System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation

Lannan Luo[†][*], Qiang Zeng[‡][*], Chen Cao[§], Kai Chen[§♭], Jian Liu[§♭],
Limin Liu[§♭], Neng Gao[§♭], Min Yang[≀], Xinyu Xing[†], and Peng Liu[†]
[†]The Pennsylvania State University, USA
[‡]Temple University, USA
[§]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
[♭]School of Cyber Security, University of Chinese Academy of Sciences
[≀]Fudan University, China
{lzl144, xxing, pliu}@ist.psu.edu, {qzeng}@temple.edu,
{caochen, chenkai, liujian6, liulimin, gaoneng}@iie.ac.cn, {m_yang}@fudan.edu.cn

ABSTRACT

Android Application Framework is an integral and foundational part of the Android system. Each of the 1.4 billion 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 vulnerable and exploitable. However, most of the existing research has been limited to analyzing Android applications, while there are very few techniques and tools developed for analyzing Android Framework. In particular, to our knowledge, there is no previous work that analyzes the framework through symbolic execution, 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 some unique characteristics of the framework, such as its middleware nature and extraordinary complexity, many new challenges arise and are tackled in CENTAUR. In addition, we demonstrate how the system can be applied to discovering new vulnerability instances, which can be exploited by several recently uncovered attacks against the framework, and to generating PoC exploits.

Keywords

Symbolic execution; concolic execution; vulnerability discovery; exploit generation; Android Framework

[*]These two authors have contributed equally.

MobiSys'17, June 19-23, 2017, Niagara Falls, NY, USA
© 2017 ACM. ISBN 978-1-4503-4928-4/17/06...$15.00
DOI: http://dx.doi.org/10.1145/3081333.3081361

1. INTRODUCTION

The global smartphone market is booming and Android dominates the market with a share of 87.6% [30]. As of 2016, there were 1.4 billion active Android devices [52] and over 65 billion Android apps that had been downloaded and installed [49]. An Android app relies on the Android Application Framework (*Android Framework*, for short) to make it useful. For example, all the user interface designs and multi-tasking features would not work without WMS (Window Manager Service) and AMS (Activity Manager Service) provided by Android Framework; as another example, apps cannot obtain the GPS location without LMS (Location Manager Service) of the framework. Therefore, 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 [26]. Recently, many vulnerabilities in Android Framework were identified [16, 17, 19, 18]. A vulnerability in the framework can lead to large-scale cyber attacks and cause serious harms to user security and privacy. For example, malicious apps can exploit such vulnerabilities to steal user passwords, take pictures in the background, launch UI spoofing attacks, and tamper with user data [45, 47, 48].

Despite the critical role of Android Framework and the security concerns due to the vulnerabilities hidden in its several million lines of code, most of the existing work has been focused on analyzing Android applications [20, 3, 13, 35, 39, 34, 44, 21, 56, 10, 11, 36, 12, 55, 53]. Very few systems and tools are available for analyzing Android Framework [4, 23, 6]; as a result, the insecurity analysis of the framework has been imprecise and/or requires significant manual effort [47, 45]. For example, Shao et al. [47] uncovered a very interesting type of Android Framework vulnerabilities that are due to inconsistent permission checking (detailed in Section 7.2); 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 were provided. 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

Figure 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 }
```

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 automatic vulnerability discovery and exploit generation [5, 9, 25].

This work is to fill the critical gap, aiming to (1) design and build a system that enables symbolic execution of the Android Framework code, and (2) given the description of a new type of attacks exploiting Android Framework, 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 [5, 15].

While many symbolic execution systems have been proposed for analyzing *Windows programs* [24, 25, 7], *Unix programs* [5, 9, 8], and *Android apps* [38, 1, 31, 37, 54], none has explored how to effectively analyze such complex middleware as Android Framework. Due to unique characteristics of Android Framework, many new challenges arise when building its symbolic execution system.

First, most of the existing systems target stand-alone executables and start analysis of the code from the `main` function, while Android Framework is a large piece of middleware with a very complex initialization phase, which parses system and app 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. On the other hand, Android Framework exports system services to apps in the form of a large number 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 analysis.

However, if the analyzer skips the initialization phase and directly analyzes a service interface method, the context information, such as the type and value of variables, is missing and thus many problems may be caused, as is the case in Under-Constrained Symbolic Execution (UCSE) [42, 22, 43].

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`. 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.

Similarly, 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 [32]; this way, however, the loop becomes unbounded and elements of the list become symbolic, which unnecessarily exacerbates the state explosion problem.

Therefore, while it increases the scalability of analysis by symbolically executing each service interface method separately, *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 combines concrete execution and symbolic execution, allowing analysis to start from an arbitrary middleware API method (Section 3).

Second, Android Framework contains a large number of data structures maintained for both system services and apps. From the perspective of finding vulnerabilities exploitable by a malicious app, 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. However, given a malicious app, the variables derived from it *scatter* in the form of object fields and array elements in numerous 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); the object 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. While it is not difficult to identify variables derived from the calling app once one

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

understands the program logic, how to automate the process, given the large number of complex data structures in the framework, is an intriguing new problem as well as a challenge (**C2**).

Our hypothesis is that, as the framework stores information for multiple clients (i.e., apps), there must exist fixed *patterns* used to access the client-specific information when the framework services a client call, and this hypothesis is validated through our 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).

Third, a straightforward design is to place the symbolic execution engine inside the Android system, such that the analyzer can make use of the host execution environment including the native libraries and other supporting processes (e.g., the Service Manager process used to register and query system services). But this way the symbolic executor is tightly coupled with Android, and the implementation has to handle compatibility with the Android Runtime (ART) in terms of thread management, instruction execution, binary representation of objects, and garbage collection. This significantly complicates the implementation of the symbolic executor and makes the system brittle and hard to debug due to various incompatibility issues. Moreover, since it is unlikely to modularize the code for symbolic execution based on this design, whenever a new version of the Android system is released the symbolic execution code has to be re-inserted. To avoid the complicated and brittle implementation and the 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 complicated and coupled implementation is a challenge (**C3**).*

We propose a decoupled design which builds the symbolic executor outside the Android system but is still able to make use of the Android execution environment. An innovative and critical component of the system is to migrate information generated in Android, such as the classes and objects, to the symbolic execution environment (Section 5).

We have overcome the challenges above and implemented the symbolic execution system named Centaur for symbolic execution of the Java code in Android Framework. The source code for Centaur is publicly available.[2] Centaur is a path exploration system that can effectively assist automatic and precise vulnerability discovery. We concretely demonstrate how Centaur can be applied to vulnerability discovery by considering several recently uncovered attacks that exploit the framework vulnerabilities. We show that the process minimizes the manual effort and guarantees zero-false positives, in contrast with recent researches on finding Android Framework vulnerabilities that rely on laborious and error-prone manual work [47, 45]. Finally, we make use of Centaur to generate PoC exploits to validate the findings. We make the following contributions.

- To our knowledge, Centaur is the first system that supports symbolic execution of Android Framework. It provides an approach to exploring paths in Android Framework automatically and precisely. The proposed techniques can potentially be applied to symbolic execution of other complex middleware.

- Unlike previous symbolic execution schemes that either start analysis from the main function or analyze a non-main function without the context information, we employ a scheme that allows service interface methods of middleware to be analyzed separately for much improved scalability and meanwhile provides a complete execution context.

- 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. A powerful algorithm that migrates the execution context information from Android to the symbolic executor is designed.

- We have implemented Centaur and evaluated it in terms of the effectiveness and precision in finding vulnerabilities and generating PoC exploits.

The remainder of the paper is organized as follows. Section 2 covers some background about Android Framework and symbolic execution. Section 3 gives an overview of the system. Section 4 describes how to identify variables as symbolic inputs. Section 5 presents how to migrate the execution context. Section 6 describes the implementation details, and Section 7 describes the evaluation. The related work is discussed in Section 8, and the paper is concluded in Section 9.

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, except for the media services, run as threads in the System Server process [26]. Thus, the System Server process plays a central role in Android Framework. This work uses the services in this process as examples to illustrate the ideas and techniques, which should be applicable to other services.

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 IPC mechanism Binder, which dispatches the call to one of the threads of the target system service. Android Framework is mainly implemented in Java. E.g., 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 Java code.

Symbolic execution. Symbolic execution provides a means of efficiently exploring execution paths [33]. For example, consider the function `getAllowedResolutionLevel` in Figure 1, by assigning a symbolic value (as opposed to assigning concrete values as in fuzzing) to `ps`, symbolic execution analysis can iterate every of the three paths and precisely provide the condition that the symbolic value should satisfy

[2] https://github.com/Android-Framewrok-Symbolic-Executor/Centaur

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 for several reasons. 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. SYSTEM OVERVIEW

Our observation of Android Framework is that its execution consists of the initialization phase and the ready-for-use phase, and 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.

When starting the symbolic execution from an entrypoint method, if only the parameters of the entrypoint method are set as symbolic inputs [40], the path exploration will be severely limited, leading to *over-constrained* symbolic execution. In the framework, variables derived from the malicious app (mainly its manifest file) are also under control of attackers and can affect the execution of system service calls; thus, those variables should also 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* to identify those variables as symbolic inputs on the fly during path exploration by capturing the characteristic access patterns (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 is suitable for PC2SE, as shown in Figure 2, where the symbolic execution engine is built outside of Android. 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 code for the symbolic executor is separated from Android code, it does not need to be maintained when the Android system is updated.

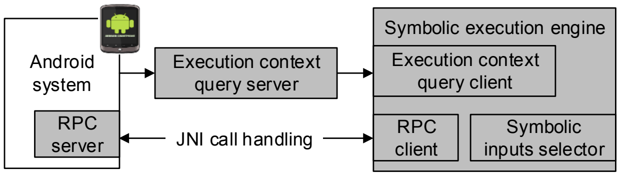
Android system
RPC server
Execution context query server
JNI call handling
Symbolic execution engine
Execution context query client
RPC client
Symbolic inputs selector

Figure 2: Architecture of Centaur.

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 to mitigate the information properly will be discussed in Section 5. Finally, an RPC server placed inside the Android system, and JNI calls during path exploration are delegated to the RPC server, which will be discussed in Section 6.

4. IDENTIFYING SYMBOLIC INPUTS

4.1 App-specific Variables

To deal with over-constrained symbolic execution, it is vital to set Android Framework variables that are derived from the malicious app as symbolic inputs, such that execution paths due to all possible values of these variables are explored by symbolic execution.

A closer look at 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. For example, the aforementioned `LocationManagerService.mProviders` ArrayList (Line 6 in Figure 1) exists no matter what apps are running. This type of variables should not be specified as symbolic inputs, since they are not derived from the malicious app.

The second type, called *app-specific* variables, stores app-specific information. Some variables store information for all installed apps; for instance, `Settings.mUserIds` is an ArrayList that stores the installation data of each installed app (the code path, signature, granted permissions, etc.). Others store the information of running apps, such as task affinities, intents, and back stacks. 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 the many data structures for the system services: Android Framework code is structured as a set of system service classes, each of which points to some objects and arrays for storing app-specific information. Therefore, given an app, the framework variables derived from it scatter and exist as objects fields and array elements among the many data structures.

Note that the task of selecting variables as symbolic inputs is not only to find the app-specific variables but also to

Figure 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);
   ...
}
```

locate fields or elements within the variables that are derived from a given malicious app. For instance, in addition to determining `Settings.mUserIds` is an app-specific variable, we need to locate which element in the array is derived from the malicious app. Thus, the task is like looking for a needle in a large pile of hay considering the large number of complex data structures.

4.2 Access Patterns

In order to determine which variables are derived from a given app, a natural method is to track how the information flows from the app to Android Framework via tainting. 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 using taint analysis. Note that existing taint analysis, such as TaintDroid [20], Chex [35], and TaintART [50], is able to track whether the information of the return values (e.g., GPS locations) of specific system service calls flow to specific sinks (e.g., sending to network), but none is able to track how the whole app-level information propagates to Android Framework variables. Moreover, conventional tainting techniques suffer from the well-know overtainting and undertainting issues, which lead to imprecise tainting results.

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 retrieved and marking them as symbolic inputs in the process of path exploration. Our hypothesis is that, as the framework stores information for multiple apps, there must exist specific ways to retrieve the information for a given app (i.e., the calling app), rather than any other app, when servicing a system service call. Our further investigation has validated the hypothesis and revealed 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 further shows that there are two such formulas 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. 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 an user app can have, while $100,000$ means `PER_USER_RANGE` (Line 27 in Figure 1), indicating the largest UID plus one. 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 32).

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)	$\tau(v_C) \leftarrow \tau(v_A)$

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.

While there are a large variety of data structures in the framework, our investigation shows that they commonly follow the two fixed access patterns to retrieve the app-specific information when servicing a system service call.

4.3 Slim Tainting

We thus propose *slim taint analysis* that tracks and recognizes the characteristic access patterns above on the fly during path exploration and sets variables as symbolic inputs when app-specific information is accessed.

Next, we elaborate slim tainting. Similar to other tainting techniques, it consists of taint sources, taint propagation logic, and taint sinks. (1) The return values of `getCallingUID` and `getPackageName` are set as taint sources; they are unique identifications of an app. (2) The taint propagation logic as shown in Table 1 is very simple, involving only two instructions and one string concatenation function. They are all derived from the access patterns described in Section 4.2. (3) Finally, 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 Figure 1 as an example to illustrate how the slim taint analysis works. First, due to the call to `getCallingUID` (Line 3), its return value `uid` obtains the taint. Second, the taint propagates along Lines 31 and 32 according to the taint propagation logic. Finally, at Line 33, the `get` function works as a sink to set the element (and *only* this element) accessed through 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 6 includes its implementation details. It is worth mentioning that the implementation intercepts some specific function calls and changes the interpretation of several bytecode instructions; it does not need to change a single line of the source code of

Android Framework and does not need any code annotation. Therefore, it is precise and automatic. When the Android system evolves, new access patterns may be used. In that case, we need to update some details of slim tainting in terms of, e.g., taint sources and propagation logic. But the idea of capturing access patterns should still be applicable.

5. EXECUTION CONTEXT MIGRATION

The decoupled architectural design requires that the execution context due to the concrete execution be mitigated from the Android system to the analyzer. Note that, in the execution context, the program counter, the register file, and the stack all obtain their 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? (2) how to conduct the migration during symbolic execution? and (3) how to bootstrap the migration?

5.1 Snapshot Parsing and Information Query

A heap snapshot is nothing but an array of bits. However, it would not work if we simply copy the array of bits to the JVM instance for symbolic execution, 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 implementation, each object in our symbolic executor needs extra space for recording the taint and the symbolic expression; plus, its heap memory management is different from the one used in Android. On the other hand, given an object, both ART and our JVM should agree on the number of the contained fields, according to the class definition file, and their values. Therefore, given an object, our 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 exectuor 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.

5.2 Migration Algorithm

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 the object. But what about fields of the reference type? A deep copy is too 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 (where the snapshot has been captured). We choose to enforce *a variant of the simple shallow copy*: when an object is migrated, we simply copy all the field values, but for each reference-typed field, we mark that it indicates 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 exectuion world.

Algorithm 1 Migration of heap information.

```
function GETFIELD(index)
    objRef = peekStackTop()
    fdInfo = getFdInfo(index)          ▷ Class-specific info.
    fd = getFd(objRef, fdInfo)         ▷ objRef-specific info.
    if !fd.getSnapshotRefAttribute() then
        return super.getfield(index)
    end if
    concRef = fd.getValue()
    symRef = conc2Sym.get(concRef)
    if symRef == NULL then
        fdType = fdInfo.getFdType()
        if fdType == strRef then
            str = snapshot.getStr(concRef)
            symRef = searchConstantPool(str);
            if symRef == NULL then
                symRef = newString(str);
            end if
        else if fdType == arrayRef then
            entryType = fdType.getEntryType()
            len = snapshot.getArrayLen(concRef)
            symRef = newArray(entryType, len)
            snapshot.copyEntries(symRef, concRef)
        else                           ▷ Other reference types
            symRef = newObj(fdType)
            snapshot.copyFields(symRef, concRef)
        end if
        conc2Sym.addPair(concRef, symRef)
    end if
    fd.setValue(symRef)
    fd.setSnapshotRefAttribute(false)
    return super.getfield(index)
end function

function INITCLASS(classInfo)
    if snapshot.isInitialized(classInfo) then
        snapshot.copyStaticFields(classInfo)
    else
        super.initClass(classInfo)
        handleBootstrapField(classInfo)
    end if
end function
```

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 the hash table, 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

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

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.

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.

The heap execution context migration algorithm, which is built into the symbolic execution engine, is implemented by overriding a couple of bytecode instructions, shown in Table 1, which includes, for each instruction, the effect that the instruction has on the operand stack and the instruction description. Algorithm 1 shows the main migration procedures.

5.2.1 Migrating Objects

The instruction `getfield` is used to access non-static fields in an object. Given a reference to an object (this object must have been migrated; Section 5.3 covers the reason) on the stack (Line 2), the instruction `getfield` pushes a field value of the object onto the stack. Below we describe how an object pointed to by a field when this field is accessed through `getfield`.

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 $\langle \mathit{concRef}, \mathit{symRef} \rangle$ 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 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., $A[5][10]$, 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.

Figure 4: Example of a test driver.

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

5.2.2 Migrating Classes

When an operation (e.g., an object of a class is created or a class's static fields are accessed 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.

5.3 Bootstrapping

An important invariant kept during migration is that, when a field of an object o (resp. an element of an array A) is accessed, o (resp. A) must have been migrated in 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 reside?*" The answer is that f, called the *bootstrap* field, resides in the test driver class, and it is a reference to the system service class that contains the entrypoint method. Listing 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 the `TestDriver` class is initialized, it sets the bootstrap field value to the reference value of the system service object 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).

Example. In Listing 4, 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. The migration of classes and objects form a *migration tree*, which grows as classes and objects get migrated, rooted at the class

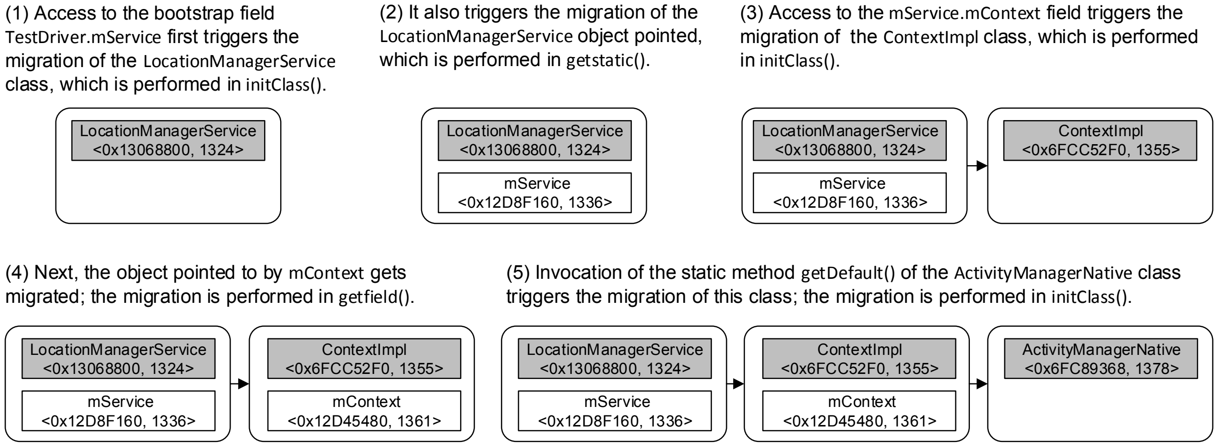
(1) Access to the bootstrap field TestDriver.mService first triggers the migration of the LocationManagerService class, which is performed in initClass().
LocationManagerService <0x13068800, 1324>
(2) It also triggers the migration of the LocationManagerService object pointed, which is performed in getstatic().
LocationManagerService <0x13068800, 1324>
mService <0x12D8F160, 1336>
(3) Access to the mService.mContext field triggers the migration of the ContextImpl class, which is performed in initClass().
LocationManagerService <0x13068800, 1324>
mService <0x12D8F160, 1336>
ContextImpl <0x6FCC52F0, 1355>
(4) Next, the object pointed to by mContext gets migrated; the migration is performed in getfield().
LocationManagerService <0x13068800, 1324>
mService <0x12D8F160, 1336>
ContextImpl <0x6FCC52F0, 1355>
mContext <0x12D45480, 1361>
(5) Invocation of the static method getDefault() of the ActivityManagerNative class triggers the migration of this class; the migration is performed in initClass().
LocationManagerService <0x13068800, 1324>
mService <0x12D8F160, 1336>
ContextImpl <0x6FCC52F0, 1355>
mContext <0x12D45480, 1361>
ActivityManagerNative <0x6FC89368, 1378>

Figure 5: An example of migrating the heap. Grey rectangles and white ones 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. The migration of a class also triggers the migration of all its super classes, which are not shown for simplicity.

and object corresponding to the bootstrap field type. We use the code in Listing 4 as an example to partially illustrate how a migration tree grows, as shown in Figure 5, where the root node is the class and object for `LocationManagerService`. Note that the step (5), `getDefault` is invoked due to the call to `getProviders`.

6. IMPLEMENTATION DETAILS

Background on SPF. We built the symbolic executor based on Symbolic PathFinder (SPF) [41], a symbolic execution framework on top of Java PathFinder (JPF) [51]. SPF can be understood as a non-standard Java bytecode interpreter, which enforces path exploration when interpreting the code; e.g., when interpreting an `if` statement, it creates two program states so that both branches will be explored. It provides a set of path selection policies that can be chosen from and various constraint solvers for resolving path conditions.

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 6,285 lines of code for implementing Centaur through extending SPF. Significant effort has been saved by building upon SPF, which is made possible thanks to the decoupled design.

Classpath. The Java source code in Android is compiled into `.jar` files, which comprise standard `.class` files, and the symbolic executor is built to analyze Java bytecode in such `.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 spercify the Android Framework code, and the last line the core libraries of ART, such as utility, io, and math libraries. Several classes (e.g., `java.lang.class`, `.Thread`, `.StackTraceElement`) are modeled by the symbolic executor, but `core-libart` contains the Android version of these classes; hences, these specific classes have to be excluded from `core-libart` to avoid failures due to duplication.

Slim tainting. Slim tainting 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. Centaur adds one attribute indicating the taint and another indicating the symbolic input property for each field, array element, and call-stack variable; Thanks to JPF's supports for interception of function calls and adding field attributes without modifying the framework source code, there is no maintenance effort needed when there are new versions of Android released, as long as the app-specific variable access patterns (Section 4.2) are not changed.

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. The standard `.hprof` file format opens up the possibility of parsing the snapshot using many existing tools. In our case, the file is then parsed to extract the list of classes and objects stored in the `.hprof` file using a hprof file parser [29]. The extracted information is then organized into the memory space of the execution context query server.

Dealing with Messaging. Two messaging mechanisms that are frequently used by system services are Message Handler and State Machine calls. 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 [28]. To deal with messages sent to a queue, we propose to replace the call to `sendMessage(message)` with a call to the destination handler's `handleMessage(message)`. We connect the senders and receivers for messages sent through State Machine in a similar way.

Handling JNI calls. Part of Android Framework is implemented in native code, which is invoked 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 (`getPackage-Name()`) of the client app are modeled to return the corresponding information for the malicious app. (2) The return values of other native methods that return app-specific information of the malicious app are specified as symbolic inputs. For example, many native methods declared in the package `android.content.res` access application resources. (3) For native methods that do not have return values are ignored. (4) 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 [46]. While jpf-nhandler delegates native calls to a host JVM, our client delegates them to an app running as the RPC server in a remote Android system (Figure 2) and handels native calls using reflection on demand. The *GSON* library [27] is used for marshalling and unmarshalling method parameters and return values, which are transmitted between the RPC server and client via socket.

7. EVALUATION

We compare CENTAUR against under-constrained symbolic execution (UCSE) in Section 7.1. Both can start symbolic execution from system interface methods to reach the code deep in the program, but CENTAUR makes use of the execution context provided by concrete execution. Ideally, we should also compare the PC2SE scheme used in CENTAUR against symbolic execution that starts from the main entry of Android Framework, that is, `SystemServer.main`, but note that our symbolic executor runs outside Android, while at the initialization phase the System Server heavily relies on the Android environment, such as the file systems and other supporting processes, to finish its initialization, meaning that it is unlikely to initialize the System Server process outside the Android environment. We thus compare PC2SE with UCSE only.

CENTAUR provides strong support for vulnerability discovery and exploit generation. To demonstrate how CENTAUR can be applied to assisting vulnerability discovery, we investigate two distinct types of recently uncovered attacks that exploit Android Framework vulnerabilities. The investigation in Sections 7.2 and 7.3 shows how zero-day vulnerability instances can be discovered through the application of CENTAUR.

Finally, the reliability of the approach is investigated. We present exploit generation experiments based on snapshots captured at different times, and analyze the consistency of the results in Section 7.4.

The experiments were performed on a machine with an Intel Core i7 4.0Ghz Quad Core processor and 32GB RAM running Linux kernel 3.13. Exploits were generated on Android Framework 5.0 and verified using different versions of Android systems.

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 any essential functionality; in particular, the skeleton app used in our experiments borrows the manifest file from the Android developer website, which has "every element that it can contain" [2]. In practice, the analyst can choose any app as the malicious app.

7.1 Comparison with Under-constrained Symbolic Execution (UCSE)

The first issue of applying UCSE to symbolic execution of 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 relying on 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. UCSE handles such symbolic inputs using lazy initialization, which causes the following problems: (1) loops that iterate through collection data structures are unbounded, and (2) the generated concrete values may be unrealistic.

We tried to perform UCSE of Android Framework using Java PathFinder, which kept crashing when it was applied directly. We spent a lot of time and tedious effort modifying the framework code (e.g., adding the type information about objects pointed to by references to assist dynamic dispatching) to make the symbolic execution possible. We thus only modified the code with respect to the `getProviders` API and the `startActivity`. UCSE spent 138m when analyzing `getProviders` and ran out of memory in the case of `startActivity`, while CENTAUR finished them within 26s and 42m 37s, respectively.

Therefore, path exploration without precise information of the execution context causes many problems, such as requiring tedious manual annotation effort and exploring spurious paths. CENTAUR resolves the problems by migrating the execution context from the concrete execution world to symbolic execution.

7.2 Investigating Inconsistent Security Policy Enforcement (ISPE)

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 [47]. The system, called *Kratos*, first builds a call graph based on the Android Framework code. With the call graph, it finds all the execution paths that can reach sensitive operations. Kratos then compares the paths pairwise to identify paths that reach the same sensitive operation with inconsistent security checks enforced, and reports them as suspected ISPE vulnerabilities, in that they violate the security property that *all paths should have consistent permissions for reaching a given sensitive operation.*

Combined approach for bug finding. Static analysis based on the reachability analysis for finding ISPE bugs may report false positives, as some paths are infeasible in real executions. Currently, manual effort is used to scrutinize the code along each reported path, 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 suspected 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 suspected vulnerabilities. It thus demonstrates the uses of Centaur comprehensively. All the steps have been performed automatically, in contrast with previous work that relies on tedious and error-prone manual inspection. Plus, zero false positives are guaranteed as all suspected vulnerabilities are validated by the runtime log.

Result summary. Table 3 summarizes the experiment results (the vulnerability shown in the last row is discussed in Section 7.3). 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 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 greater than that of objects. In the majority of the 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 iterated, 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.

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 (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.

A detailed example. As an example, we describe in detail how the combined approach was applied to finding the first vulnerability in Table 3. (1) First, the static analysis based on path reachability and pairwise path comparison finds that both `getProviders(Criteria, boolean)` 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 suspected vulnerability. (2) Next, Centaur is applied to check automatically whether there exist paths that can reach the sensitive operation from the two service interface methods. Specifically, after the Android system is initialized and the skeleton app is launched, a heap memory snapshot of the System Server process is captured, and provides execution context for symbolic execution, and symbolic execution starts from the two service interface methods respectively.

Figure 6: `startActivityUncheckedLocked()`.

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

Compared to previous work that relies on enormous and error-prone manual inspection, the combined approach of 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.

7.3 Investigating Task Hijacking Attacks

Background. The Activity Manager Service 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 [45]. For example, 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 (a command parameter, a network packet, etc.) but a separate app.

Vulnerability discovery. We use the `EditEventActivity` activity of the `calendar` app as an example victim activity. In the skeleton app, the main activity of the skeleton app starts the malicious activity, denoted by 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 snapshot when the victim app and the skeleton app are started and the main activity of the skeleton app is to start the malicious activity.

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()
		LMS.getProviders(Criteria,boolean)
			55
			77
				55
				93
					4
					14
						4
						42
							—
							66
								—
								66
									19s
									26s
										92.3
										45.8

2	Read phone state with insuf. privilege	TSI.getCallState()
		TSI.isInCall()
		TSI.isRinging()
			48
			62
			60
				48
				69
				65
					3
					17
					16
						3
						20
						18
							—
							1
							1
								—
								1
								1
									14s
									32s
									35s
										72.4
										83.5

3	End phone calls with insuf. privilege	TSI.endCall()
		PIM.endCall()
			81
			80
				83
				85
					21
					23
						24
						26
							1
							1
								1
								1
									29s
									38s
										91.3
										89.4

4	Close system dialogs with insuf. privilege	WMS.closeSystemDialogs(String)
		AMS.closeSystemDialogs(String)
			57
			63
				57
				67
					6
					11
						6
						15
							—
							2
								—
								2
									17s
									29s
										69.1
										56.0

5	Set up HTTP proxy working in PAC mode with insuf. privilege	WSI.addOrUpdateNetwork()
		WSI.getWifiServiceMessenger()
			67
			65
				122
				84
					23
					21
						52
						24
							16
							1
								16
								1
									26s
									18s
										30.4
										57.3

6	Enable/Disable mDNS daemon with insuf. privilege	NS.setEnabled(boolean)
		NS.getMessenger()
			75
			80
				114
				81
					28
					11
						53
						14
							1
							1
								1
								1
									44s
									18s
										47.2
										62.4

7	Task hijacking	ASS.startActivityUncheckedLocked()	324	387	136	182	2,020	810	42m 37s	58.3

Figure 7: An example set of concrete input values.

```
(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")
```

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 as shown in Figure 6. 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.

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 and activity names to be equal to those of the victim activity. Simple scripts were written to filter out illegal concrete values (1,210 sets totally). Figure 7 shows an example of the rest 810 sets 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; for example, `r.launchMode` is mapped to the `android:launchMode` in the manifest file. Figure 8 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.

Figure 8: 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.

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

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 API `startActivity(Intent, Bundle)` is not included in version 4.0, and thus only exploits with `options == null` can be used for invoking `startActivity(Intent)`.

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* be 0) that was not reported in previous work [45]. 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.

7.4 Consistency of Exploits Generated 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 Android 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 one or more apps; hence, during symbolic execution, reasonable setting up is established consistently; for example, the victim activity should already be started in the task hijacking case prior to capturing snapshots. The results show that an attack succeeds as long as the same statuses recur.

Finally, the values of non-app-specific variables do not affect path exploration, at least, in our cases. For example, in the case of inconsistent security policy enforcement vulnerabilities due to accessing the names of installed providers, the path exploration 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.

8. RELATED WORK

DART is the first concolic testing tool that uses symbolic analysis in concert with concrete execution to improve code coverage [24]. 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 [24], EXE [9], KLEE [8], and S^{2}E [14] all make use of concrete execution to execute external code or uninterpreted functions, similar to our handling of JNI 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 set up the execution context for the symbolic execution phase.

Symbolic PathFinder (SPF) can switch to symbolic execution at any point of concrete execution [40]. SPF aims at generating unit test cases, so it simply specifies function parameters as symbolic inputs, which leads to over-constrained symbolic execution. We make use of concrete execution in the initialization phase, which makes the analysis results *sensible* and *easy to interpret*. Plus, PC2SE aims at generating exploits that can be embedded into malicious apps, so it finds variables derived from the malicious app, in addition to the function parameters, and uses them as symbolic inputs. Finally, we have proposed a novel architectural design that decouples symbolic executor from the concrete execution environment to obtain a modularized implementation.

Symbolic execution has been applied to vulnerability discovery and exploit generation in previous work. For example, SAGE has successfully found many vulnerabilities of Windows programs [25], leveraging the technique described in DART [24]. SAGE has demonstrated symbolic execution can be very useful for finding vulnerabilities. AEG has shown vulnerability discovery and exploit generation given vulnerable Unix programs [5]. Automatic patch-based exploit generation (APEG) generates exploits based on information in patches [7]. SAGE, APEG, and AEG all consider stand-alone native Windows or Unix executables, while our work analyzes Android Framework, a piece of middleware that manages all executables running upon it. Many unique challenges arise and are addressed in this work.

There has been a lot of work that applies symbolic execution to Android apps for test input generation or security enhancement [38, 1, 31, 37, 54]. For example, Jensen et al. proposed to use concolic execution to build summaries of individual event handlers and then generate event sequences backward, in order to find event sequences that reach a given target line of code in the Android app [31]. To our knowledge, our system is the first that supports symbolic execution of Android Framework.

9. CONCLUSIONS

We have introduced the first system, called Centaur, for symbolic execution of Android Framework. To avoid state space explosion due to the complex initialization, the Phased Concrete-to-Symbolic Execution is proposed that runs concrete execution for the initialization phase, providing execution context to symbolic execution. Among the large number of variables in the execution context, slim tainting tracks characteristic access patterns to identify variables derived from the malicious apps as symbolic inputs. In order to decouple the implementation of Centaur from Android, the execution context provided by concrete execution is migrated from an Android ART process to a Java VM. 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, we plan to apply Centaur to other purposes in future work, such as automatic API specification generation, fine-grained malware analysis, and testing.

10. ACKNOWLEDGMENTS

We thank anonymous reviewers and our shepherd Dr. Luca Mottola for their constructive comments. Dr. Peng Liu and Lannan Luo were 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 supported by NSFC U1536106, 61100226, Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701). Dr. Min Yang was funded by the National Program on Key Basic Research (NO.2015CB358800) and NSFC U1636204.

11. REFERENCES

[1] S. Anand, M. Naik, H. Yang, and M. J. Harrold. Automated concolic testing of smartphone apps. In *FSE*, 2012.

[2] App Manifest. https://developer.android.com/guide/topics/manifest/manifest-intro.html.

[3] 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.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the android permission specification. In *CCS*, 2012.

[5] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *Communications of the ACM*, 2014.

[6] 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.

[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] 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.

[11] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *ICSE*, 2014.

[12] 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.

[13] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252, 2011.

[14] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[15] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, 2007.

[16] CVE-2015-6628. https://www.cvedetails.com/cve/CVE-2015-6628/.

[17] CVE-2016-2496. https://www.cvedetails.com/cve/CVE-2016-2496/.

[18] CVE-2016-3750. https://www.cvedetails.com/cve/CVE-2016-3750/.

[19] CVE-2016-3759. https://www.cvedetails.com/cve/CVE-2016-3759/.

[20] 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.

[21] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[22] D. Engler and D. Dunbar. Under-constrained execution: marking automatic code destruction easy and scalable. In *ISSTA*, 2007.

[23] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[25] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[26] Google. Android Interfaces and Architecture. https://source.android.com/devices/.

[27] GSON. https://sites.google.com/site/gson/Home.

[28] Handler. https://developer.android.com/reference/android/os/Handler.html.

[29] HPROF Parser. https://github.com/eaftan/hprof-parser.

[30] IDC. Smartphone OS Market Share, 2016. https://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[31] C. S. Jensen, M. R. Prasad, and A. Moller. Automated testing with targeted event sequence generation. In *ISSTA*, 2013.

[32] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.

[33] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] 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 *Proceedings of the 37th International Conference on Software Engineering*, pages 280–291, 2015.

[35] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012.

[36] 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.

[37] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. SIG-Droid: automated system input generation for android applications. In *ISSRE*, 2015.

[38] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. In *Software Engineering Notes*, 2012.

[39] 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 *Presented as part*

of the 22nd USENIX Security Symposium (USENIX Security 13), pages 543–558, 2013.

[40] 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.

[41] 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*, 2013.

[42] D. A. Ramos and D. Engler. Under-Constrained Symbolic Execution: correctness checking for real code. In *USENIX Security*, 2015.

[43] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *CAV*, 2011.

[44] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220, 2013.

[45] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, 2015.

[46] N. Shafiei and F. van Breugel. Automatic handling of native methods in Java PathFinder. In *SPIN Symposium on Model Checking of Software*, 2014.

[47] 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.

[48] Stagefright. https://en.wikipedia.org/wiki/Stagefright_(bug).

[49] Statista. Cumulative number of apps downloaded from the Google Play, 2016. https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.

[50] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. ACM, 2016.

[51] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. In *ASE*, 2003.

[52] WSJ. Google says android has 1.4 billion active users. www.wsj.com/articles/google-says-android-has-1-4-billion-active-users-1443546856.

[53] L.-K. Yan and H. Yin. DroidScope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.

[54] 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 *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[55] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *S&P*, 2012.

[56] 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.