Heap Memory Snapshot Assisted Program Analysis for Android Permission Specification

Lannan Luo
University of South Carolina, USA

***Abstract*—Given a permission-based framework, its *permission specification*, which is a mapping between API methods of the framework and the permissions they require, is important for software developers and analysts. In the case of Android Framework, which contains millions of lines of code, static analysis is promising for analyzing such a large codebase to derive its permission specification. One of the common building blocks for static analysis is the generation of a global call graph. However, as common for object-oriented languages, the target of a virtual function call depends on the runtime type of the receiving object, which is undecidable *statically*. Existing work applies traditional analysis approaches, such as class-hierarchy analysis and points-to analysis, to building an over-approximated call graph of the framework, causing much imprecision to downstream analysis. We propose the *heap memory snapshot assisted program analysis* that leverages the dynamic information stored in the heap of Android Framework execution to assist in generating a more precise call graph; then, further analysis is performed on the call graph to extract the permission specification. We have developed a prototype and evaluated it on different versions of Android Framework. The evaluation shows that our method significantly improves on prior work, producing more precise results.**

I. Introduction

The global mobile market is booming, amounting to more than $3.1 trillion of economic value [29]. Capturing 87.5% of the global smartphone shipments, Android dominates the market with 1.4 billion Android devices and over 65 billion apps downloaded [45]. As users frequently interact with smartphones, plenty of sensitive information is stored in smartphones. Thus, a permission system is needed to control access of applications to sensitive information as well as resources, such as the user's contact list and the phone's microphone [2], [36], [48].

In Android, the permission system is implemented in the *Android Framework*, which exposes application programming interfaces (APIs) to third party applications for interacting with the underlying system [36], [48], [37], [11], [5], [30]. For instance, if an app wants to obtain the GPS location, it needs to invoke the `getGPSLocation` method of the Location Manager Service provided by the framework. The framework will then check whether the calling app has been granted the required permission before retrieving the sensitive data.

As the permission system plays an indispensable role to protect sensitive information and resources, it is important to have a good understanding of the *permission specification*, which is a mapping between each of the API methods and its required permission(s). In the case of Android, the mapping is given by the official documentation, which, however, is not always up-to-date or clear [3], [35]. As a result, developers are often confused with the use of permissions and, hence, either under- or over-estimate the required permissions. Missing a permission causes application failures, while adding too many permissions is not secure, as they may be exploited by injected malicious code and malicious developers [22], [3].

Researchers have attempted to extract the permission specification using different approaches, which can be divided into the following two categories: dynamic analysis and static analysis. Stowaway [22] uses unit testing and feedback directed API fuzzing to run apps and extract the permission specification; however, even with significant time and effort invested in testing, it is still difficult to examine the entire Android Framework and achieve a high code coverage in order to extract a complete mapping.

Some approaches resort to static analysis, which is more promising for analyzing a large codebase. One of the building blocks in the process is the computation of a global (inter-procedural) call graph of Android Framework [3], [4], [41], [13]. However, as common for object-oriented languages (Android Framework is mainly implemented in Java), the target of a method call depends on the runtime type of the receiving object, which is undecidable, in general, *statically*. For example, `PScout` [3] builds a call graph that considers each virtual function call via a reference to target all the subclasses of the reference type. `AXPLORER` [4] uses points-to analysis to infer the possible dynamic type of the receiving object of a virtual function call. Both the class-hierarchy analysis and points-to analysis based approaches cause much imprecision, since they cannot infer runtime type information accurately.

Therefore, *how to generate a precise call graph of Android Framework is the main challenge for extracting the permission specification via static analysis.* Our observation about resolving this challenge is that the execution of Android Framework consists of the *initialization* phase and the *ready-for-use* phase. Once the initialization of the system services is done, much object-type information is determined and plenty of useful dynamic information is stored in the heap of the framework process. We thus propose the *heap memory snapshot assisted program analysis* that leverages the dynamic information stored in the heap to generate a more precise call graph. Very interestingly, as all the system services in Android Framework adopt the singleton design pattern [34], given the system service, there is no ambiguity to locate the object for the system service from the heap and, hence, there is no ambiguity to obtain the dynamic types of all the objects pointed to by the reference fields of the system service object.

During the call graph generation, for each virtual function invocation, whenever possible we extract the runtime type of the receiving object from the heap memory snapshot, and precisely connect the invocation to its target. This way, we minimize the imprecision introduced by static analysis based inference, which largely lowers the number of *false positives* (that is, permissions not needed by an API method are falsely considered as needed). After the call graph is built, further analysis is performed on it to extract the permission specification.

The heap may vary with time (e.g., the runtime object types of some references may change as the framework runs). However, our another observation is that, once the framework enters the ready-for-use phase, any API method could be issued by an application. Given an API method, the needed permission(s) should *not* change over time, as the permission specification should keep consistent, no matter whether the method is invoked right after the initialization or after a long time. Our experiments based on snapshots captured at different times confirm this point (Section VI-D).

We develop a prototype, called HEAPHELPER. Our experiment results show that our method significantly improves on the prior work that entirely relies on static analysis and produces much more precise results. We further conduct experiments based on heap memory snapshots captured at different times and from different devices, and analyze the consistency of the permission specification in terms of these varying heap memory snapshots. In addition, the effectiveness of HEAPHELPER is verified on different versions of Android Framework.

We made the following contributions.

- We propose the *heap memory snapshot assisted program analysis* that leverages the dynamic information stored in the heap to assist in generating a more precise call graph of Android Framework. To the best of our knowledge, this is the first work that approaches the call graph generation problem by leveraging the heap information. It is a novel and effective combination of dynamic information and static analysis.
- We build HEAPHELPER and evaluate its effectiveness in extracting the Android permission specification. Heap memory snapshots captured at different times are used for analyzing the permission specification. Some interesting findings and internals of the framework are uncovered.
- The more precise call graph of Android Framework may benefit other downstream analysis work that relies on the call graph, such as static tainting analysis [1] and inter-component communication [13]. Plus, the proposed technique can potentially be applied to other complex middleware, such as the Core Services layer in iOS, to generate a more precise call graph for static analysis.

II. BACKGROUND

A. Android Boot Process

The Android boot process involves the initialization of Android Framework. The first program to run is the `bootloader` which initializes the environment and high-level kernel subsystems. Then the root file system is mounted and the `init` process is started. The `init` process mounts file systems and starts daemons such as the *zygote*. The *zygote* is a core process from which new Android processes are forked. The initialization of *zygote* starts the `system_server` process which in turn initializes most system services (e.g., the Activity Manager Service and Package Manager Service) and managers (e.g., the Activity Manager and Package Manager). In short, during Android's boot process, the system services and managers are instantiated and initialized.

B. Android Framework

Android Framework provides a collection of system services, which implements many functionalities, such as managing the life cycle of all apps, organizing activities into tasks, and managing app packages, etc. Most system services are implemented as bound services [18] and run as threads in the `system_server` process [25], while some run as threads in other processes, e.g., `com.android.inputmethod.latin`, `com.android.phone`, and `com.android.keychain`.

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 via 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. E.g., Android Framework 5.0 contains 2.4 million lines of Java code and 880 thousand lines of C/C++ code. HEAPHELPER is designed to analyze the Java code.

C. Android Permissions

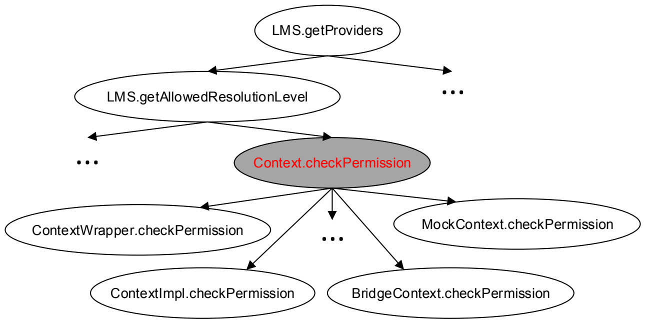
Android Framework utilizes a permission based security model, which provides controlled access to system resources. E.g., an app must hold a particular permission in order to access the security critical methods of the framework. Each permission is presented as a string. An app declares the required permissions in the manifest file (i.e., `AndroidManifest.xml`). During installation, the system parses the manifest file and stores the permissions along with the app's unique UID. When the app wants to access a privileged system resource or a security critical method, the corresponding system service will query the system whether the required permissions are hold by this calling app, which is identified by its unique UID.

There are two major types of permissions in Android: (1) signature or system permissions that are only available to privileged services and content providers, and (2) regular permissions that are available to all apps. To conduct a more comprehensive analysis of Android permissions, our work focuses on extracting permission specification for both types.

Android permissions are checked at two different levels. High-level permissions are checked at the framework level (implemented in the Java code), while low-level permissions are checked in C/C++ native services (the media service for instance) or in the kernel (e.g., when creating a socket). Most permissions are high-level, e.g., Android 2.2 contains a total of 134 permissions, where 126 are checked in the Java code

```
1  // Defined in the LocationManagerService class
2  Context mContext;
3  List<String> getProviders(Criteria cr, boolean e) {
4    int level = getAllowedResolutionLevel(pid, uid);
5    ...
6  }
7
8  int getAllowedResolutionLevel(int pid, int uid) {
9    if (mContext.checkPermission(android.Manifest.
         permission.ACCESS_FINE_LOCATION, pid, uid) ==
         PackageManager.PERMISSION_GRANTED) {
10     ...
11   }}
```

(a) Code snippet of the `LocationManagerService` class.


LMS.getProviders
LMS.getAllowedResolutionLevel
...
...
Context.checkPermission
ContextWrapper.checkPermission
...
MockContext.checkPermission
ContextImpl.checkPermission
BridgeContext.checkPermission

(b) Sub-call graph rooted at the framework API method `getProviders(Criteria, boolean)`.

Fig. 1: A motivating example. (Legend: LMS represents `LocationManagerService`.)

and 8 checked in the native C/C++ code. Our work focuses on the high-level permissions, similar to the prior work [3], [4].

D. Heap Memory Snapshot

A heap memory is a collection of classes and objects for a Java program. It is created when the program starts, and may change as the program runs.

A heap memory snapshot (called a *snapshot* for short) is a memory image of the *current* execution context of a program. It contains all the classes and objects for a program. Android Framework is mainly implemented in Java. A heap memory snapshot of a process of Android Framework is a collection of classes and objects for the process at the time when the snapshot is captured.

III. Overview

A. Motivation and Observation

To analyze the Android permission specification, a critical step is to build a global (inter-procedural) call graph of the framework, which involves many challenges. One main challenge is *how to deal with virtual function calls.* As common for object-oriented languages, the target of a method call depends on the runtime type of the receiving object, which is undecidable, in general, *statically*.

Consider the example in Figure 1, where Figure 1(a) shows the code snippet of the `LocationManagerService` class.[1] We want to generate the call graph for the `getProviders` method. It first invokes the `getAllowedResolutionLevel` method (Line 4), which further invokes the `checkPermission` method via `mContext` (Line 9), a reference variable of the `Context` type (Line 2). `Context` is an abstract class extended by four classes, including `ContextWrapper`, `ContextImpl`, `BridgeContext`, and `MockContext`; each implements the `checkPermission` method and is further inherited by other classes. Without the runtime type information of the object pointed to by `mContext`, it is hard to determine the dispatch target of the method call. Such virtual function calls prevail in the framework code.

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

Most existing work adopts a conservative approach and attempts to extract an *over-approximated* of the framework's call graph, where a virtual function call to a class is considered to target all its subclasses [3], [13]. For example, `PScout` generates the call graph using Class Hierarchy Analysis [15], based on the following rules: (1) a virtual call to a class can potentially targets all its subclasses; (2) an interface call can be resolved to call any class that implements the interface and its sub-interfaces; and (3) the target method of each subclass is the closest ancestor that implements the method. This approach obviously results in a *low-precision* call graph, as `PScout` needs to consider all possibilities if the runtime type of the receiving object is undecidable.

Figure 1(b) shows the sub-call graph rooted at the `getProviders` method generated by `PScout`, where the `getAllowedResolutionLevel` method is first connected to the `checkPermission` method of the `Context` class, which is further connected to the `checkPermission` method of *all* the subclasses of `Context`.

To address the issue, `AXPLORER` uses points-to analysis to infer the possible dynamic type of the receiving object of a virtual function call, and only connects the invocation to the corresponding target [4]. However, its points-to analysis cannot always infer runtime type information accurately, and still causes much imprecision (Section VI-C).

Thus, *how to precisely determine the dispatch targets of virtual function calls is challenging and critical for generating a precise call graph of Android Framework for extracting the permission specification via static analysis.*

Our observation is that the execution of Android Framework consists of the *initialization* phase and the *ready-for-use* phase. After the initialization of the system services is done, much object-type information is determined and plenty of useful dynamic information is stored in the heap of the framework process. We thus propose the *heap memory snapshot assisted program analysis* that leverages the dynamic information stored in the heap to assist in generating a more precise call graph of the framework. Very interestingly, as all the system services in Android Framework adopt the singleton design pattern [34], given the system service, there is no ambiguity to locate the

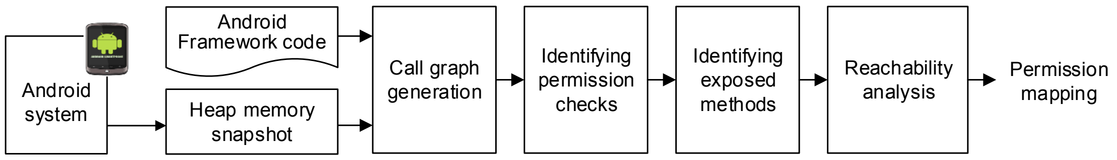


Fig. 2: Architecture of HEAPHELPER.

object for the system service from the heap and, hence, there is no ambiguity to obtain the dynamic types of all the objects pointed to by the reference fields of the system service object.

During the call graph generation, for each virtual function call, whenever possible we extract the runtime type of the receiving object from the heap snapshot, and precisely connect the invocation to its target. This way, we minimize the imprecision introduced by static analysis based inference, which largely lowers *false positives*—i.e., permissions not needed by an API method are falsely considered as needed.

The heap may vary with time (e.g., the runtime object types of some references may change as the framework runs). However, our another observation is that, once the framework enters the ready-for-use phase, any API method could be issued by an application. Given an API method, the needed permission(s) should *not* change over time, as the permission specification should keep consistent, no matter whether the method is invoked right after the initialization or after a long time. Our experiments based on snapshots captured at different times confirm this point (Section VI-D).

B. Architecture

The architecture of HEAPHELPER is shown in Figure 2. The extraction of the permission specification from Android Framework consists of the following five steps. (1) The initialization phase of Android Framework is first run as whole system concrete execution; then the heap memory snapshot is dumped, which is fed to the call-graph generation component along with the Android Framework class files. (2) The call-graph generation component leverages Soot [43], the Java bytecode analysis framework to perform static analysis and builds a precious call graph over the entire Android Framework (Section IV). (3) All the permission checks in the framework code are identified and labeled (Section V-A). (4) The framework API methods that are exposed to third-party applications are identified (Section V-B). (5) Finally, HEAPHELPER performs a backward reachability traversal over the call graph to identify all exposed API calls that could reach a particular permission check (Section V-C). The output is a mapping between each of the framework API methods and its required permission(s) found by HEAPHELPER. We next present these steps separately.

IV. CALL GRAPH GENERATION

The main challenge of building a precise call graph of Android Framework is how to handle virtual function calls. We divide virtual function calls into four categories, as showed in Figure 3. The first three are due to unique characteristics of

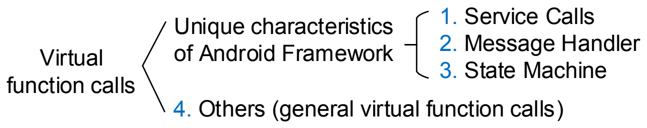

Virtual function calls
Unique characteristics of Android Framework
1. Service Calls
2. Message Handler
3. State Machine
4. Others (general virtual function calls)

Fig. 3: Four categories of virtual function calls.

```
1  Load Class:
2    class object id: 319195136
3    class string: com.android.server.
         LocationManagerService
4  Load Class:
5    class object id: 1875661552
6    class string: android.app.ContextImpl
7  ...
8  Instance Dump:
9    object id: 316207456
10   class object id: 319195136
11   instance field values:
12     mContext = 315905152
13     ...
14 Instance Dump:
15   object id: 315905152
16   class object id: 1875661552
17   instance field values:
18     ...
```

Fig. 4: Part of a standard `.hprof` file.

Android Framework, and the last one is general virtual function calls, commonly existing in object-oriented programs. Below we first present how to parse a heap snapshot and then discuss how each of the four categories is addressed.

A. Capturing and Parsing Snapshot

We first run the initialization phase of Android Framework as concrete execution in a device, and then dump the heap snapshot of the framework process (using the *dumpheap* utility in the Android SDK) from the device. Next, the snapshot is converted to a `.hprof` file using the `hprof-conv` utility.

The standard Java `.hprof` file is then passed by our tool to build a database, called *ObjType*, storing the runtime type information of the objects in the heap. Figure 4 shows an example of the `.hprof` file. For each *instance dump* (corresponding to an instance object) in the `.hprof` file, we first check its class type, and then extract all its instance fields. If an instance field is a reference variable, we search for the runtime type of the object pointed to by it from the `.hprof` file. E.g., in Figure 4, (1) the object at Line 8 is the `LocationManagerService` type, as its class object id (Line 10) is associated with the class `LocationManagerService` (Line 2-3). (2) We then extract all its instance fields; one of them is the reference variable `mContext` (Line 12). (3) Based on the value of mContext, we look for the object pointed to by mContext, and find it is the instance at Line 14. (4) Next, based on the class id of this instance (Line 16), we identify its runtime type is ContextImpl (Line 4-6). (5) Finally, we insert an entry to the *ObjType* database, where the key is the variable mContext and the value is ContextImpl.

Besides *instance dumps*, we also parse *class dumps* (each class dump corresponds to a class object). For each class dump, we extract all its static fields, and search for the runtime type of the object pointed to by each static field if it is a reference variable. Due to space limitation, we skip the discussion here, which is similar to the process above.

B. Handling General Virtual Function Calls

We first present how to handle the general virtual function calls, which belong to the fourth category in Figure 3.

Process. During the call graph generation, for each virtual function call, we first determine the reference variable through which the function call is issued, and then query the runtime type of the object pointed to by the reference variable, if initialized, from the *ObjType* database. After that, we precisely connect the invocation to its target. Consider the code snippet in Figure 1(a), we first determine the reference variable that issues the virtual function call to checkPermission is mContext (Line 9). We then find the runtime type of the object pointed to by mContext is ContextImpl. Finally, we directly connect getAllowedResolutionLevel to checkPermission of the ContextImpl class.

One situation needs special attention. Assume the runtime type of the receiving object of a virtual function call to the method M is identified to be the class A; but the class A does not implement M. In such a case, rather than considering the method M of A as the target, we need to find the *closest ancestor* of A, assume it is B, that implements M; then the method M of B is considered as the target.

JVM instruction needed to be intercepted. The JVM has four instructions for invoking methods: invokevirtual, invokeinterface, invokestatic, and invokespecial. The first two instructions are used to invoke instance methods based on *dynamic dispatch* mechanism to determine the correct method to invoke at runtime. The third one is used to invoke class methods based on the *static* type of the receiving object. The last one is used in certain special cases to invoke the initialization method, or a method in a superclass, or a private method where the invocation is based on the compile-time type of the receiving object. Therefore, only the invokevirtual and invokeinterface instructions are needed to be intercepted during the call graph generation.

C. Handling Service Calls

Android Framework provides a collection of system services, where most run as threads in the system_server process [25], and some in other processes (e.g., com.android.server.telecom and com.android.keychain). Thus the heap snapshots of all

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

Fig. 5: An *intra*-process service call example.

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

Fig. 6: An *inter*-process service call example.

the processes need to be parsed to build the *ObjType* database. To correctly distinguish references variables from different snapshots, each reference variables is extended into the form of pid:reference_variable, where pid refers to the ID of the process from which the snapshot is captured.

A system service provides a client-server interface. Depending on whether the client and the service are in the same process, the service call is handled in two different ways.

Handling* intra*-process service calls. When the client and the service are in the same process, the service call is made through *intra*-process communication. Figure 5 shows an example, where both the Location Manager Service and User Manager Service belong to the system_server process. The call at Line 4 leads to a service call at Line 10, which issues an intra-process service call.

Note that UserManager.mService is the type IUserManager, which is extended by many classes (including the Proxy/Stub classes to be introduced below and UserManagerService). Previous research relies on expert knowledge to manually specify the dispatch target of the call at Line 10 to facilitate static analysis [41], [13], [3], while HEAPHELPER uses the runtime information provided by the execution context. We thus can find that the object pointed to by UserManager.mService is the UserManagerService type, and hence the call is handled as an ordinary method call. Thus, manual effort are not needed.

Handling* inter*-process service calls. When a system service invokes a method of another service in a *different* process, the call is handled via an *inter*-process communication mechanism named Binder [17]. The system service interface is defined using the Android Interface Definition Language (AIDL). 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 corresponding service.

Figure 6 shows an inter-process system service call example, where the `Telecom Account Registry` (running in the `com.android.phone` process) invokes the `getAllPhoneAccountHandles` method exposed by the `Telecom Service` (running in the `com.android.server.telecom` process). The call at Line 15 invokes the method at Line 19. To perform the 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, in Figure 6, the client first invokes `ServiceManager.getService(TELECOM_SERVICE)` to get the `Proxy` of the `Telecom Service` (Line 20), and then performs the service call via this `Proxy`. After that, the `Binder` marshalls parameters of the service call and reassembles the objects in the remote process, one thread of which will execute the corresponding service method.

The functionality of `Binder` is mainly implemented in native libraries and the kernel, which cannot be interpreted by HEAPHELPER. To solve it, 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. Through a simple class hierarchy analysis of the framework code, 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 mapping, we automatically build a hash table containing the mapping between each `Proxy` type and the name of the corresponding service class (which extends the `Stub` generated together with the `Proxy` type).

Through this, during the call graph generation, HEAPHELPER intercepts `I*.Stub.asInterface()` (Line 20). Using the `Proxy` type of the parameter of `asInterface()`, it queries the hash table to obtain the corresponding system service class name, and directly connects the service call to the method of this service class.

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.

D. Handling Message Handler

Two messaging mechanisms that are frequently used by system services are `Message Handlers` and `State Machines`. `Message Handlers` are implemented through `Binder`, so they cannot be interpreted by our tool.

To deal with `Message Handler`, we first infer the run-time type of the handler object, which can be obtained from the *ObjType* database. We then connect the call to `sendMessage` to a call to the destination handler's `handleMessage`.

However, only a part of `handleMessage` is responsible for reacting to the `sendMessage` method (determined by the particular message sent), which is implemented as a `switch` statement in `handleMessage`; we thus perform the *path-sensitive* analysis to improve the precision—based on the message sent by `sendMessage`, only the functions included in the corresponding code block of the `switch` statement in `handleMessage` is connected to this `sendMessage`.

E. Handling State Machine

A `State Machine` can also be used to send and process messages. A `State Machine` sends a message by invoking `sendMessage`, while the *current state*'s `processMessage` is called to process a message.

To handle `State Machine`, we need to connect the invocation of `sendMessage` to *all* possible states' `processMessage`, as the *current state* may vary with time. A `State Machine` object contains a field that points to an `mSmHandler` object, one field of which, `mStateStack` (an ArrayList), is used to identify all possible states. Specifically, we iterate `mStateStack` and retrieve each element. Each element is the `StateInfo` type, and contains a field, `state`, which points to a possible state (= `mStateStack[index].state`, where $0 \leq$ `index` $<$ `mStateStack`.length). Next, we search for the class type of the object pointed to by each `state` from the *ObjType* database, and connect `mSmHandler.sendMessage` to the code block of each state's `processMessage` that processes the corresponding message sent by the sender (i.e., we also perform the *path-sensitive* analysis). This way, we connect the sender and receiver for messages sent via `State Machine`.

V. PERMISSION SPECIFICATION ANALYSIS

HEAPHELPER next extracts the permission specification based on the generated call graph, involving three steps.

A. Identifying Permission Checks

We adopt the method of `PScout` [3] to identify the permission checks. There are three types of permission checks. The first type is `checkPermission` or its wrapper functions. Permissions in Android appear as string literals, which are passed to `checkPermission` or its wrappers to check whether the calling app is granted with the particular permissions. HEAPHELPER searches the call graph for all the permission checks of this type, and extracts the corresponding permission string literals, which are used to label the permission checks. HEAPHELPER can successfully extract 98% of the permission strings; e.g., one of the failing cases is in `ActivityManagerService$PermissionController`, where the permission string is an argument of the method called from native code and, hence, cannot be resolved.

The second type is methods involving `Intent`. Two situations are involved. (1) The required permission is specified in the manifest file; HEAPHELPER then extracts each permission and its associated `Intent` actions from the manifest file. (2) The required permission is programmatically coded in the framework methods, including `sendBroadcast`, `registerReceiver`, and their variants; HEAPHELPER then searches the call graph for these methods, and extracts the corresponding permission strings and the associated `Intent` actions. Through this, HEAPHELPER builds a mapping between permissions and `Intent` actions.

The third type is methods involving `Content Providers`. (1) The permissions required to read/write to a `content provider` can be declared in its manifest file; thus, HEAPHELPER parses the manifest file to extract the information, and builds a mapping between permissions and content provider URIs. (2) The permission can be checked at runtime by each `content provider` via invoking `checkPermission`, which is handled in a similar way as the first type of permission check.

B. Identifying Exposed Framework Methods

For each permission check, many methods can reach it; e.g., for the call path, $A \rightarrow B \rightarrow C \rightarrow P$, where P is a permission check, A, B, and C are the methods that can reach P. It is impractical to report all these methods (A, B, and C), as from the perceptive of app developers, they are only interested in the required permissions for the methods exposed to third-party applications. Thus, it is necessary to identify the exposed framework methods and only report the permission mapping involving such methods.

Android Framework contains a large number of framework classes, which expose functionality via `Binder` interfaces to the application layer. We thus first locate the framework classes that are exposed via `Binder` interfaces, and then identify the exposed framework methods (including both *public* and *private*, as applications can invoke the private methods via Java reflection) from these classes.

C. Reachability Analysis

The last step performs a backward reachability analysis for each permission check, and records every exposed framework method that can be reached from a permission check.

Similar to `PScout` [3], the backward reachability analysis stops if one of the two conditions is met. (1) Any calls located between `Binder.clearCallingIdentity` and `Binder.restoreCallingIdentity` are irrelevant for permission analysis, which is a stopping point. (2) HEAPHELPER stops when it reaches a class or subclass of `ContentProvider`; it then extracts the URI string of the content provider; and the permission mapping is the URI and the permission included in the permission check where the reachability analysis starts.

VI. EVALUATION

A. Experimental Settings

We have implemented the system HEAPHELPER, which is built upon the Java bytecode analysis framework Soot [43].

We first analyze the statistics of various heap snapshots captured from different Android versions (Section VI-B). We then compare HEAPHELPER against `PScout` and `AXPLORER` in terms of the accuracy of the generated permission mapping (Section VI-C). We next investigate the reliability of our approach—we conduct experiments based on snapshots captured at different times and from different devices, and analyze the consistency of the results (Section VI-D).

The experiments are performed on a machine with an Intel Core i7 4.0Ghz Quad Core processor and 32GB RAM running Linux kernel 3.13. We conduct our experiments on three different devices, including Nexus 4, Nexus 5X, and Nexus One, with different versions of Android systems.

B. Heap Snapshot Statistics

Once the framework is initialized, we dump the heap memory snapshots of all the framework processes. Table I summarizes the statistic results with respect to different Android versions. As we do not have access to the source code of `AXPLORER`, we only compare the statistics to that of `PScout`.

It can be observed that the call graph generated by our tool is largely simplified compared to that generated by `PScout`. E.g., for Android version 5.0, the call graph generated by `PScout` consists of 953,602 edges, while that generated by HEAPHELPER only contains 406,638. The reason is that HEAPHELPER extracts the runtime type of the receiving object of a virtual function call from the snapshot (if contained) and precisely connects the call to its target, while `PScout` considers all subclasses of a virtual method call as receivers.

However, some objects have not been instantiated during the initialization phase—e.g., function local variables will be instantiated when the function is executed and the variables are accessed. In this case, we adopt the points-to analysis to infer the possible types of the receiving objects. We appreciate the authors [9] to share their points-to analysis code to us.

Note that although we apply points-to analysis to *non-instantiated* variables, our generated call graph—by a novel and effective combination of dynamic information and static analysis—is still much more precise than existing work that relies on *pure* static analysis approaches.

C. Comparison with Prior Work

We compare HEAPHELPER to `PScout` [3] and `AXPLORER` [4]. Due to space limitation, we focus on presenting the comparison for Android version 5.0.

1) How many framework APIs check the same permission? We first seek to understand the number of framework APIs that check a certain permission. The comparison results are fairly deviating. To confirm the correctness of our result, we manually investigate all the deviating cases, and find the following reasons that cause the differences.

TABLE I: STATISTIC RESULTS OF HEAP SNAPSHOTS DUMPED ONCE ANDROID FRAMEWORK IS INITIALIZED. THE DEVICE IS NEXUS 4.

Android version	4.1	4.2	4.4	5.0	5.1	6.0	9.0
# of objects requested when building call graph	16,042	16,257	16,734	17,867	17,895	18,387	20,024
# of instantiated objects in heap snapshot	5,823	5,882	6,023	6,270	6,017	7,105	7,625
# of call graph edges generated by `PScout`	2,062,839	1,174,387	2,365,902	953,602	1,028,134	1,424,065	1,861,473
# of call graph edges generated by HEAPHELPER	496,774	403,165	502,354	406,638	419,526	502,381	584,428

TABLE II: COMPARISON BETWEEN OUR RESULTS AND `PSCOUT`'S RESULTS BASED ON ANDROID VERSION 5.0.

Permission set	# of methods
# of APIs in `PScout`	7,287
# of APIs in both HEAPHELPER and `PScout`	1,314
# of APIs in HEAPHELPER and `PScout` with the same permission size	872
# of APIs in HEAPHELPER that have less permission checks	434
# of APIs in HEAPHELPER that have more permission checks	8

First, `PScout` includes mappings for unexposed framework methods. E.g., it includes the mappings for `State Machine` methods; however, `State Machines` are used framework-internally and their functionality is not exposed to applications; thus, all the mappings involving `State Machine` methods should be excluded, and there are totally 1,692 of such mappings. Second, it reports mappings for synthetic accessor methods as well as methods of inner classes, which should also be excluded, and there are totally 12,368 of such mappings. Third, for the methods of AIDL-based classes, it counts for several times, which, however, should be only counted once, E.g., it counts four times for `getCompleteVoiceMailNumber` (in `Stub`, `Proxy`, manager, service class, respectively), while only one is actually needed. The problem is probably due to `PScout` does not have a proper definition of the exposed framework methods and includes many unnecessary mappings. `AXPLORER` reports higher numbers for `BROADCAST_STICKY` and `SET_WALLPAPER`, which mainly refers to abstract methods from the `Context` class that are implemented in its subclass `ContextWrapper` and then inherited by a set of non-abstract subclasses.

We further compare the points-to set of the instantiated objects to their runtime types stored in the heap snapshot, and find that points-to analysis identifies more than one possible types for 4,372 objects (69.7%). E.g., for the virtual function call to `removeCallBacks` via `ViewPropertyAnimator.mView`, the points-to set of this object has 36 types, including `ViewGroup`, `WaveView`, `TextView`, and `EditText`, etc, while the snapshot stores only `LinearLayout`. Thus, `AXPLORER`, relying on points-to analysis, still introduces spurious paths in the call graph.

Note that the runtime types of some objects may change as the framework runs; however, our observation is that given an API method, the needed permission(s) should *not* change over time, as the permission specification should keep consistent, no matter when and how the method is invoked, which is confirmed by our experiment (see Section VI-D).

II) How many permissions are checked by the same framework API? We seek to understand the number of permissions checked by framework APIs. Table II summarizes the comparison result between `PScout` and HEAPHELPER. For Android 5.0, (1) 1,314 methods are found by `PScout` and HEAPHELPER. (2) Among these 1,314 methods, 872 have the same permission size in both `PScout` and HEAPHELPER. (3) HEAPHELPER finds 434 methods with less permissions, and 8 methods with more permissions, which are confirmed via manual investigation.

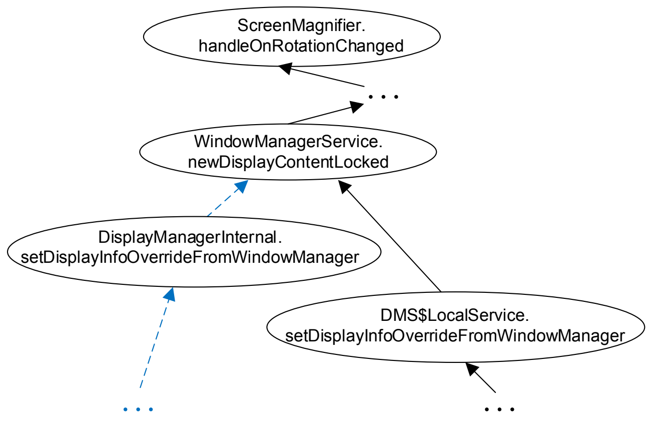


Fig. 7: Sub-call graph generated by `PScout`. `DMS` represents `DisplayManagerService`. The blue dashed lines are spurious paths causing an extra permission `WRITE_SETTINGS` to be reported.

For instance, `PScout` finds more than ten permissions for `TouchExplorer.clear`, whereas HEAPHELPER finds only three, including `INTERACT_ACROSS_USERS`, `READ_PROFILE`, and `INTERACT_ACROSS_USERS_FULL`. `PScout` reports more than nine permissions for `ScreenMagnifier.handleOnRotationChanged`, whereas HEAPHELPER only three. The reason why `PScout` reports more permissions is that it builds an over-approximated call graph that includes too many spurious paths. E.g., Figure 7 shows the sub-call graph rooted at the `handleOnRotationChanged` method, where one virtual method call is `setDisplayInfoOverrideFromWindowManager`. As `PScout` considers all subclasses as the targets, the `setDisplayInfoOverrideFromWindowManager` method of the two classes, `DisplayManagerInternal` and `DisplayManagerService$LocalService`, are both included as the call sites. On the contrary, we use the heap snapshot to determine the runtime type of the receiving object, which is `DisplayManagerService$LocalService`, and only connect the correct target. The spurious paths (i.e., the blue dashed lines in Figure 7), which are included by `PScout` but cause an extra permission `WRITE_SETTINGS` to be reported, is excluded. There are a plenty of such cases that `PScout` fails to identify the correct targets.

TABLE III: Statistic results of heap snapshots dumped after random user interactions. The Android version is 5.0.

Devices and user interactions	Nexus 4			Nexus 5X			Nexus One		
	5m	20m	20m, 4 apps	5m	20m	20m, 4 apps	5m	20m	20m, 4 apps
# of instantiated objects in heap	6,645	7,192	6,822	5,978	6,321	6,014	5,881	6,105	6,262
# of call graph edges generated	406,880	432,320	432,450	406,857	406,732	406,854	406,898	406,762	406,843

Interestingly, we also find 8 methods where HEAPHELPER finds more permissions than `PScout`. E.g., `PScout` reports one permission `CONNECTIVITY_INTERNAL` for invoking `NsdService.setEnabled`, while HEAPHELPER reports two, `CONNECTIVITY_INTERNAL` and `WRITE_SETTINGS`. Another example is `SendUiCallback.onSendConfirmed`. `PScout` reports `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL`, while HEAPHELPER reports `INTERACT_ACROSS_USERS`, `INTERACT_ACROSS_USERS_FULL`, and `UPDATE_APP_OPS_STATS`. Moreover, `PScout` misses `WRITE_SETTINGS` for the `setTimeFromNITZString` method in the `GsmServiceStateTracker` class.

We are particular interested in understanding why `PScout` misses permissions. As it builds an over-approximated call graph, theoretically it should not miss permissions. We analyze the `PScout`'s code on how it builds the call graph and how it performs the backward reachability analysis. Our finding shows that `PScout` sets up some experience-based stop conditions for the backward analysis to avoid the path-explosion problem due to the over-approximated call graph. For example, it stops when it meets `onTerribleFailure`; it also stops when the backward traversal depth reaches 5. However, as HEAPHELPER builds a more precise call graph (by leveraging the heap information) with many spurious paths removed, HEAPHELPER does not stop under such conditions and keeps backtracking the call graph, and hence, can find more permissions. Besides, `PScout` probably does not correctly link some entry point methods to all methods they can reach, and thus misses some permissions. Therefore, our analysis yields more precise results.

D. *Consistency of Permission Specification*

As the dynamic information stored in the heap snapshot may change as the framework runs, we investigate *the consistency of the permission specification* in terms of different snapshots captured at different times and from different devices. We use three devices with Android 5.0, including Nexus 4, Nexus 5X, and Nexus One. After the system was initialized and different kinds of user interactions were performed, three snapshots were captured from each device. There are totally nine snapshots.

We first investigate whether user interactions affect the dynamic information stored in the heap. Table III summarizes the results; "*20m, 4 apps*" represents a snapshot is captured at intervals of 20 minutes and 4 apps are installed during the interval. It can be observed that user interactions trigger more objects to be instantiated. E.g., the snapshot dumped from Nexus 4 after the framework is initialized contains 6,270 instantiated objects (see Table I), and the snapshot dumped after 5 minutes user interactions (e.g., changing system settings, storing new contacts) contains 375 (=6,645-6,270) more instantiated objects. One example of the new instantiated objects is `mContentResolver` in the `PduPersister` class, and its runtime type is `ApplicationContentResolver`. For Nexus 5X, the snapshots dumped after 5 minutes and 20 minutes contain 5,978 and 6,321 instantiated objects, respectively. Upon comparing the two snapshots, we find that 5,783 objects are instantiated with the same type in both snapshots.

Some instantiated objects' types changed after user interactions. E.g., for Nexus 5X, the `mView` variable in the `ViewPropertyAnimator` class has the type of `CellLayout` and `LinearLayout` in the snapshots dumped with and without apps installed, respectively; `mDrawable` in the `ImageSpan` class has the type of `BitmapDrawable` and `VectorDrawable` in the snapshots with and without apps installed, respectively. The differences are mainly due to various status of the installed apps.

We also analyze how the heap changed cross different devices. The snapshots dumped from Nexus 5X and Nexus One after 5 minutes user interactions contain 5,978 and 5,881 initialized objects, respectively. Upon comparison, we find that there are 5,631 objects whose types do not change in both snapshots, and 250 objects are only contained in the heap from Nexus One, and 347 only in the heap from Nexus 5X. Similarly, for the snapshots dumped from Nexus 5X and Nexus 4 after 5 minutes user interactions, there are 5,547 objects instantiated with the same type in both snapshots.

We now seek to understand whether different heap snapshots lead to different permission specifications. We conduct experiments on each of the 9 snapshots, and the results show that the permission specification is consistent.

Several reasons can explain the consistency. First, although apps can manipulate the values of some system variables in the framework (e.g., GPS locations, contact information, etc.), the values of the system variables do not affect the call graph generation. For example, in Figure 1a, the `getProviders` method returns the names of the GPS providers that the calling app is allowed to access. The framework's call graph does not depend on the concrete values of the related system variable (i.e., `LocationManagerService.mProviders`), although different provider names may be returned by the system service call when different providers are installed.

Second, the configurations and statuses of installed and running applications themselves do not affect the framework's call graph. E.g., if an app has been installed, the influence is that one element would be inserted to the object `Settings.mUserId`, an ArrayList storing the information of all the installed apps with one element for each app. However, this will not affect the dynamic type of `Settings.mUserId`, and thus it causes no impact on the framework's call graph.

This result confirms our observation that given an API method, its needed permission(s) should *not* change over time, as the permission specification should keep consistent, no matter whether the method is invoked right after the initialization of Android Framework or after a long time.

VII. Discussion

A. Applications

Inconsistent Security Policy Enforcement (ISPE) Vulnerabilities. A sensitive operation may be reached from different execution paths enforcing security checks inconsistently, such a vulnerability is called inconsistent security policy enforcement (ISPE). A system called `Kratos` was built to such vulnerability [41]. It first builds an *over-approximated* call graph of Android Framework, and then finds all the paths that can reach the same sensitive operation, and reports it as an ISPE vulnerability. It can benefit from our work by using a more precise call graph; false positives originating from a low-precision call graph could be automatically eliminated.

Static (App) Analysis. Many static analysis techniques have been proposed to analyze Android app vulnerabilities and privacy violations. Enabling precise static app analysis requires to include Android Framework. `EdgeMinder` [13] analyzes Android Framework to extract implicit control flow transitions for the security analysis of apps. `DroidSafe` [26] distills a data-dependency-aware model of the Android app API and runtime from the framework code. One of the building blocks that these systems rely on is the computation the framework's call graph. Most of them either use manual work, or apply ad-hoc heuristics, or build an over-approximated call graph of the framework, which are insufficient and imprecise and allow malicious apps to evade detection. Thus, it is beneficial to integrate our technique into these analysis systems and build a more precise call graph to improve the precision.

B. Limitations

Android framework is written in different languages and mainly implemented in Java. Our approach is designed to analyze Java code, and will miss the permissions (only a small part) checked in C/C++ native code.

HEAPHELPER cannot analyze reflection. Android Framework uses reflection in only seven classes. Six are in debugging classes, and the last one in the `View` class for handling animations. But no permission is checked in these cases.

Static analysis cannot handle dynamic class loading as the loaded classes are only known at runtime. Android Framework uses the `loadClass` method in eight classes, which are either not linked to permission checks, or permissions checks are made before loading classes; no permission check is missed.

HEAPHELPER leverages the dynamic information in the heap to generate a more precise call graph of the framework. Some objects may not be instantiated when the heap is dumped. Although we apply points-to analysis to non-instantiated variables, our generated call graph is still much more precise than existing work that relies on *pure* static analysis approaches.

VIII. Related Work

There is a large body of security research on permission-based systems [23], [24], [9], [3], [4], [16], [40], [35] and in Android security [10], [12], [14], [19], [20], [21], [31]. The permission specification has been a valuable input to different Android security researches, such as permission analysis [28] and compartmentalization [39], [42] of third-party code, studying app developer behavior [22], [44], detecting component hijacking [33], app virtualization [6], [32], [7], and risk assessment [38], [27], [47], [46]. The approaches for extracting Android permission specification can be generally divided into two categories: dynamic-based, and static-based.

Dynamic-based Approaches. Stowaway [22] extracts the Android permission specification using unit testing and feedback directed API fuzzing; however, it is very difficult to examine the entire Android Framework—i.e., it is difficult to achieve a high code coverage, which is consistent with the well-known challenge in software testing: even with significant time and effort invested in testing, a commercial program typically can only be tested under a small portion. Dypermin [35] also relies on dynamic analysis and suffers from the same limitation.

Static-based Approaches. Bartel et al. [8] extract the permission specification from the call graph of the framework. But their result is incomplete as they only infer permission checks on the `checkPermission` function while skipping Intent and Content Provider functions; in addition, their framework call graph is over-approximated, causing high false positives. `PScout` [3] considers all the permission checks, but it also builds an over-approximated call graph that introduces many spurious paths, resulting in unnecessary imprecision. `AXPLORER` [4] builds a static runtime model of the framework by inferring the dynamic types of receiving objects and connecting the invocations to the corresponding targets; but the inference is based on points-to analysis, and the inferred results are not always accurate. In contrast, our work combines dynamic information and static analysis by leveraging the dynamic information stored in the heap to generate a more precise call graph for assisting in the permission specification analysis and improves the result precision.

IX. Conclusion

We introduce the *heap memory snapshot assisted program analysis* that leverages the dynamic information stored in the heap of Android Framework execution to assist in generating a more precise call graph of the framework. It is a novel and effective combination of dynamic information and static analysis. This technique can be integrated into the (existing) static analysis systems to improve their precision. We have developed a prototype, and evaluated it on different versions of Android Framework. Our experiment results show that our approach improves on the prior work that relies on *pure* static analysis, and produces much more precise results.

Acknowledgment

This work was supported by NSF CNS-1850278 and NSF CNS-1815144.

References

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices*, volume 49, pages 259–269. ACM, 2014.

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie. Short paper: a look at smartphone permission models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 63–68. ACM, 2011.

[3] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[4] M. Backes, S. Bugiel, E. Derr, P. D. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *USENIX Security Symposium*, pages 1101–1118, 2016.

[5] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th annual computer security applications conference*, pages 46–55. ACM, 2014.

[6] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security Symposium*, pages 691–706, 2015.

[7] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard-enforcing user requirements on android apps. In *TACAS*, pages 543–548. Springer, 2013.

[8] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM, 2012.

[9] 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.

[10] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2008.

[11] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security Symposium*, pages 131–146, 2013.

[12] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[13] 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.

[14] A. Chaudhuri. Language-based security on android. In *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security*, pages 1–7. ACM, 2009.

[15] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[16] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis. Reaper: Real-time app analysis for augmenting the android permission system. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 37–48. ACM, 2019.

[17] A. D. Documentation. Binder. https://developer.android.com/reference/android/os/Binder.html.

[18] A. D. Documentation. Bound Services. http://developer.android.com/guide/components/bound-services.html.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[20] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[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] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[23] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7, 2011.

[24] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, 2011.

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

[26] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

[27] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[28] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[29] GSMA. The Mobile Economy 2016. http://www.gsma.com/mobileeconomy/.

[30] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *USENIX Security Symposium*, pages 1005–1019, 2014.

[31] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.

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

[33] 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. ACM, 2012.

[34] 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 *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 225–238. ACM, 2017.

[35] C. Lyvas, C. Lambrinoudakis, and D. Geneiatakis. Dypermin: dynamic permission mining framework for android platform. *Computers & Security*, 77:472–487, 2018.

[36] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 328–332. ACM, 2010.

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

[38] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium*, volume 2013, 2013.

[39] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72. Acm, 2012.

[40] A. Sadeghi, R. Jabbarvand, N. Ghorbani, H. Bagheri, and S. Malek. A temporal permission analysis and enforcement framework for android. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 846–857. IEEE, 2018.

[41] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[42] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, volume 2012, 2012.

[43] Soot. http://sable.github.io/soot/.

[44] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proceedings of the Web*, volume 2, pages 91–96, 2011.

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

[46] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[47] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[48] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *International conference on Trust and trustworthy computing*, pages 93–107. Springer, 2011.