implemented CENTAUR and demonstrated its effectiveness and precision through applications of vulnerability discovery and PoC exploit generation. To our best knowledge, CENTAUR is the first system that supports symbolic execution of Android Framework. It exemplifies symbolic analysis of complex middleware, a largely omitted but important research problem.

## 2 BACKGROUND

**Android Framework.** Android Framework provides a collection of system services, which implements many fundamental functionalities, such as managing the life cycle of all apps, organizing activities into tasks, and managing app packages. Most of the system services run as threads in the `system_server` process [29], while some others run as threads in other processes, e.g., `com.android.phone`, `com.android.keychain`, and `com.android.inputmethod.latin`, etc.

A system service exposes its service interface methods invokable by apps, and a *system service call* is handled in the form of a remote procedural call through the `Binder` IPC mechanism, which dispatches the call to one of the threads of the target system service. Android Framework is mainly implemented in Java. For example, in Android Framework 5.0, there are 2.4 million lines of Java code and 880 thousand lines of C/C++ code. Currently, CENTAUR can only perform symbolic execution of the Java code.

**Symbolic Execution.** Symbolic execution provides a means of efficiently exploring execution paths [40]. For example, consider the function `getAllowedResolutionLevel` in

2. https://github.com/Android-Framework-Symbolic-Executor/Centaur

Figure 1, by assigning a symbolic value to `ps`, symbolic execution analysis can iterate every of the three paths and precisely provide the condition that the symbolic value should satisfy for executing a given path; e.g., the symbolic execution analysis can produce the path condition for reaching Line 18: `ps.contains(ACCESS_FINE_LOCATION)`.

Symbolic execution is particularly suitable for vulnerability discovery. First, it performs an efficient and automatic path exploration and ideally explores all possible paths, so that it is able to discover as many vulnerability instances as possible. Second, for each path explored, it records a path condition, which is a symbolic expression describing the condition that should be satisfied by the input values in order that the path is taken. Consequently, by resolving the path condition, one can obtain the concrete input values that force the execution to follow the corresponding path; the concrete input values can be used to construct exploits, and can be fed into real program execution for verifying the suspected vulnerability. This way, it guarantees zero false-positives in vulnerability discovery.

One of the main challenges in applying symbolic execution to large-sized programs is to cope with the path explosion problem, as the number of distinct execution paths is exponential in the number of branches that depend on symbolic values. We mitigate the problem using multiple ways, such as analyzing service interface methods separately and precisely identifying variables as symbolic inputs.

## 3 Overview

### 3.1 Approach Overview

When a vulnerability is exploited, some security property is violated. For example, a vulnerability is exploited if the property that "*some resource/service can only be accessed by an app with proper permissions*" is violated; a task-hijacking attack succeeds when the security property that "*the malicious activity should not be placed onto the back stack hosting the victim activity*" is broken. In order to discover vulnerabilities that can be exploited by a given type of attacks $A$ that breaks some security property $P$, we employ *symbolic execution* to explore program paths. Specifically, first, the violation of $P$ is represented as a set of constraints $\mathcal{C}$, called *security-property violation constraints*, which is a quantifier-free first-order logic formula in terms of Android Framework variables. Then, during path exploration, $\mathcal{C}$ is added to each path condition, wherever the variable scopes allow, before the path condition is resolved. If the augmented path condition is resolvable, a path that can be exploited by $A$ is found. The resolved variable values are used to construct PoC exploits.

### 3.2 System Overview

Our observation of Android Framework is that its execution consists of the *relatively stable* initialization phase and the ready-for-use phase; the initialization phase is fairly stable when the system restarts, since the system boots mainly according to the system configuration, which itself is stable. Thus, to resolve the problem of the missing context information (**C1**), we propose a *phased concrete-to-symbolic execution* (PC2SE) for analyzing middleware software like Android Framework; it runs the initialization phase as whole-system concrete execution and then performs symbolic execution starting at one of the entrypoint methods under the execution context provided by the concrete execution. It avoids the state space explosion due to the complex initialization phase and meanwhile provides the context for symbolic execution, such that the type and value information of the *input variables* (i.e., non-locally defined variables read during symbolic execution) is available. Plus, the proper combination of concrete and symbolic executions delivers analysis results that are sensible and easy to interpret.

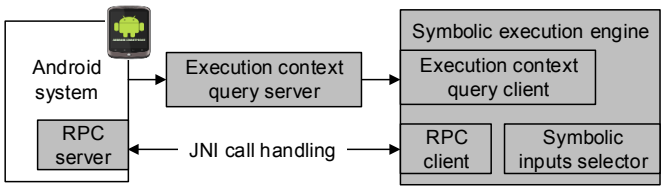


Fig. 2. Architecture of CENTAUR.

When starting the symbolic execution from an entrypoint method, if only the parameters of the entrypoint method are set as symbolic inputs [49], the path exploration will be severely limited, leading to *over-constrained* symbolic execution. In the framework, any variables derived from the malicious app (such as its manifest file) are under the control of attackers and can affect the execution of system service calls; hence, these variables should be set as symbolic inputs. To resolve **C2** (i.e., identifying variables derived from the malicious app as symbolic inputs), instead of tracking how information is flowed from an app to the framework, we investigate how the app-specific variables in the framework are accessed, and propose *slim tainting* that recognizes such accesses to identify those variables as symbolic inputs on the fly during path exploration (Section 4). This way, the path exploration considers all possible values of these variables.

To address **C3** (i.e., to avoid complicated implementation and endless maintenance due to the coupled design), we propose a novel architecture that fits PC2SE, as shown in Figure 2, where the symbolic execution engine is built outside Android and makes use of the execution context migrated from an Android system. As the symbolic execution engine does not need to take care of the comparability issues but is specialized for path exploration, its design and implementation are largely simplified. Plus, since the engine is decoupled from Android, it does not need to be maintained when the Android system code is updated.

The whole-system concrete execution is performed in the Android system. Between the Android system and the symbolic execution engine is the execution context query server, which migrates the context information from Android to the symbolic execution engine. How to correctly interpret the *semantics* of the bytes and bits in the heap captured at Android and how to mitigate the information seamlessly will be discussed in Section 5. Finally, an RPC server is installed on the Android system, such that JNI calls during path exploration are delegated to the RPC server, which will be discussed in Section 7.

## 4 Identifying Symbolic Inputs

While phased concrete-to-symbolic execution (PC2SE) allows analyzing each system service API separately, which signifi-

cantly improves the analysis scalability, this strategy alone is inadequate. Compared to standalone programs, a *uniqueness of Android Framework* is that it contains not only variables representing the statuses of the system services and the underlying system, but also variables derived from the apps. From the perspective of finding vulnerabilities exploitable by a malicious app, (1) variables that are derived from the malicious app are under control of attackers, and thus should be specified as symbolic inputs, such that *branches depending on them are all explored* during the analysis; and (2) on the other hand, if variables that are not derived from the malicious app are set as symbolic inputs, then *spurious paths will be explored*, which will harm the scalability of symbolic execution and cause the results to be difficult to interpret.

### 4.1 App-specific Variables

Our investigation on Android Framework reveals that there are two distinct types of variables. The first type, called *non-app-specific* variables, are allocated regardless of apps in the system. Take the code in Listing 1 as an example. The ArrayList `LocationManagerService.mProviders` (Line 6) exists no matter what apps are running or installed. It should *not* be regarded as a symbolic input, since it is not under the control of the malicious app; otherwise, symbolic execution of the loop at Line 6 will hurt the analysis scalability, and also generate results difficult to interpret.

The second type, called *app-specific* variables, stores app-specific information. For instance, `mUserIds` at Line 28 is an ArrayList that stores the information of the installed apps. with one element for each app. The object pointed to by `ps` at Line 16 is an element of `mUserIds` (Line 33); it is derived from the calling app and thus should be handled as a symbolic input, such that all the branches in the function `getAllowedResolutionLevel` will be explored by considering different values of the object.

Unlike the Linux kernel, which stores most information of a process in a centralized structure `task_struct`, the app-specific information in Android is stored in many different data structures organized by the system services. Given an app, the framework variables derived from it scatter and exist as objects fields and array elements among the many data structures.

Therefore, the task of selecting variables as symbolic inputs is not only to find data structures for storing app-specific information but also to locate fields or elements within the data structures that are derived from a given malicious app. For instance, in addition to determining `mUserIds` is an app-specific variable, we need to locate *which element in the array* is derived from the malicious app. Hence, the task is like looking for a needle in a large pile of hay considering the many complex data structures.

### 4.2 Why not Use Traditional Tainting?

To determine which variables are derived from a given app, a natural method is to use *tainting* to track how the information flows from the app to the processes of Android Framework. However, such information flow is very complex involving multiple intricate steps, including app installation, system boot, and starting the app. Given the complexity of these steps and the huge amount of code involved, it is very difficult, if not impossible, to precisely track the information flow. Note that existing Android taint analysis systems, such as TaintDroid [24], FlowDroid [2], and TaintART [58], target Android *applications*; e.g., they are able to track whether the return value (e.g., GPS locations) of a system service call flows, according to the app code, to specific sinks (e.g., Internet), but none is able to track how the whole app-level information propagates within *Android Framework*.

Fig. 3. Example of retrieving information from a hash-table-based variable `mPackages`.

```
// Defined in the PackageManagerService class
HashMap<String, PackageParser.Package> mPackages;
int checkPermission(String perm, String pkgNm){
  PackageParser.Package p = mPackages.get(pkgNm);
   ...
}
```

### 4.3 Access Patterns

Instead of proposing a even more complex taint analysis technique to track the information flow, we resolve the challenge from a novel angle by looking at how the app-specific variables are accessed. As the framework stores information for multiple apps, when a system service call from an app is handled, we suspected there should exist specific ways to retrieve the information for the calling app. We thus first manually analyzed part of the framework code (related to vulnerabilities in Table 3). During the manual investigation, for the code used to retrieve the calling app's information, we summarized the access patterns. We then verified whether the summarized access patterns kept by checking the code for over twenty other system calls. Moreover, we further verified the access patterns backward starting from the locations when elements of various data structure instances were retrieved. Our investigation shows that app-specific variables are stored in two categories of data structures, array-based ones (built-in arrays, ArrayList, SparseArray, etc.) and hash-table-based ones (HashMap, HashSet, etc.), and the two categories are accessed in two characteristic ways, respectively.

First, given an array-based variable, the framework retrieves an app's information in the array using an index that is a function of the app's unique UID (an app's UID is assigned upon installation and not changed). Our investigation shows that two formulas are used to calculate the index. One is $(uid\%100,000 - 10,000)$, converting the user app's UID into an index to retrieve the element for the app from a built-in array or ArrayList; the other one is $(uid\%100,000)$, which is used to calculate the index into a SparseArray.[3] For example, as shown in Figure 1, the function `getUserIdLpr` (Line 29) utilizes the first formula to calculate the index into the ArrayList `Settings.mUserIds` (Lines 31 and 1).

Second, for hash-table-based variables, no matter it is hash table or a set, the package name (or the package name concatenated with a component name) is used as the key to access elements. Figure 3 shows an example of retrieving information from a hash-table-based variable.

3. Two magic numbers appear in the formulas and are worth interpretation. $10,000$ means `FIRST_APPLICATION_UID` (Line 26 in Figure 1), indicating the smallest UID a user app can have, while $100,000$ means `PER_USER_RANGE` (Line 27), indicating the largest UID plus one.

TABLE 1
Taint Propagation Logic. Register variables are referenced by $v_X$. $\tau(y) \leftarrow \tau(x)$ means setting the taint tag of $y$ to the taint tag of $x$.

| Inst. / Function | Operation Semantics | Taint Propagation |
|---|---|---|
| isub | $v_B \leftarrow v_A - C$ | if C == 10,000, $\tau(v_B) \leftarrow \tau(v_A)$ |
| irem | $v_B \leftarrow v_A \,\%\, C$ | if C == 100, 000, $\tau(v_B) \leftarrow \tau(v_A)$ |
| concat | $v_C \leftarrow v_A$.concat$(v_B)$ | $\tau(v_C) \leftarrow \tau(v_A)$ |

TABLE 2
Bytecode instructions (and function) used for migrating heap information.

| Instruction | Stack [before]→[after] | Description |
|---|---|---|
| getfield | objRef → value | get a field value of an object |
| getstatic | →value | get a static field value of a class |
| aaload | arrayRef, index → value | load onto the stack a reference from an array |
| initClass | N/A | invoked for class initialization |

In short, while there are a large variety of data structures in the framework, our manual investigation on the framework code shows that they commonly follow the two fixed access patterns to retrieve the app-specific information when servicing a system service call. This is verified on Android Framework versions from 4.0 to the latest version 9.0 that we have investigated.

### 4.4 Slim Tainting

Based on the insight, we propose *slim taint analysis* that tracks and recognizes the characteristic access patterns on the fly during path exploration and, when app-specific information is accessed, sets those variables as symbolic inputs.

Slim taint analysis consists of the following three parts. **(1) Taint sources:** the return values of `getCallingUID()` and `getPackageName()` are set as taint sources; they are unique identifications of an app involved in the characteristic access patterns. **(2) Taint propagation:** as shown in Table 1, the taint propagation logic is very concise, involving only two instructions and one string concatenation function, which are used in the access patterns aforementioned. **(3) Taint sinks:** the taint sinks include the `get` functions of the collection data structures as well as bytecode instructions for loading elements from built-in arrays, such as `iaload` (loads from an array of integers) and `aaload` (loads from an array of references); they check whether the index or key is *tainted*, and if so, the target element is flagged as a symbolic input.

**Example:** Let us take the code in Listing 1 as an example to illustrate how slim tainting works. First, the return value of `getCallingUID()` (Line 3) is the taint source. Second, the taint propagates along Lines 31 and 1 according to the taint propagation logic in Table 1. Finally, at Line 33, the `get` function works as a sink to set the element (and *only* this element) accessed via the tainted index as a symbolic input.

Slim tainting comprises very specific taint sources and a simple but precise taint propagation logic; it thus avoids the overtainting and undertainting issues. Section 7 includes its implementation details. It works through interception of the function calls and bytecode instructions aforementioned, so it does not need to change any code of Android Framework and does not need code annotation.

## 5 Execution Context Migration

The PC2SE involves both concrete execution and symbolic execution. As described in Section 3, it would be very difficult to modify the Android system (or its emulator) to support both concrete execution and symbolic execution. To enable the decoupled architectural design, however, two challenging tasks should be resolved: (a) how to mitigate the execution context from the host system to the symbolic executor, and (b) how to handle function calls (e.g., JNI calls) that are not interpreted by symbolic execution. This section discusses the solution to the first task, and the solution to the second one will be discussed in Section 7.

### 5.1 Execution Context

In the execution context, the program counter, the register file, and the stack all obtain proper fresh content when symbolic execution starts at the analyzer; only the heap in the execution context, which is a collection of classes and objects, needs to be migrated. The heap memory image in the execution context is called a *snapshot* for short. Three problems have to be resolved for migrating the heap information captured in a snapshot: (1) how to obtain the semantics of the bits and bytes in a snapshot? (Section 5.2) (2) how to conduct the migration during symbolic execution? (Section 5.3) and (3) how to bootstrap the migration? (Section 5.4)

### 5.2 Snapshot Parsing and Context Query Server

A snapshot is nothing but an array of bits. However, it would not work if we simply copy the array of bits to the symbolic executor, because the ART process in Android and the JVM instance for symbolic execution differ significantly in terms of the low-level representation of classes and objects. E.g., in our symbolic executor implementation, each object needs extra space for recording the taint and the symbolic expression; plus, its heap memory management is different from the one used in Android. Our insight is that, given an object, both ART and the JVM instance should agree on the number of the contained fields, according to the class definition file, and their values. Therefore, given an object, the migration is not to copy its bits but to copy the values of all its fields.

Thus, the parser analyzes the snapshot to obtain all the active objects (and classes) and, for each object (and class), records the values of its fields. The information is organized in a two-tier data structure: the first tier maps an object (or class) address to a second-tier data structure instance, which maps field names of an object (or class) or element indexes of an array to their values.

After the snapshot is parsed and its information is stored, the *execution context query server* (in Figure 2) is used to service requests from the symbolic executor by returning the information about objects, classes and arrays. Multiple query interfaces are provided: given a reference value, the type of the corresponding object can be queried; given a reference value of an object and the name of one of its fields, the field value can be queried. Snapshot parsing and information query provide the foundation for heap information migration.

### 5.3 On-the-fly Migration

Given an object in the concrete execution world, for fields of primitive types we can simply copy the field values after allocating the space from the symbolic executor for

**Algorithm 1** Migration of heap information.

```
 1: function GETFIELD(index)
 2:     objRef = peekStackTop()
 3:     fdInfo = getFdInfo(index)            ▷ Class-specific info.
 4:     fd = getFd(objRef, fdInfo)           ▷ objRef-specific info.
 5:     if !fd.getSnapshotRefAttribute() then
 6:         return super.getfield(index)
 7:     end if
 8:     concRef = fd.getValue()
 9:     symRef = conc2Sym.get(concRef)
10:     if symRef == NULL then
11:         fdType = fdInfo.getFdType()
12:         if fdType == strRef then
13:             str = snapshot.getStr(concRef)
14:             symRef = searchConstantPool(str);
15:             if symRef == NULL then
16:                 symRef = newString(str);
17:             end if
18:         else if fdType == arrayRef then
19:             entryType = fdType.getEntryType()
20:             len = snapshot.getArrayLen(concRef)
21:             symRef = newArray(entryType, len)
22:             snapshot.copyEntries(symRef, concRef)
23:         else                              ▷ Other reference types
24:             symRef = newObj(fdType)
25:             snapshot.copyFields(symRef, concRef)
26:         end if
27:         conc2Sym.addPair(concRef, symRef)
28:     end if
29:     fd.setValue(symRef)
30:     fd.setSnapshotRefAttribute(false)
31:     return super.getfield(index)
32: end function
33:
34: function INITCLASS(classInfo)
35:     if snapshot.isInitialized(classInfo) then
36:         snapshot.copyStaticFields(classInfo)
37:     else
38:         super.initClass(classInfo)
39:         handleBootstrapField(classInfo)
40:     end if
41: end function
```

the object. But the handling of reference-type fields needs more consideration. A deep copy is inefficient while a simple shallow copy of the reference value will not work as the reference value only indicates the object location in the concrete execution world. We choose *a variant of the simple shallow copy*: when an object is migrated, we still simply copy all the field values, but for each reference-typed field, we set a flag indicating it is a reference value in the concrete execution world (a boolean attribute **`snapshotRef`** is associated with each reference-typed field to indicate whether the filed value is a location in the concrete or symbolic execution world); later, when one of such reference-typed fields is used to access its target object, the target object is either migrated, or (if it has been migrated) the field value is updated with the reference value in the symbolic execution world.

Therefore, a hash table, `conc2Sym`, is maintained to map reference values in the concrete execution world to ones in the symbolic execution world. Every time an object $o$ is migrated, a new pair $\langle r_c, r_s \rangle$ is added to `conc2Sym`, where $r_c$ is the reference value of $o$ in the concrete execution world and $r_s$ symbolic. The hash table is maintained for two purposes. First, it prevents duplicate migration of an object; that is, an object pointed to by $r_c$ is migrated only if $r_c$ is not found in the hash table. Second, the hash table is used to translate reference values in the concrete execution world, if they exist in the hash table, to ones in the symbolic execution world.[4]

**Algorithm.** We have designed an algorithm that migrates classes and objects from the snapshot to the symbolic executor. It runs by overriding the interpretation of specific instructions and function in the JVM for supporting migration. Table 2 shows the list of bytecode instructions and function whose interpretation is overridden; for each instruction, the effect that the instruction has on the operand stack and the description are included. (1) `getfield` and `getstatic` are overridden in order that, whenever a reference-typed field is accessed, if the object referenced by the field is not migrated yet, the object gets migrated and the hash table `conc2Sym` updated. (2) If a class has been initialized in concrete execution and is used in the symbolic executor for the first time, which automatically triggers the invocation of `initClass`, that class is migrated. (3) `aaload` is overridden to migrate multi-dimensional arrays. Algorithm 1 shows the main migration procedures. It involves much complexity due to the *JVM specification*, which is not familiar to many readers, so it is interpreted in detail as follows.

### 5.3.1 Migrating Objects

A `getfield` instruction is used to access non-static fields of an object. Given a reference `objRef` to an object (this object must have been migrated; Section 5.4 explains how this is ensured) on the stack (Line 2 in Algorithm 1) and the field `index` (Line 1; the index value is part of the instruction), the semantics of `getfield` is to pop `objRef` and push the field value onto the stack. Assuming the field points to an object that has not been migrated, we regard the access to the field as a proper occasion to trigger the migration of the object.

If the field's `snapshotRef` attribute is `false` (Line 5), which means that either it is a primitive-typed field or it has a reference value in the symbolic execution world, the instruction's interpretation is not changed (Line 6); i.e., the filed value is simply pushed onto the stack. If `snapshotRef` is `true` and the field value `concRef` is not found in `conc2Sym` (Line 10), the object should be migrated (Lines 11–26); after migration, the pair $\langle concRef, symRef \rangle$ is added to `conc2Sym` (Line 27).

How to migrate an object is determined by its type (Line 11). (Recall that, given the reference value, which is the value of the field being accessed, the execution context query server can locate and return the target object information, i.e., its type and contained field values, from the concrete execution world.) (1) If the object is a string, the algorithm first searches for a string that has the same value within the runtime constant pool in the VM for symbolic execution. If not found, a new string with the same value is created in the symbolic world (Lines 12–17). (2) If the object is an array, an array is allocated and all the elements are copied to the new array (Lines 18–22). This algorithm performs a shallow copy. Thus, for a multi-dimensional array, e.g.,

4. The hash table `conc2Sym` is handled as part of the process state, and gets stored and restored as the path exploration advances and backtracks, respectively; this way, the migration status keeps consistent during path exploration.

Fig. 4. Example of a test driver.

```
1 public TestDriver() {
2   @fromSnapshot
3   private static com.android.server.
      LocationManagerService mService;
4   public static void main() {
5     // The parameters are configured as symbolic
      inputs, so their values do not matter
6     mService.getProviders(null, false);
7   }
8 }
```

$A[5][10]$, only the five elements in the top-level array are copied at this moment. Later, when any of the five elements is accessed, the instruction `aaload` has to be invoked, which is the reason the interpretation of `aaload` (not shown in Algorithm 1) is also overridden, i.e., to migrate second-level arrays. Due to the shallow copy, an array object is not copied until a reference to the object is accessed. (3) A reference to an ordinary object is handled by allocating a new object and copying all its fields (Lines 23–25).

While non-static fields are accessed through `getfield`, access to static fields is through `getstatic`. Thus, to migrate objects pointed to by static fields, the interpretation of `getstatic` has to be overridden, and the interpretation is similar to that of `getfield` and is thus omitted.

#### 5.3.2 Migrating Classes

When an operation (e.g., creating an object of a class, or accessing a class's static fields for the first time) triggers initialization of a class during symbolic execution, `initClass` is invoked by the underlying VM for symbolic execution automatically. For classes that have been initialized during concrete execution, the symbolic executor has to make sure that they are migrated instead of being initialized, considering that the static fields have obtained their values during concrete execution. Thus, when `initClass` is invoked, the symbolic executor first checks whether the class has been initialized in the concrete execution world; if so, the enclosed static fields in the class are copied from the snapshot to the symbolic execution world (Line 36). In particular, when an object of some class is created in the symbolic world for the first time due to migration (Line 24), it triggers the invocation of `initClass` first, which migrates the class.

### 5.4 Bootstrapping

An important invariant kept during migration is that, whenever a field of an object $o$ (resp. an element of an array $A$) is accessed, $o$ (resp. $A$) must have been migrated to the symbolic execution world. Assume $f$ is the field whose access triggers the migration of the first object; a natural question is *"where does $f$ resides?"*. We resolve this bootstrapping problem by regarding the reference to the system service class containing the entrypoint method under investigation as the *bootstrap* field, and put it in the test driver class. Figure 4 shows an example of a test driver. A custom annotation `fromSnapshot` is used to specify the bootstrap field, which is recognized and handled by the migration algorithm; specifically, when a class (`TestDriver` in this example) is initialized, it sets the bootstrap field value to the reference value of the system service object (`mService` at Line 3) in the concrete execution world (note that all the system service classes adopt the *singleton* design pattern, so there is no ambiguity when specifying the reference value).

#### 5.4.1 Migration Tree

The migration of classes and objects forms a *migration tree*, which grows as new classes and objects are migrated, rooted at the class and object corresponding to the bootstrap field type. We use the test driver in Figure 4 as an example to illustrate how the migration tree is built.

`getProviders` in the `LocationManagerService` class is the entrypoint method under investigation. When the `TestDriver` class is initialized, the migration algorithm sets the value of the bootstrap field to the reference to the Location Manager service object in the snapshot; as a result, when the bootstrap field is accessed, the service object is migrated correctly. Figure 5 shows how the migration tree grows, due to the execution of the code in Listing 4; here, the root node is the class and object for `LocationManagerService`.

Part of the resulted migration tree is showed in Figure 6. It also shows how the symbolic input is identified. Because of slim tainting, the index 54 (in this example, the malicious app's UID is 10,054; 54 is due to the formula $(uid\%100,000-10,000)$ presented in Section 4.3) is tainted; thus, the element in the array `mUserIds` accessed through the tainted index is identified as a symbolic input.

## 6 Handling Service Calls

Service calls are frequently used among system services. Most system services of Android Framework run as threads in the `system_server` process [29], but a few run in other processes (e.g., `com.android.keychain`, `com.android.inputmethod.latin`, etc.). Depending on whether the caller service and the callee service are in the same process, a service call is handled in two distinct ways.

### 6.1 Handling Intra-process Service Calls

When the caller service and the callee service are in the same process, the service call is handled as an ordinary function call. Fig. 7 shows an example, where the Location Manager Service invokes `getProfiles` exposed by the User Manager Service; *both services belong to the `system_server` process.* The call at Line 4 invokes the function at Line 10, which issues an intra-process service call at Line 11.

Note that `UserManager.mService` is a variable of the `IUserManager` reference, and `IUserManager` is extended by multiple classes (including the `Proxy/Stub` classes to be introduced in Section 6.2 and the `UserManagerService` class). Previous research relies on expert knowledge to manually specify the dispatch target of the call at Line 11 to facilitate their static analysis of the framework code [3], [11], [56] (none performs symbolic execution), while CENTAUR makes use of the runtime information provided by the execution context. Specifically, based on the heap snapshot of the execution context, we can find that the object pointed to by `UserManager.mService` is the `UserManagerService` type, and thus the call is handled as an ordinary method call. Thus, expert knowledge and manual effort are not needed.

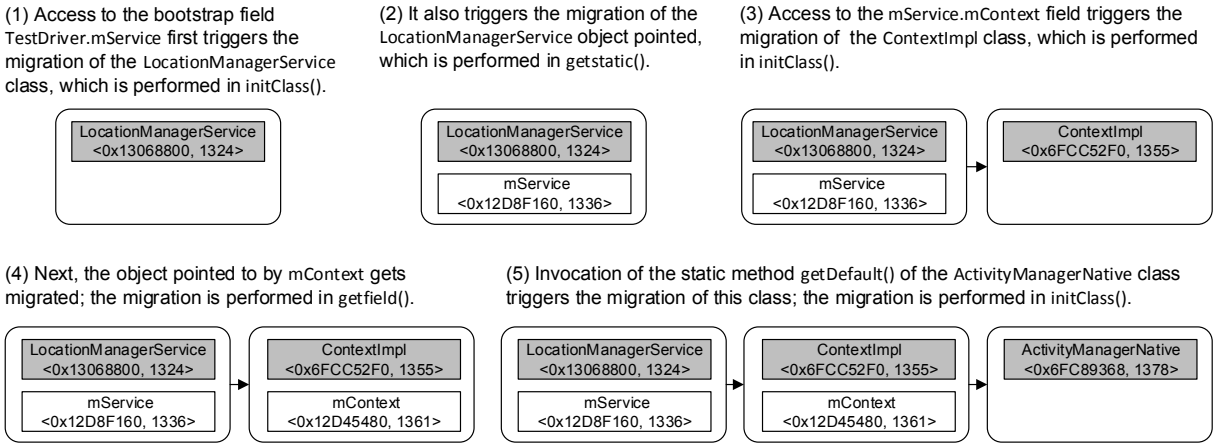


Fig. 5. An example of migrating the heap. Grey and white rectangles denote classes and objects, respectively. For each class and object, `<conRef, symRef>` denotes the mapping between the reference value in the concrete execution world and that in the symbolic world, which is added to the `conc2Sym` hash table. The migration of a class also triggers the migration of all its super classes, which are not shown for simplicity.

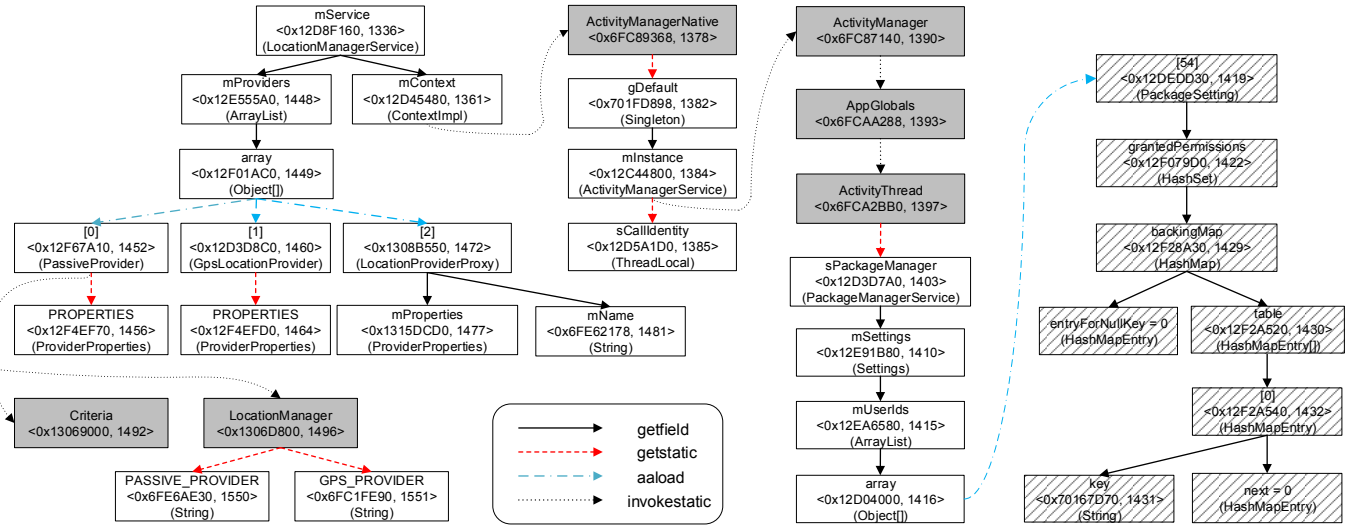


Fig. 6. Part of a migration tree (with some classes omitted). Grey and white rectangles denote classes and objects, respectively. Rectangles with diagonal stripes denote objects identified as symbolic inputs. Different arrows denote different instructions that have triggered the migration.

### 6.2 Handling Inter-process Service Calls

In the framework, when a system service invokes a method of another running in a different process, the call is handled via an inter-process communication mechanism called `Binder` [23]. The system service interface is defined using the Android Interface Definition Language (AIDL). From the defined interface, the AIDL compiler automatically generates `Stub` and `Proxy` classes that implement the interface-specific `Binder`-based IPC protocol. A `Stub` is an abstract class that implements the `Binder` interface and needs to be extended by the actual service implementation, while a `Proxy` is used by clients to invoke the service.

In order to perform an IPC service call, the client needs to first invoke `ServiceManager.getService(String)` using a unique string associated with the requested system service to obtain the `Proxy` of the service. For example, if a client wants to call the Telecom Service, it needs to first invoke `ServiceManager.getService(TELECOM_SERVICE)` to obtain the `Proxy` of the Telecom Service (Line 23 in Figure 8), and then perform the service call through this `Proxy` object (Line 24). After that, the `Binder` marshalls parameters of the service call, passes the data across process boundaries, and finally reassembles the objects in the remote process, one thread of which (the Telecom Service thread in this example) then executes the corresponding service method.

The functionality of `Binder` is mainly implemented in native libraries and the kernel, which cannot be interpreted by CENTAUR. To enable symbolic execution of such inter-process service calls, we have designed an automated solution that fits our decoupled design and migration algorithm. First, from the heap snapshots of all the processes of the framework, we build a database containing the type and value information of all active objects and classes (Section 5.2).

Second, we take advantage of the fact that Android uses AIDL to generate `Stub/Proxy` for all IPCs. By parsing the AIDL files, we can get the list of `Stub/Proxy` class pairs. Next, through a simple class hierarchy analysis, we can determine the mapping between each `Stub` type and the system service class that has extended the `Stub`. Based on the list and the mappings, we automatically build a hash table containing the mapping between each `Proxy` type and the name of the corresponding service class (which extends

Fig. 7. An *intra*-process service call example.

```
1 public class LocationManagerService {
2   private UserManager mUserManager;
3   void updateUserProfiles(int id) {
4     List<UserInfo> p = mUserManager.getProfiles(id);
5     ...
6   }
7 }
8 public class UserManager {
9   private final IUserManager mService;
10  public List<UserInfo> getProfiles(int uHandle) {
11    return mService.getProfiles(uHandle, false);
12  }
13 }
```

Fig. 8. An *inter*-process service call example.

```
14 class TelecomAccountRegistry {
15   private TelecomManager mTelecomManager;
16   void cleanupPhoneAccounts() {
17     hd = mTelecomManager.getAllPhoneAccountHandles();
18     ...
19   }
20 }
21 public class TelecomManager {
22   List<PhoneAccountHandle> getAllPhoneAccountHandles
       () {
23     ITelecomService mService = ITelecomService.Stub.
       asInterface(ServiceManager.getService(Context.
       TELECOM_SERVICE));
24     return mService.getAllPhoneAccountHandles();
25   }
26 }
```

the `Stub` generated together with the `Proxy` type).

Third, during the symbolic execution analysis, CENTAUR intercepts all the function calls `I*.Stub.asInterface()`. Using the `Proxy` type of the parameter of `asInterface()`, it queries the hash table to obtain the corresponding system service class name. Based on the system service type name, CENTAUR searches in the migration database to locate the snapshot that contains the system service and obtain the reference value of the system service object. (Note that to correctly distinguish references values of different snapshots, each reference value is extended into the form of `pid:reference_value`, where `pid` refers to the ID of the process from which the snapshot is captured.) After that, `mService` is assigned with the obtained reference value and its `snapshotRef` attribute is set to `true`. Finally, when `mService` is accessed, the corresponding system service object is migrated from the concrete world to the symbolic execution world, such that, the target service method can be analyzed under a correct execution context.

**Example.** Figure 8 shows an inter-process system service call example, where the `Telecom Account Registry` (running in the `com.android.phone` process) invokes `getAllPhoneAccountHandles()` exposed by the `Telecom Service` (running in the `com.android.server.telecom` process). The call at Line 17 invokes the function at Line 22, which first obtains the `Proxy` object (Line 23) and then issues an inter-process service call through the proxy at Line 24. Our system intercepts the `asInterface` call to return the reference value that points to the `Telecom Service` object. When `mService` is accessed (Line 24), the system service object is migrated and the service call can be symbolically executed.

A small number of system services do not use AIDL to generate their `Stub/Proxy` classes; instead, manually implemented custom classes are provided. One example is the `ActivityManagerService` (AMS), whose interface is also called from the native code; thus, a manual implementation of its `Stub/Proxy` is provided. We then can add the mapping between the `Proxy` and the corresponding system service class name into the hash table aforementioned.

Through this, the intricate `Binder` mechanism that cannot be analyzed by the symbolic executor can be successfully handled without losing the execution context information.

## 7 OTHER IMPLEMENTATION DETAILS

### 7.1 Background on SPF

We built the symbolic executor based on *Symbolic PathFinder* (SPF) [50], a symbolic execution framework on top of *Java PathFinder* (JPF) [60]. SPF can be understood as a non-standard Java bytecode interpreter, which enforces path exploration when interpreting the code.

SPF can be extended by overriding methods that are used to interpret bytecode instructions. It also supports the interception of arbitrary function calls for customized handling during the analysis. Specifically, JPF provides a mechanism called *Model Java Interface* (MJI) that intercepts method invocations for custom handling. CENTAUR makes use of MJI to intercept certain method calls (e.g., `getCallingUid`, `getPackageName`, and the `get` functions of various collection data structures), and redirects them to our custom implementation of these functions. Finally, attributes can be added to associate with each of the class/object fields on the heap and variables on the stack to record and track states of interest, such as taints and symbolic expressions.

We added 7,419 lines of code for implementing CENTAUR through extending SPF. Significant effort has been saved by building upon SPF, thanks to the decoupled design.

### 7.2 Classpath

The Java source code in Android is compiled into `.jar` files, which comprise standard `.class` files. The classpath below shows the classes analyzed by the symbolic executor.

```
classpath=test_driver_dir;\
services_intermediates/classes-full-debug.jar;\
framework_intermeidates/classes-full-debug.jar;\
core-libart_intermediates/classes-full-debug.jar
```

The first line specifies the directory containing the test driver, the next two lines specify the Android Framework code, and the last line the core libraries of ART.

### 7.3 Slim Tainting

Slim tainting (Section 4.4) is built into the symbolic executor by modifying the interpretation of instructions, such as `isub` (subtraction), `irem` (modular), and `*aload`, and intercepting functions, such as `getPackageName`, `getCallingUID`, `String.concat` (string concatenation), and various `get` functions of collection data structures. For each field, array element, and call-stack variable, CENTAUR adds one attribute indicating the taint and another indicating the symbolic input property. Through interception of function calls and

adding field attributes, our implementation does not need to modify or annotate the framework code, which means that no maintenance efforts are needed when new versions of Android are released.

### 7.4 Capturing and Parsing Snapshots

After a heap memory snapshot of an Android Framework process is captured (using the `dumpheap` utility), it is first converted to a standard `.hprof` file using the `hprof-conv` utility in the Android SDK. Next, the `.hprof` file is parsed to extract the list of classes and objects stored using a hprof file parser [33]. The extracted information is then organized into the memory space of the execution context query server.

### 7.5 Dealing with Messaging

Two messaging mechanisms that are frequently used by system services are `Message Handlers` and `State Machines`. A `Message Handler` is associated with a thread's message queue, and is used to send messages to the message queue and handle them as messages come out of the queue [32]. Message Handlers are implemented through `Binder`, so they cannot be interpreted by our symbolic executor directly. To deal with them, we propose to replace the call to `sendMessage(message)` with a call to the destination handler's `handleMessage(message)`. Our symbolic executor interposes the `invokevirtual` instruction and enforces the replacement on the fly.

A `State Machine` can also be used to send and process messages. It allows processing of messages depending on the *current state* of the associated model. A State Machine sends a message by invoking `sendMessage`, while the current state's `processMessage` is invoked when a message is processed. Thus, it is critical to identify the current state. The State Machine object contains a field that points to the `mSmHandler` object, two fields of which, `mStateStack` and `mStateStackTopIndex`, are used to find the *current* state (= `mStateStack[mStateStackTopIndex].state`). To handle State Machines, our symbolic executor replaces the call to `mSmHandler.sendMessage(message)` with a call to the current state's `processMessage(message)`. This way, we connect the senders and receivers for messages sent through State Machines.

### 7.6 Handling JNI Calls

The framework invokes native code through the *Java Native Interface* (JNI) mechanism. Multiple ways are adopted to handle JNI calls during symbolic execution. **(1)** Methods that return the calling UID (`getCallingUid`) and the package name (`getPackageName`) of the client app are modeled to return the corresponding information of the malicious app. Their return values are set as taint sources (Section 4.4). **(2)** The return values of other native methods that return app-specific information of the malicious app are specified as symbolic inputs; e.g., native methods declared in the package `android.content.res` return app-specific information. **(3)** Other calls to native methods are delegated back to Android through remote procedure calls (RPCs). The RPC client in the symbolic executor is built similar to `jpf-nhandler` [55]. While `jpf-nhandler` delegates native calls to a host JVM, ours delegates them to an app running as an RPC server in a remote Android system (Figure 2), which issues native calls using reflection on demand. The `GSON` library [31] is used for marshalling and unmarshalling method parameters and return values, which are transmitted between the RPC server and client via socket.

## 8 Evaluation

### 8.1 Experiment Overview

We first compare CENTAUR against under-constrained symbolic execution (UCSE) in Section 8.2. Both can start symbolic execution from system interface methods to reach the code deep in a program, but CENTAUR makes use of the execution context provided by concrete execution. We should also compare CENTAUR against symbolic execution that starts directly from the main entry of Android Framework (i.e., `SystemServer.main`), but note that our symbolic executor runs outside Android, and it is unlikely to initialize the framework outside the Android environment, since the initialization phase of the framework heavily relies on the Android environment, such as the file systems and other supporting processes. Concolic execution also makes use of concrete execution to assist symbolic execution. Specifically, it uses some input to run a program in order to collect the symbolic constraints along the concrete execution and then negates the constraints to explore other paths. Ideally, we should also compare CENTAUR with concolic execution. However, we are not aware of any tools that support concolic execution of Android Framework. Enormous effort would be required to enable it, as the concolic execution tool has to be able to track the very complex framework initialization phase to precisely collect the symbolic constraints. We thus compare CENTAUR with UCSE only.

Second, CENTAUR provides strong support for vulnerability discovery and exploit generation. To demonstrate this, we investigate two distinct types of recently uncovered attacks that exploit Android Framework vulnerabilities and show the results in Sections 8.3 and 8.4.

Finally, the reliability of the approach is investigated. We present exploit generation experiments based on heap memory snapshots captured at different times, and analyze the consistency of the results in Section 8.5.

The experiments were performed on a machine with an Intel Core i7 4.0Ghz Quad Core processor and 32GB RAM running the Linux kernel 3.13. Exploits were verified on different versions of Android systems from 4.0 to 9.0.

### 8.2 Comparison with Under-constrained Symbolic Execution (UCSE)

The first issue of applying UCSE to Android Framework is that virtual function calls are frequently used in the framework code, but the runtime types of the receiver objects are unknown. UCSE constructs the receiver objects based on the type hierarchy or manual specifications, which either explores spurious paths or requires much manual effort.

The second issues is that input variables which are treated as concrete inputs in CENTAUR are treated as symbolic inputs in UCSE. E.g., `LocationManagerService.mProviders` in Figure 1 is an ArrayList instance that stores the geographical location providers. It is a non-app-specific variable, and

is treated as a concrete input in CENTAUR by migrating its value from the heap snapshot. But UCSE treats such variables as symbolic inputs and handles them using lazy initialization, which causes the following problems: (1) loops that iterate through collection data structures are unbounded, and (2) the generated concrete values of such variables may be unrealistic and difficult to interpret.

We tried to perform UCSE of Android Framework using *Symbolic PathFinder*, which kept crashing when it was applied directly. We spent a lot of time and effort modifying the framework code (e.g., adding the type information about objects pointed to by references to assist dynamic dispatching) to make UCSE possible. We thus only modified the code with respect to the `getProviders` API and `startActivityUncheckedLocked` API (related to task hijacking attacks described in Section 8.4). UCSE spent 138m when analyzing `getProviders` and ran out of memory in the case of `startActivityUncheckedLocked`, while CENTAUR finished them within 26s and 42m37s, respectively.

Thus, path exploration without precise information of the execution context causes many problems. CENTAUR resolves the problems by migrating the execution context from the concrete execution world to the symbolic execution world.

### 8.3 Investigating Inconsistent Security Policy Enforcement (ISPE)

#### 8.3.1 Background

Android Framework utilizes a permission-based security model, which provides controlled access to various system resources. However, a sensitive operation may be reached from different paths, which may enforce security checks inconsistently. As a result, an attacker with insufficient privilege may perform sensitive operations by taking paths that lack security checks. Recently, static analysis combined with manual code inspection has been applied to finding such inconsistent security enforcement cases in Android Framework [56]. The system, called *Kratos*, first builds a call graph based on the framework code. With the call graph, it finds all the paths that can reach sensitive operations, and then compares these paths to identify paths that reach the same sensitive operation with inconsistent security checks enforced, and reports them as suspect ISPE vulnerabilities.

#### 8.3.2 Combined Approach for Bug Finding

While static analysis is very scalable, it is well known that the analysis results may be imprecise. In the case of finding ISPE bugs, static analysis based on the reachability analysis may report false positives, as some paths are infeasible in real executions. Currently, manual effort is used to scrutinize the code, which is laborious and tedious; moreover, it is difficult to verify the correctness of the manual inspection results.

We propose to combine static analysis and symbolic execution to find ISPE bugs. For each suspect vulnerability reported by static analysis, CENTAUR (1) finds all feasible paths that reach the sensitive operation, (2) gives permissions needed for each feasible path (the needed permissions are included in each path condition), (3) verifies permission consistency among the feasible paths, and (4) generates inputs that exercise the feasible paths to verify suspect vulnerabilities. All the steps are performed automatically, in contrast with the previous work that relies on tedious and error-prone manual inspection. Plus, zero false positives are guaranteed as all suspect vulnerabilities are validated.

**Skeleton App.** We use a *skeleton app* to act as the malicious app; it contains all the aspects of a regular app, including the manifest file, activities, and services, but does not implement specific functionalities. Specifically, the *skeleton app* borrows the manifest file from the Android developer website, which has *"every element that it can contain"* [30]. In practice, the analyst can instead choose any app as the malicious app.

**Result Summary.** Table 3 summarizes the experiment results for Android Framework 5.0 (the vulnerability shown in the last row is discussed in Section 8.4). We also examined each vulnerability on Android Framework 9.0; due to space limit, we omit the table showing the results. In Android Framework 9.0, all the vulnerabilities, except the third one, still exist. Android 9.0 has fixed the third vulnerability: the same permission, `MODIFY_PHONE_STATE`, is required by the two entrypoints. For each vulnerability, the table lists the vulnerability description, entrypoint(s), the min/max number of migrated classes among different paths, the min/max number of migrated objects among different paths, the number of sets of concrete values generated ("—" means it can be exploited unconditionally; note that we generate one set of concrete values for each *unique* path explored), the number of sets that can be used to generate exploits, the symbolic execution time, and the code coverage.

Given an entrypoint method, there may be multiple paths that can reach the sensitive operation, and the classes and objects involved in the paths may vary, as illustrated by the min/max number of migrated classes and objects. Note that when migrating a class, all its super classes are also migrated, which is the reason the number of migrated classes is larger than that of objects. For most cases, the symbolic execution of an entrypoint method is finished within less one minute. Note that in some cases we have a relatively *low code coverage*, e.g., in `WSI.addOrUpdateNetwork()`; it is mainly because branches that rely on non-app-specific variables are not explored, as we consider those variables as concrete inputs. We are only interested in branches that can be affected by the variables derived from the malicious app. We regard this (i.e., lower code coverage due to only exploring branches depending on app-specific variables) as an advantage for improving the analysis scalability and speed, but also discuss its limitation in Section 10.

**New Findings.** It is notable that some of our results are inconsistent with those of Kratos. First, for the fifth vulnerability in Table 3, Kratos reports that it does not exist in Android Framework 5.0, while CENTAUR shows that it still exist in the version 5.0 and 9.0 (i.e., different permissions are required by the two system interface methods for reaching the sensitive resource) and the result is verified by the log. Second, for the sixth vulnerability in Table 3, Kratos reports only one permission `CONNECTIVITY_INTERNAL` for invoking `NsdService.setEnabled`, while CENTAUR reports two permissions, `CONNECTIVITY_INTERNAL` and `WRITE_SETTINGS`. The more thorough and accurate results demonstrate the advantages of the hybrid approach.

TABLE 3
List of vulnerabilities and analysis statistics. (*LMS*, *TSI*, *PIM*, *WMS*, *AMS*, *WSI*, *NS*, and *ASS* represent LocationManagerService, TelecomServiceImpl, PhoneInterfaceManager, WindowManagerService, ActivityManagerService, WifiServiceImpl, NsdService, and ActivityStackSupervisor, respectively.)

| No. | Vulnerability description | Entrypoint(s) | # of migrated classes | | # of migrated objects | | # of all sets | # of legal sets | Analysis time | Code coverage (%) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | min | max | min | max | | | | |
| 1 | Access installed providers with insuf. privilege | LMS.getAllProviders()<br>LMS.getProviders(Criteria,boolean) | 55<br>77 | 55<br>93 | 4<br>14 | 4<br>42 | —<br>66 | —<br>66 | 19s<br>26s | 92.3<br>45.8 |
| 2 | Read phone state with insuf. privilege | TSI.getCallState()<br>TSI.isInCall()<br>TSI.isRinging() | 48<br>62<br>60 | 48<br>69<br>65 | 3<br>17<br>16 | 3<br>20<br>18 | —<br>1<br>1 | —<br>1<br>1 | 14s<br>32s<br>35s | 72.4<br>83.5 |
| 3 | End phone calls with insuf. privilege | TSI.endCall()<br>PIM.endCall() | 81<br>80 | 83<br>85 | 21<br>23 | 24<br>26 | 1<br>1 | 1<br>1 | 29s<br>38s | 91.3<br>89.4 |
| 4 | Close system dialogs with insuf. privilege | WMS.closeSystemDialogs(String)<br>AMS.closeSystemDialogs(String) | 57<br>63 | 57<br>67 | 6<br>11 | 6<br>15 | —<br>2 | —<br>2 | 17s<br>29s | 69.1<br>56.0 |
| 5 | Set up HTTP proxy working in PAC mode with insuf. privilege | WSI.addOrUpdateNetwork()<br>WSI.getWifiServiceMessenger() | 67<br>65 | 122<br>84 | 23<br>21 | 52<br>24 | 16<br>1 | 16<br>1 | 26s<br>18s | 30.4<br>57.3 |
| 6 | Enable/Disable mDNS daemon with insuf. privilege | NS.setEnabled(boolean)<br>NS.getMessenger() | 75<br>80 | 114<br>81 | 28<br>11 | 53<br>14 | 1<br>1 | 1<br>1 | 44s<br>18s | 47.2<br>62.4 |
| 7 | Task hijacking | ASS.startActivityUncheckedLocked() | 324 | 387 | 136 | 182 | 2,020 | 810 | 42m 37s | 58.3 |

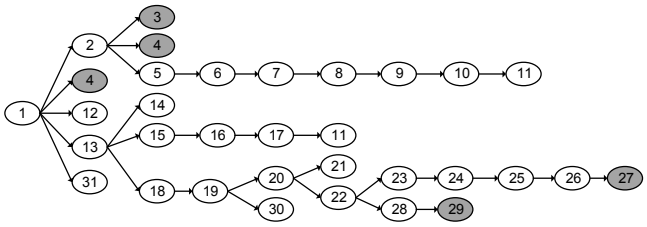


Fig. 9. Sub-call graph rooted at `getProviders()`. (*LMS*, *AM*, *AMS*, and *PMS* represent `LocationManagerService`, `ActivityManager`, `ActivityManagerService`, and `PackageManagerService`, respectively. The grey nodes denote native methods.)

### 8.3.3 Symbolic Execution within a Single Process

As an example, we describe how the combined approach was applied to investigating the first vulnerability in Table 3. All involved method calls are performed in the `system_server` process. (1) First, the static analysis based on path reachability and pairwise path comparison finds that both `getProviders()` and `getAllProviders()` (in the `LocationManagerService` class) have paths reaching the same sensitive operation that returns the names of the installed GPS providers, and the two paths can be executed with inconsistent permissions; thus, it is a suspect vulnerability. (2) Next, CENTAUR is applied to automatically checking whether there exist *feasible* paths that can reach the sensitive operation from the two service interface methods.

**Entrypoint 1**: `getProviders()`. Figure 9 shows the sub-call graph rooted at this entrypoint with collection and string operations omitted. It leads to invocation of multiple methods of other services, e.g., `ActivityManagerService` and `PackageManagerService`. These services run in the same process, so are handled as ordinary method calls using the runtime type information in the execution context.

Fig. 10. Examples of concrete input values generated for the first vulnerability Table 3.

```
(mUserIds.array[54].grantedPermissions.backingMap.
    table[836059052 & (length_SYM - 1].key_SYM ==
    permission.ACCESS_FINE_LOCATION) &&
(criteria != null) && (enabledOnly = false)
(criteria.mHorizontalAccuracy == 2) &&
(criteria.mPowerRequirement == 0) &&
(criteria.mAltitudeRequired == false) &&
(criteria.mSpeedRequired == false) &&
(criteria.mBearingRequired == false) &&
(criteria.mCostAllowed == false)
//output: ["gps"]
```

```
(mUserIds.array[54].grantedPermissions.backingMap.
    table[836059052 & (length_SYM - 1].key_SYM ==
    permission.ACCESS_FINE_LOCATION) &&
(criteria == null) && (enabledOnly = true)
//output: ["gps", "passive"]
```

Four native methods are involved: `getCallingUid()`, `getCallingPid()`, `native_get_long()`, and `getuid()`. Calls to the methods are intercepted using `MJI` and are redirected to our handlers of these methods. The first two return the UID and PID of the client app, respectively, and `getuid()` returns UID $= 1000$, which is the UID of the `system_server` process. The call to `native_get_long` is delegated back to the Android system through RPC.

Figure 10 shows two sets of generated concrete values. The variable `mUserIds.array[54]` is identified as a symbolic input through slim tainting during symbolic execution (in this example, the app's UID is 10,054; 54 is due to the formula $(uid\%100,000 - 10,000)$ presented in Section 4.3). Take the first set as an example; it provides clear information for building an app in terms of how to configure the app (i.e., requiring the `ACCESS_FINE_LOCATION` permission) and prepare the parameter values (i.e., `criteria` and `enabledOnly`) for invoking the entrypoint method in order to exercise the path that reaches the sensitive operation.

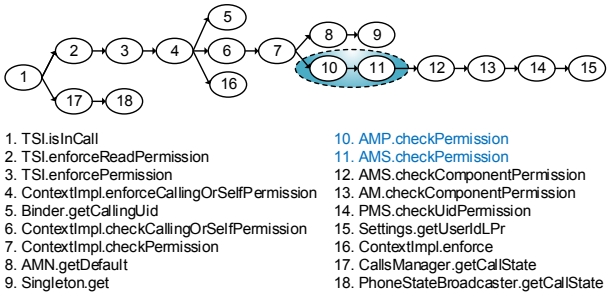


Fig. 11. Sub-call graph rooted at `isInCall()`. (*TSI*, *AMN*, *AMP*, *AMS*, and *PMS* represent `TelecomServiceImpl`, `ActivityManagerNative`, `ActivityManagerProxy`, `ActivityManagerService`, and `PackageManagerService`, respectively. The blue area denotes an IPC service call made through `Binder`.)

**Entrypoint 2**: `getAllProviders()`. The generated path condition is constantly true, which means this method can be invoked with no permissions needed.

As the needed permissions required by the two entry-points differ, it is identified as an ISPE vulnerability.

#### 8.3.4 Symbolic Execution Involving Multiple Processes

We now describe how CENTAUR was applied to analyzing multiple processes, using the second vulnerability in Table 3 as an example.[5] Starting from Android 4.4, the `Telecom` service runs in the `com.android.server.telecom` process, rather than the `system_server` process. Again, the combined approach is applied in two steps. (1) First, the static analysis based on path reachability and pairwise path comparison finds that `getCallState()`, `isInCall()`, and `isRinging()` (in the `TelecomServiceImpl` class) have paths reaching the sensitive operation that returns the current phone state. (2) Second, CENTAUR is applied to automatically checking whether there exist *feasible* paths that can reach the sensitive operation from the three methods.

**Entrypoint 1**: `isInCall()`. Figure 11 shows the sub-call graph rooted at `isInCall()` with collection and string operations omitted. It invokes a service method (nodes 10 and 11) of `ActivityManagerService` running in the `system_server` process; thus, the invocation is made via `Binder`. To avoid interpreting the `Binder` mechanism (see Section 6.2), CENTAUR handles the inter-process service call as follows. First, the reference variable that points to a `Proxy` object, which is used to invoke `ActivityManagerService`, is set to a reference value that points to `ActivityManagerService`. Second, when the reference variable is used, `ActivityManagerService` object is migrated to the symbolic execution world; after that, the service method `checkPermission` of `ActivityManagerService` is handled as an ordinary function call. In effect, the call on the `Proxy` object (node 10) is skipped, and node 7 is directly connected to node 11.

The generated concrete value is showed below. It provides information on how to configure the app, i.e., requiring the `READ_PHONE_STATE` permission.

```
(mUserIds.array[54].grantedPermissions.backingMap.
    table[1].key == permission.READ_PHONE_STATE)
//output: false
```

5. In [44], we used much manual effort to handle multiple processes.

Fig. 12. `startActivityUncheckedLocked()`.

```
final int startActivityUncheckedLocked(
    ActivityRecord r, ActivityRecord sourceRecord,
    IVoiceInteractionSession voiceSession,
    IVoiceInteractor voiceInteractor, int
    startFlags, boolean doResume, Bundle options,
    TaskRecord inTask) {...}
```

**Entrypoint 2**: `isRinging()`. The result of analyzing `isRinging()` is the same as that of `isInCall()`.

**Entrypoint 3**: `getCallState()`. The generated path condition is constantly true, which means this method can be invoked with no permissions needed.

As the needed permissions required by the three entry-points differ, it is identified as an ISPE vulnerability.

**Summary.** Compared to previous work that relies on tedious and error-prone manual inspection, the approach combining call graph reachability analysis and symbolic execution eliminates the need for manual work and guarantees zero false positives. It is potential to apply this approach to finding other types of vulnerabilities in Android Framework.

### 8.4 Investigating Task Hijacking Attacks

#### 8.4.1 Background

The *Activity Manager Service* (AMS) allows activities of different apps to reside in the same *task*, which is a collection of activities that users interact with when performing a certain job. The activities in a given task are arranged in a *back stack*, pushed in the order they were opened; users can navigate back using the "*Back*" button. This feature can be exploited by a malicious app if its activities are manipulated to reside side by side with the victim apps in the same task and hijack the user sessions of the victim apps.

This is a design flaw rather than a program bug, but can be exploited to implement UI spoofing, denial-of-service, and user monitoring attacks [54]. E.g., a malicious app may start a malicious activity that impersonates the victim activity, and the UI spoofing attack succeeds if the fake activity resides in the same back stack as the victim activity, and the user may mistake the fake malicious activity for the victim one. This case illustrates unique characteristics of exploits that take advantage of Android Framework vulnerabilities: *the malicious "input" is not some single input (e.g., a command parameter, a network packet, etc.), but a whole app.*

#### 8.4.2 Vulnerability Discovery

We use the `EditEventActivity` activity of the `calendar` app as an example victim activity. In the skeleton app, the main activity starts the malicious activity, denoted as $M$. The goal of the attack is that $M$, when it is started, will reside in the same task as the victim activity. A bug is identified if such attacks against the victim activity is feasible. We capture the heap memory snapshots when the victim app and the skeleton app are started and the main activity of the skeleton app is to start the malicious activity.

While the method for starting an activity is `startActivity`, the task selection is done in `startActivityUncheckedLocked`, which is invoked by `startActivity`. We thus performed the symbolic execution of `startActivityUncheckedLocked` to simplify the path exploration; it has eight parameters

Fig. 13. An example set of concrete input values for task hijacking.

```
(r.intent.mFlags == 0x10080000) &&
(r.launchMode == LAUNCH_SINGLE_TASK) &&
(r.mLaunchTaskBehind == true) &&
(options == null) && (r.resultTo == null) &&
(r.info.documentLaunchMode == 0) &&
(r.info.targetActivity == null) &&
(r.taskAffinity == "android.task.calendar")
```

Fig. 14. Task hijacking exploit example.

```
// Snippet of AndroidManifest.xml
<activity android:name=".maliciousActivity"
     android:launchMode="singleTask"
     android:taskAffinity="android.task.calendar"
     android:documentLaunchMode="none" />
// The main activity starts the malicious activity
public void onCreate(Bundle savedInstanceState) {
  Intent i= new Intent(this,maliciousActivity.class);
  intent.setFlags(0x10080000);
  // null is due to "options == null"
  startActivity(i, null);
}
```

TABLE 4
Effectiveness of the generated exploits on different Android versions.

| Android version | 4.0 | 4.1 | 4.2 | 4.3 | 4.4 | 5.0 | 9.0 |
|---|---|---|---|---|---|---|---|
| # of effective exploits | 434 | 674 | 674 | 674 | 702 | 810 | 734 |

as shown in Figure 12. The first parameter `r` is an `ActivityRecord` instance storing the information of $M$, while the second storing that of the caller activity. The description of other parameters is omitted. They are set to symbolic inputs. The constraint indicating that *the task selected for $M$ is exactly the one hosting the victim activity* is added to each of the path conditions when it is to be resolved; that is, $\langle m.task.taskId == v.task.taskId \rangle$, where $m$ and $v$ represent the `activityRecords` of the malicious activity and the victim activity, respectively. A feasible path is found if the path condition is resolvable.

#### 8.4.3 Exploit Generation

The symbolic execution generated 2,020 sets of concrete input values (each set corresponds to a *unique* path), among which some contain illegal concrete values, e.g., due to requiring the malicious activity's package name and activity name to be equal to those of the victim activity. Simple scripts were written to filter out illegal concrete values (1,210 sets totally). Figure 13 shows an example of legal concrete values. In this example, *r.intent.mFlags* and *options* (whose type is `Bundle`) guide how to set the two parameters of `startActivity(Intent, Bundle)`, respectively, and others instruct how to configure the malicious activity; e.g., `r.launchMode` is mapped to the `android:launchMode` in the manifest file. Figure 14 shows the exploit generated according to the set of concrete values, and it has verified that the exploit can be used to launch task hijacking successfully.

We then examined whether the exploits generated on Android 5.0 were effective on other versions of Android systems. Table 4 lists the results, which show that the effectiveness of the exploits are affected by the versions of Android systems. Further investigation has revealed that the difference is mainly caused by code changes. For example, the new exploiting condition `FLAG_ACTIVITY_NEW_DOCUMENT` is not introduced until Android 5.0 (discussed below); the `startActivity(Intent, Bundle)` API is not included in version 4.0, and thus only exploits with `options == null` can be used for invoking `startActivity(Intent)`. For Android 9.0, a new variable `matchedByRootAffinity` is introduced to control whether or not the vulnerability can be exploited, causing some exploits generated in Android 5.0 ineffective in Android 9.0.

**Newly Discovered Exploiting Condition.** The path conditions generated from symbolic execution reveal an extra exploiting condition (requiring a specific bit in the bitflags $r.intent.mFlags$ to be 0) that was not reported in previous work [54]. Compared to previous work that relies on *ad hoc* manual effort for discovering the exploiting conditions, CENTAUR finds them in a systematic and automatic way.

### 8.5 Consistency of Exploits with Different Snapshots

We then investigated whether snapshots captured at different times affected exploit generation. After the system was initialized, 20 snapshots were captured at intervals of 5 minutes on Android 5.0 with random user interactions during the intervals. For each vulnerability listed in Table 3, symbolic execution was performed with each of the 20 snapshots providing the execution context. The results show that, for each vulnerability, the same sets of path conditions were generated with different snapshots, which means that the resulting exploits with the different snapshots are consistent.

There are several reasons that explain the consistency of exploits. First, if a malicious app does not rely on other apps to exploit a vulnerability (e.g., inconsistent security policy enforcement), access control is enforced in the framework to make sure the information of other apps is not accessed. Thus, the configurations and statuses of other apps do not affect the path exploration. On the other hand, for exploits that rely on the statuses of other apps (e.g., the victim app in task hijacking attacks), the path exploration may depend on the statuses of other apps; hence, during symbolic execution, reasonable setting up is established consistently; e.g., in the task hijacking case, the victim activity should already be started prior to capturing snapshots. The results show that an attack succeeds as long as the same statuses recur.

Finally, recently revealed attacks that exploit the framework do not rely on system-specific configurations. Take the ISPF vulnerability that accesses the names of installed GPS providers as an example; the exploit does not depend on the concrete values of the related non-app-specific variable (i.e., `LocationManagerService.mProviders`), although different provider names may be returned by the service calls when different providers are installed.

## 9 Related Work

**Concrete-Execution-Assisted Symbolic Analysis.** DART is the first concolic testing tool that uses symbolic analysis in concert with concrete execution to improve code coverage [27]. It runs the tested unit code on random inputs and symbolically gathers constraints at decision points that use input values; then, it negates one of these symbolic constraints to generate the next test case. DART [27], EXE [9], and $S^2E$ [15] all use concrete execution to *handle uninterpreted functions (e.g., system calls)*. In CENTAUR, concrete execution is not only used for handling uninterpreted functions (i.e., JNI calls), but also *for the initialization of Android Framework in order to obtain the execution context for symbolic execution.*

**Switching Concrete Execution to Symbolic Execution.** Symbolic PathFinder (SPF) begins with concrete execution and can switch to symbolic execution at any point in the program [49]. $S^2E$ can also start with concrete execution and then selectively analyze a function of interest; compared to SPF, it allows specifying the symbolic inputs initially and then gathers constraints during concrete execution. Due to the complexity of Android Framework, it is difficult to track the symbolic data derived from the target app throughout its initialization phase. *We propose* slim tainting *to identify variables derived from the target app as symbolic inputs* on the fly during *symbolic execution. It is a novel symbolic analysis approach to handling such complex middleware as Android Framework.*

**Vulnerability Discovery and Exploit Generation via Symbolic Execution.** Many systems demonstrate that symbolic execution is useful for finding vulnerabilities from Windows programs [27], Linux/Unix programs [8], [9], [15], Java programs [50], firmware [22], [65], while CENTAUR demonstrates it on a large piece of middleware. Many challenges such as handling analysis scalability and complex features of the framework code arise and are overcome in CENTAUR. AEG performs exploit generation given vulnerable Unix programs [4]. APEG generates exploits based on information in patches [7]. Instead of generating an exploit as a network packet, a string, or a parameter value, in our case we consider an exploit as a whole malicious application consisting of not only the exploit code but also its configuration file.

**Symbolic Execution of Android Apps.** There has been a lot of work that performs symbolic execution of Android apps for test input generation or security purposes [1], [37], [46], [47], [52], [63], [64], [68]. E.g., Anand et al. proposed a system based on concolic testing for generating screen tap events to exercise Android apps [1]. Jensen et al. proposed to use concolic execution to build summaries of event handlers and generate event sequences backward, to find event sequences that reach a given target line of code in an Android app [37]. SIG-Droid combines program analysis techniques with symbolic execution to generate event sequences [46]. Compared to analyzing *applications*, the analysis of the framework code raises many unique challenges, which require new insights, ideas and techniques. *To our knowledge,* CENTAUR *is the first that supports symbolic execution of Android Framework.*

**Analysis of Android Framework.** Analysis of Android Framework has been very limited. Prior work has performed fuzzing [10], [26], [35] and some static analyses of the framework code [3], [5], [6]. They are used for applications such as inferring the Android permission specification and probing system services; however, more powerful analysis capabilities, such as symbolic execution, are absent. There is also work that generates summaries/syntheses of Android Framework APIs for the purpose of, e.g., taint analysis or symbolic execution of *Android apps* [2], [24], [59], or investigation of ICC based attacks [14], [21], [42]. None is able to perform symbolic analysis of *Android Framework*. Indeed, the literature has recognized the prominent challenges for symbolically executing Android Framework; e.g., Jeon et al. pointed out that "*Frameworks are large, complicated*" and turned to synthesize the framework behavior to facilitate symbolic execution of *apps* [38]. CENTAUR takes a big step towards more in-depth analysis of Android Framework.

## 10 DISCUSSION

CENTAUR is shown to be effective for discovering two distinct types of vulnerabilities in Android Framework and generating PoC exploits. The reason we chose ISPE and task-hijacking vulnerabilities as the examples is that they are both due to *high-level* logic errors and design flaws, which have been less explored by the research community. Besides, it is potential to apply CENTAUR to discovering other common bugs, such as `BufferOverflowException` and `NullPointerException`. Moreover, CENTAUR can be used to generate test inputs (e.g., unusual apps) for testing the system services.

A lot of work performs symbolic execution of Android *apps* for test input generation or security purposes [1], [37], [46], [47], [52], [63], [64], [68]. As apps frequently interact with the framework, one challenge for such tasks is to deal with the complex framework code. To handle this, existing work either models the framework APIs by synthesizing the framework behavior, which is time consuming and error prone, or simply sets the return values of a framework API as symbolic variables, which introduces significant imprecision. Thus, an interesting solution may be cross-layer symbolic execution that integrates our technique and these systems for more precise analysis.

CENTAUR performs symbolic execution of bytecode, and does not rely on the availability of source code. Its migration algorithm is not to simply copy raw data (i.e., the bytes) from the heap snapshot to the symbolic executor; instead, it has an anatomical view of the heap when copying the objects and their fields, and properly updates the reference fields. Thus, the migration algorithm works for all versions of Android.

**Limitations.** CENTAUR uses the concrete values of the non-app-specific variables during path exploration. This avoids exploring many paths unnecessarily, such that it could attain a high scalability. But it may introduce false negatives, as other values of those non-app-specific variable are not considered. In some attack scenarios, attackers may exploit certain system settings, which lead to different values for non-app-specific variables. To investigate such attacks, we suggest researchers capture multiple heap snapshots under different settings for multiple rounds of analysis or consider related non-app-specific variables as symbolic.

The access patterns were found via manual code review (Section 4.3). We did not analyze all the framework code, and consider the access patterns as incomplete heuristic. We plan to explore a systematic way of analyzing all the framework code to verify the found access patterns as future work.

The creative combination of concrete execution and symbolic execution allows the analysis to start from any of the framework APIs, rather than tackling the millions of lines of code as a whole; thus, the path explosion problem is greatly mitigated. A follow-up question is how to find the APIs that can reach the target of the attack under investigation. This can be done either by static program analysis (e.g., ISPE vulnerability) or based on expert knowledge (e.g., task-hijacking vulnerability). Like our work, other symbolic execution methods that begin with concrete execution and then switch to symbolic execution when a function of interest is triggered also assume the function is known by the security analysts [15], [49].

## 11 Conclusions

We have introduced the first system, called CENTAUR, for symbolic execution of Android Framework. To improve the analysis scalability, instead of analyzing the framework as a whole, we propose to analyze the system service interface methods separately; moreover, a proper execution context is provided to avoid under-constrained symbolic execution. Among the large number of variables in the execution context, *slim tainting* is proposed to precisely identify variables derived from the malicious app as symbolic inputs, which benefit the analysis completeness and scalability. In order to decouple the implementation of CENTAUR from Android, the execution context provided by concrete execution is migrated from the Android system to the symbolic executor. Moreover, CENTAUR is able to handle many unique features of the framework for precise symbolic execution, such as inter-process service calls and messaging. We have implemented the system and evaluated it. The evaluation shows that CENTAUR is very effective in both vulnerability discovery and exploit generation. Given that symbolic execution has proven to be a very useful technique, CENTAUR can also be applied to automatic API specification generation, fine-grained malware analysis, and testing.

## Acknowledgments

Dr. Lannan Luo was supported by NSF CNS-1850278 and NSF CNS-1815144. Dr. Qiang Zeng was supported by NSF CNS-1856380. Dr. Peng Liu was supported by ARO W911NF-13-1-0421 (MURI), NSF CCF-1320605, NSF SBE-1422215, NSF CNS-1422594, and NSF CNS-1505664. Dr. Kai Chen was funded by NSFC U1536106, 61100226, Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701). Dr. Jian Liu was funded by NSFC No. 61572481. Dr. Min Yang was funded by the National Program on Key Basic Research (NO.2015CB358800) and NSFC U1636204.

## References

[1] S. Anand, M. Naik, H. Yang, and M. J. Harrold. Automated concolic testing of smartphone apps. In *FSE*, 2012.
[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI*, 2014.
[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the android permission specification. In *CCS*, 2012.
[4] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *Communications of the ACM*, 2014.
[5] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *USENIX Security*, 2016.
[6] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.
[7] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *USENIX Security*, 2008.
[8] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.
[10] C. Cao, N. Gao, P. Liu, and J. Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *ACSAC*, 2015.
[11] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.
[12] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *ICSE*, 2014.
[13] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, WeiZou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale. In *USENIX Security*, 2015.
[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.
[15] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.
[16] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, 2007.
[17] CVE. CVE-2015-6628. https://www.cvedetails.com/cve/CVE-2015-6628/.
[18] CVE. CVE-2016-2496. https://www.cvedetails.com/cve/CVE-2016-2496/.
[19] CVE. CVE-2016-3750. https://www.cvedetails.com/cve/CVE-2016-3750/.
[20] CVE. CVE-2016-3759. https://www.cvedetails.com/cve/CVE-2016-3759/.
[21] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *International Conference on Information Security*, pages 346–360, 2010.
[22] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.
[23] A. D. Documentation. Binder. https://developer.android.com/reference/android/os/Binder.html.
[24] W. Enck, 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 *OSDI*, 2010.
[25] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS*, 2009.
[26] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.
[27] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.
[28] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
[29] Google. Android Interfaces and Architecture. https://source.android.com/devices/.
[30] Google. App Manifest. https://developer.android.com/guide/topics/manifest/manifest-intro.html.
[31] GSON. https://sites.google.com/site/gson/Home.
[32] Handler. https://developer.android.com/reference/android/os/Handler.html.
[33] HPROF Parser. https://github.com/eaftan/hprof-parser.
[34] H. Huang, S. Zhu, K. Chen, and P. Liu. From system services freezing to system server shutdown in android: all you need is a loop in an app. In *CCS*, 2015.
[35] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *ISSRE*, 2017.
[36] IDC. Smartphone OS Market Share, 2016. https://www.idc.com/prodserv/smartphone-os-market-share.jsp.
[37] C. S. Jensen, M. R. Prasad, and A. Moller. Automated testing with targeted event sequence generation. In *ISSTA*, 2013.
[38] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing framework models for symbolic execution. In *ICSE'16*.
[39] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.
[40] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
[41] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: detecting inter-component privacy leaks in android apps. In *ICSE'15*.
[42] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *CCS*, 2012.
[43] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *FSE*, 2014.

[44] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *MobiSys*, pages 225–238. ACM, 2017.
[45] L. Matney. Google has 2 billion users on Android, 500M on Google Photos. https://techcrunch.com/2017/05/17/google-has-2-billion-users-on-android-500m-on-google-photos/.
[46] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. Sig-droid: automated system input generation for android applications. In *ISSRE*, 2015.
[47] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. In *Software Engineering Notes*, 2012.
[48] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security*, pages 543–558, 2013.
[49] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA*, 2008.
[50] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for java bytecode analysis. In *ASE'13*.
[51] D. A. Ramos and D. Engler. Under-Constrained Symbolic Execution: correctness checking for real code. In *USENIX Security'15*.
[52] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *ICSE*, pages 300–311, 2017.
[53] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODASPY*, 2013.
[54] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, 2015.
[55] N. Shafiei and F. van Breugel. Automatic handling of native methods in Java PathFinder. In *SPIN Symposium on Model Checking of Software*, 2014.
[56] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.
[57] Stagefright. https://en.wikipedia.org/wiki/Stagefright_(bug).
[58] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *CCS'16*.
[59] H. van der Merwe, O. Tkachuk, S. Nel, B. van der Merwe, and W. Visser. Environment modeling using runtime values for JPF-Android. In *Software Engineering Notes*, 2015.
[60] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. In *ASE*, 2003.
[61] K. Wang, Y. Zhang, and P. Liu. Call me back!: Attacks on system server and system apps in android through synchronous callback. In *CCS*, pages 92–103, 2016.
[62] L.-K. Yan and H. Yin. DroidScope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.
[63] G. Yang, J. Huang, and G. Gu. Automated generation of event-oriented exploits in android hybrid apps. In *NDSS*, 2018.
[64] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, pages 1043–1054. ACM, 2013.
[65] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.
[66] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *S&P*, 2012.
[67] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.
[68] C. Zuo and Z. Lin. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *WWW*, 2017.

**Lannan Luo** is an Assistant Professor at University of South Carolina. She received received her BS degree from Xidian University in 2009, MS degree from the University of Electronic Science and Technology of China in 2012, and PhD degree from the Pennsylvania State University in 2017. Her research interests are software and system security.

**Qiang Zeng** is an Assistant Professor at University of South Carolina. He received his bachelor's and Master's degrees from Beihang University, and his Ph.D. degree from the Pennsylvania State University in 2014. His main research interest is software analysis.

**Chen Cao** is a postdoctoral fellow in the Cyber Security Lab at The Pennsylvania State University. He received his PhD degree from the University of Chinese Academy of Sciences, and BS degree from China University of Mining and Technology. His research interests are in operating system design and implementation, system security.

**Kai Chen** received the BS degree from Nanjing University, in 2004 and the PhD degree from the University of Chinese Academy of Sciences, in 2010. He is a professor in the Institute of Information Engineering, Chinese Academy of Sciences and also in the University of Chinese Academy of Sciences. His research interests include software security and security testing.

**Jian liu** received the PhD degree in computer science from the Institute of Software, Chinese Academy of Sciences, in 2005. He is now an associate research professor at Institute of Information Engineering, Chinese Academy of Sciences. His current research interests include 2191 system and software security, mobile security, web security, program analysis, testing and model checking.

**Limin Liu** is a senior engineer of the Institute of Information Engineering, Chinese Academy of Sciences. She was the local arrangement chair of IEEE CNS 2018 and SecureComm2014. Her research focuses on system security and cloud computing security. Her research has been published on RAID, ICISC, etc.

**Neng Gao** is a professor of the Institute of Information Engineering, Chinese Academy of Sciences. Her research focuses on big data analysis, privacy protection technology, cloud security, and password application technology. Her research has been published on AAAI, ESORICS, PAKDD, etc. She is a member of National Information Security Standardization Committee, and a member of Electronic Authentication Committee of Chinese Association for Cryptology Research.

**Min Yang** is currently a Professor with School of Computer Science, Fudan University, Shanghai, P.R China. His research interests are primarily in mobile security.

**Xinyu Xing** is an Assistant Professor at the Pennsylvania State University, and currently working at JD Inc. as a visiting researcher. His research interest includes exploring, designing and developing tools to automate vulnerability discovery, failure reproduction, vulnerability diagnosis (and triage), exploit and security patch generation. He has received best paper awards from academic conferences such as CCS and ACSAC. His works have been featured by many mainstream media, such as Technology Review, New Scientists and NYTimes etc. He was also the organizer of NSA memory corruption forensics competition.

**Peng Liu** received his BS and MS degrees from the University of Science and Technology of China, and his PhD from George Mason University in 1999. Dr. Liu is the Raymond G. Tronzo, M.D. Professor of Cybersecurity, founding Director of the Center for Cyber-Security, Information Privacy, and Trust, and founding Director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer security. He has published numerous papers on top conferences and journals. His research has been sponsored by NSF, ARO, AFOSR, DARPA, DHS, DOE, AFRL, NSA, TTC, CISCO, and HP. He has served as a program (co-)chair or general (co-)chair for over 10 international conferences (e.g., Asia CCS 2010) and workshops (e.g., MTD 2016). He chaired the Steering Committee of SECURECOMM during 2008-14. He has served on over 100 program committees and reviewed papers for numerous journals. He is an associate editor for IEEE TDSC. He is a recipient of the DOE Early Career Principle Investigator Award. He has co-led the effort to make Penn State a NSA-certified National Center of Excellence in Information Assurance Education and Research. He has advised or co-advised over 35 PhD dissertations to completion.