Hey, You, Get Off of My Image: Detecting Data Residue in Android Images

Xiao Zhang, Yousra Aafer, Kailiang Ying, and Wenliang Du

Dept. of Electrical Engineering & Computer Science,
Syracuse University, Syracuse, New York, USA
{xzhang35,yaafer,kying,wedu}@syr.edu

Abstract. Android's data cleanup mechanism has been called into question with the recently discovered data residue vulnerability. However, the existing study only focuses on one particular Android version and demands heavy human involvement. In this project, we aim to fill the gap by providing a comprehensive understanding of the data residue situation across the entire Android ecosystem. To this end, we propose ANRED[1], an ANdroid REsidue Detector that performs static analysis on Android framework bytecode and automatically quantifies the risk for each identified data residue instance within collected system services. The design of ANRED has overcome several challenges imposed by the special characteristic of Android framework and data residue vulnerability. We have implemented ANRED in WALA and further evaluated it against 606 Android images. The analysis results have demonstrated the effectiveness, efficiency and reliability of ANRED. In particular, we have confirmed the effect of vendor customization and version upgrade on data residue vulnerability. We have also identified five new data residue instances that have been overlooked in the previous study, leading to data leakage and privilege escalation attacks.

1 Introduction

The prosperity of Android ecosystem contrasts sharply with the short lifespan of applications (apps, in short) on devices. Seemingly harmless, a recent study [40] has uncovered the data residue vulnerability arising from the app uninstallation process on the Android platform. In particular, when an app is uninstalled from the device, its data may be still scattered stealthily around privileged system services within Android framework. More surprisingly, the protection on these data leftover is problematic, empowering unauthorized adversaries to access users' sensitive information, such as credentials, emails and bank statements. As a pioneer that sheds lights on the data residue vulnerability, the above-mentioned study limits its scope to AOSP version 5.0.1 with the requirement of source code and significant manual effort. However, the extensive customization on Android devices from different vendors and the high fragmentation of Android operating system demand an automatic, scalable and source code independent framework for data residue detection.

[1] ANRED is a former French public institution for the recovery and disposal of waste.

To this end, we propose ANRED, an ANdroid REsidue Detector that takes an Android device image as input and automatically quantifies the risk for each identified data residue instance within collected system services. The design of ANRED has overcome several challenges. First, we have employed several novel techniques to generically preprocess Android images from different vendors, accurately pinpoint all system services within the given image, identify entry points for each system service and connect the broken links on the call graph resulting from the event-driven nature of Android system. Second, we have inferred data residue instances from mismatches between the deleting data set and the saving data set. To retrieve those two data sets, we have divided the original call graph into two subgraphs originating from the saving and deleting entry points. While the saving entry points capture all interactions with the apps, deleting entry points are functions that handle app uninstallation. Further complicating the detection process is when the data removal operation is present in Android framework, but the underlying logic is flawed. In these cases, we have taken the complexity of the deleting logic into consideration and quantified the possibility of each detected data residue instance.

We have evaluated ANRED against 606 Android images from several major vendors, such as Google, Samsung, Xiaomi and CyanogenMod, covering all platform versions from Gingerbread to the newest Marshmallow. ANRED detects 191 likely data residue instances on average for each image, of which 106 (55.5%) are missing data deletion logic upon app uninstallation. We have confirmed that, vendor customization is indeed a major factor in introducing new data residue instances, while the effect of version upgrade varies from vendor to vendor. To evaluate its effectiveness, we compare the analysis result from ANRED with that from the previous study [40] for the same image. Totally, 253 likely data residue instances are identified on this image by ANRED, with 205 (%81) of them labelled as highly risky. We have manually validated all 205 risky instances and uncovered 15 data residue vulnerabilities. The other 190 instances are neither app specific nor security relevant. Among those 15 identified vulnerabilities, 10 of them have been captured in the previous study, while the other 5 vulnerabilities are newly discovered, leading to data leakage and privilege escalation attacks.

Contributions The contribution of our work is three-fold:

- *New Framework*: we have designed and implemented ANRED as an automatic, scalable and source code independent framework for data residue detection on Android.
- *New Understanding*: we have evaluated ANRED against 606 images and presented an accurate and comprehensive understanding of the data residue situation across the entire Android ecosystem.
- *New Findings*: we have identified a large amount of risky data residue instances and further confirmed 5 new vulnerabilities with severe real-world damage.

Roadmap The rest of this paper is organized as follows: Section 2 explains the necessary background knowledge and presents a motivating example to drive

```
final Class SystemServer{
  public SystemServer(){
    ...
    // starting the AbcService
    ServiceManager.addService
      ("Abc", new AbcService());
    ...
  }
}
```

(a) Service Start-up

```
/* @hide */
interface IAbc{
  ...
  // one of the exposed APIs
  void setComponent(String s);
  ...
}
```

(b) Service Interface

```
Class AbcService extends SystemService
    extends BroadcastReceiver{
  @Override
  onStart(){
    publishBinderService(
        new BinderService());
  }
  Class BinderService extends IAbc.Stub{
    Public void setComponent(String s){
      new MyHandler().sendEmptyMessage();
    }
  }
  Class MyHandler extends Handler{
    @Override
    public void handleMessage(Message m){
      db.putStringForUser("Abc",s);
    }
  }
  @Override
  onReceive(){
    prepare();
    removeData();
  }
}
```

(c) Service Implementation

Fig. 1. A motivating example that shows the working flow of Android system services and origin of data residue instances.

our framework design. Section 3 breaks down the design details of each building block in ANRED. Section 4 further explains the technical details. Section 5 breaks down the evaluation results of ANRED on 606 images. Finally, Section 6 describes the related work and Section 7 makes conclusions.

2 Background

In this section, we present necessary background knowledge to facilitate the design of ANRED.

Android System Services In Android, system services provide privileged operations that can be requested by apps via permission declaration in their `AndroidManifest` files. Android system services execute in the `System_Server` process, and expose their functionalities through APIs defined in corresponding interfaces. Figure 1 demonstrates this process with a manually crafted system service `AbcService` based on real instances (`DreamManagerService` and `SpellCheckerService`). During system booting up, the `SystemServer` class adds the `AbcService` to the system service list (Figure 1(a)). By doing so, it triggers the lifecycle event `onStart()` in the `AbcService` implementation (Figure 1(c)). Upon starting, `AbcService` exposes its functionalities via an IBinder object that implements the APIs defined in the `IAbc` interface (Figure 1(b)). An interface essentially defines the protocol between two communication endpoints across the process boundary. In this case, an app can use the exposed IBinder

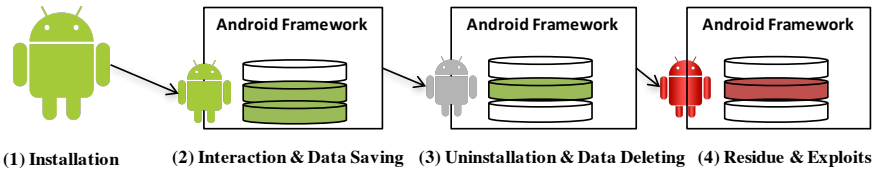


Fig. 2. Flow of data residue generation and exploits on Android

object to invoke the `setComponent()` API in the `System_Server` process by complying with the protocol defined in the `AbcService` interface. All functions defined in the interface are entry points for outside apps to trigger the saving operations within this privileged system service.

Android App Uninstallation When an app is uninstalled from the Android system, the `PackageManagerService` will remove any resources stored in the app's private directories. Then, it sends out a broadcast event to awake the uninstallation handling logic in other parts of the system. In Figure 1(c), the `AbcService` will receive such a broadcast event in its `onReceive()` function, and responds by removing related data after preparation. Apart from the generic `BroadcastReceiver` approach in Figure 1(c), there exist three other ways to get notified upon app uninstallation, including `PackageMonitor`, `DeathRecipient` and `RegisteredServicesCacheListener`. All of them are entry points for triggering data deleting logic within system services upon app uninstallation.

Android Data Residue Vulnerability Data residue exist on Android due to the mismatch between saved data and deleted data within a system service upon app uninstallation. This process usually involves four steps, as illustrated in Figure 2. It starts with the installation of a normal app (step 1). Upon user interaction, the data related to this app will be saved by certain system services within Android framework (step 2). For instance, the implementation of `AbcService` shown in Figure 1(c) asynchronizely handles the IPC invocation of `setComponent()` by saving user configuration into a database with `Abc` as the entry key. Files, databases and well-marked data structures (e.g. `Hashmap`) are normally used to store app-specific data. Later on, when the app is uninstalled, Android will try to delete related data from memory and persistent storage (step 3). In this step, the above-mentioned `Abc` entry will become residue if the app uninstallation handling logic is not in place or flawed. Having data leftover does not necessarily lead to security breaches, as long as the protection is sound. Otherwise, sensitive information will be leaked out to adversaries (step 4).

3 Design

The high-level flow of ANRED is depicted in Figure 3. To mitigate the absence of framework source code and limitation of hardware resources, ANRED depends on static analysis to directly work on Java bytecode. In this process, ANRED takes the entire Android image as input, extracts the framework's and preloaded

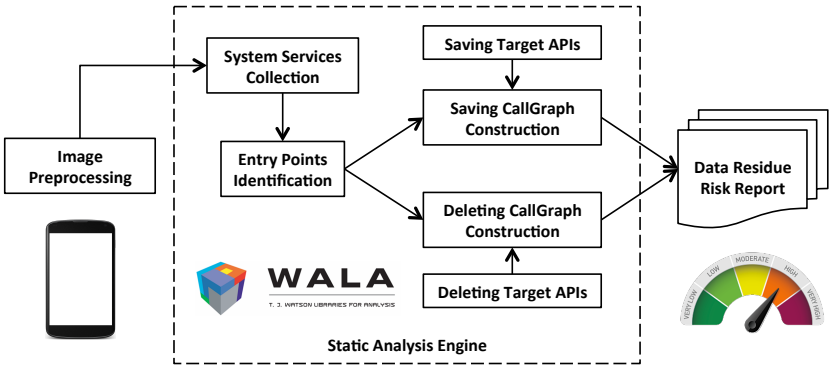


Fig. 3. Overview of ANRED Design

apps' bytecode, and collects candidate system services. As the data residue vulnerability occurs at the level of system services, ANRED's static analysis logic also resides at the same level. For each system service, we are interested in two types of operations inside: saving operation and deleting operation. Mixing both operations on the same call graph will make them indistinguishable at the end. Thus, ANRED generates two call graphs for each system service: a saving graph and a deleting graph. The saving graph captures the data that will be saved inside the system service, while the deleting graph indicates what data will be removed when handling app uninstallation. Based on that, data residue instances are inferred from mismatches between the deleting data set and the saving data set. To cover instances that are caused by flawed deleting logic, ANRED also takes into consideration the complexity of the deleting logic to quantify the underlying risks. We explain the design challenges of each block in following sections.

3.1 Image Preprocessing

Given an Android image, ANRED extracts the Android framework code and its preloaded apps. Since different vendors or different versions of Android pack the code in different formats (i.e., `apk`, `dex`, `odex` to `oat`), ANRED employs several utilities to handle each format gracefully. We employ `dex2jar` utility to convert `dex` and `apk` files into `jar`s, suitable for standard static analysis platforms. Additionally, we use `apktool` to retrieve the `AndroidManifest` configuration file from each app. We use `baksmali` and `smali` to convert `odex` code to `dex` files. Images that contain `oat` files or target Android Lollipop require special handling with `dextra` [7] and `deodex-lollipop` [13].

3.2 System Service Collection

To detect data residue instances, we need to identify all system services from a compiled image. Such list is challenging to obtain, as the registration place of each system service varies greatly among images. Our key observation is that

system services registration APIs are more stable across Android customization and version upgrade. The service example shown in Figure 1 demonstrates the two most representative APIs, `addService()` and `publishBinderService()`, for publishing a system service. Thus, to retrieve the system service list, ANRED first collects functions that invoke `addService()` or `publishBinderService()` and marks them as entry points. For each entry, ANRED constructs a call graph. By traversing through the call graph, not only can we pinpoint the places where system services are registered, but also resolve the service class and exposed interface class. For system services that are within preloaded apps, the above process is simplified by directly searching the `AndroidManifest` file for services that are accessible from the outside world. In both cases, we further filter out unnecessary `jar`s and generate input specifications for the static analysis platform.

3.3 Entry Point Identification

Static analysis on Android platform demands the construction of a precise call graph. For that purpose, an accurate and complete list of entry point functions needs to be provided. In normal Java programs, the entry point is the `main` function. However, the widely adopted callback mechanism in Android framework complicates this task. In ANRED, we consider entry points as asynchronized invocations where their callees are present in the analysis scope, but their callers are not. As shown in Figure 1(b), the `AbcService` specifies a list of exposed APIs in the interface file, namely `AIDL`, for apps to interact with. Naturally, all public `AIDL` functions become entry points. However, a great amount of other asynchronized invocation patterns are present within different system services. One representative example is the `onClick` function inside the `onClickListener` interface. In this case, the system service provides the implementation of `onClick` function (callee), while the caller (user event) triggers the invocation. Clearly, the caller does not exist in the analysis scope, and thus, ANRED considers the callee function, `onClick`, as an entry point.

To collect all entry points, we take a similar approach as in EdgeMiner [22] by searching all internal classes within each system service for interfaces and abstract classes. The reason behind is that, both interfaces and abstract classes rely on other parties to provide the actual implementation, indicating the absence of a caller. Different from EdgeMiner, we exclude functions within `Handler`, `Thread`, `AsyncTask` and `ServiceConnection` classes, since they are asynchronized invocations where the caller and callee are both present in the analysis scope, as illustrated in Figure 4(a). Such cases lead to broken links in the constructed call graph, which we will handle differently in Section 3.4. After identifying all entry points, we further divide them into saving entry points and deleting entry points for the construction of the saving call graph and the deleting call graph, respectively. We consider all the APIs that handle app uninstallation as deleting entry points, and the rest as saving entry points. Specifically, ANRED includes the following four classes as the source for delet-

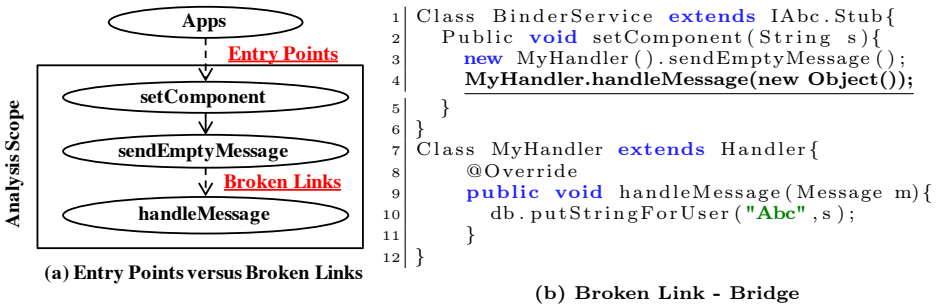


Fig. 4. Connecting Broken Links in ANRED via Bytecode Rewriting

ing entry points: `PackageMonitor`, `BroadcastReceiver`, `DeathRecipient` and `RegisteredServicesCacheListener`.

3.4 Call Graph Construction

We choose WALA [16] as the static analysis platform for call graph construction, due to its popularity and strength in data flow analysis, which is our main concern in the data residue detection process. The analysis stays at the Android framework level, but inherits all challenges (i.e., broken links) from the app level. Take `MyHandler` class in Figure 1(c) as an example. Its execution is conducted with the help of a `Message` queue. The caller side (`setComponent()`) puts a message on the queue, and waits until it is consumed by the callee side (`handleMessage()`). Even though both the caller and callee are present in the analysis scope, the connection is missing. Those broken links greatly affect the code coverage in static analysis, and thus, hinder ANRED's performance. Existing solutions [33,17,35] model the behavior of caller and callee functions, and add edges in the constructed call graph to connect them. Such bridges mitigate the reachability issues (control flow) on the call graph, but do not explicitly catenate the data flow.

To this end, we would like to connect broken links at the bytecode level, independent from the static analysis platform. The output will be a fixed `jar` file with all broken links connected. Benefit from this, researchers can apply it directly for different purposes with their own choices of static analysis platforms. More specifically, we propose to rewrite the bytecode of collected system services to connect the broken links. ANRED performs bytecode rewriting using the `shrike` utility in WALA, which is capable of looping through all instructions from the given bytecode. Ideally, once the invocation that causes broken links is found, we rewrite the instruction to bridge the connection. The real challenges lie in patching a diversity of invocations with a generic scheme. We use the `MyHandler` example in Figure 1(b) to explain the details.

First of all, there are various functions for delivering messages with different arguments. In Java, replacing the current invocation to `handleMessage()` requires loading the correct handler instance and constructing the proper arguments. An easier and more stable approach is to add instructions at the

end, instead of replacing existing ones. In the `MyHandler` case, ANRED adds the invocation to `handleMessage()` at the end of `setComponent()` function, as demonstrated in Figure 4(b). However, adding the instruction still requires the creation of corresponding handler instance and argument instances. To be more specific, in order to invoke `handleMessage()`, an instance of the `MyHandler` class is needed, as well as a `Messsage` instance if the argument is not empty. It is quite challenging to statically accommodate all situations. Our design choice is to invoke the `handleMessage()` API as a static function and provide generic objects as necessary arguments. Although the generated `jar` may not execute properly, it is not an issue since static analysis does not check for conflicting method descriptors. To handle the case where the handler class type is inherited, ANRED locates the outermost class, and searches all internal classes for the ones that extend the base handler class or implement the callback interface. Theoretically, such approach will generate false mappings when multiple handler classes are present within the same outer class, however, our observation is that, inside each Android system service, there is usually only one handler class in use. Eventually, the bridge will be connected as in Figure 4(b).

3.5 Target APIs Harvest

To detect mismatches between storage operations and cleanup logic, a complete list of saving and deleting API pairs is necessary. For instance, the `AbcService` shown in Figure 1 uses `db.putStringForUser()` to save entries into a database. To check the existence of data residue, we would like to know the usage of corresponding deleting API (in this case, identical to the saving API) in handling app uninstallation. The manual identification of such API pairs is a tedious work, and lacks the guarantee on code coverage. Instead, ANRED applies heuristics generalized from manual inspection to harvest API pairs automatically. In particular, we categorize all saving/deleting API pairs into `SQLiteDatabase`, `SharedPreference`, `Settings`, `Java_Util` and `XML`. For each category, we then generalize heuristics at the level of package, class and function. At the top level, each category corresponds to a Java package, and all classes in it will be the source of API pairs. At the second level, we exclude classes that are irrelevant to storage operations. To illustrate, although thousands of classes exist in the `Java_Util` category, we can safely remove those related to `Exception`, `Concurrency` or `Thread`. At the last stage, both saving and deleting APIs follow strict naming schemes, like `put*()`/`remove*()` in `Java_Util` classes and `create*()`/`delete*()` in `File` classes. Such naming schemes allow a further filtering on the API descriptors directly. With these three levels of heuristics, ANRED is able to reliably exclude 95% of total API candidates.

3.6 Risk Evaluation

Following the steps above, ANRED constructs two call graphs, i.e., saving graph and deleting graph, for each collected system service. Guided by the storage API pairs, we can further gather all saving and deleting operations via traversing

```
<image carrier="" manufacturer="" model="" name="" region="" version="">
  ......
  <service category="java_util" finished="" ibinder="" name=""
    output_time_cost="" size="" type="">
    ......
    <residue complexity="1000" deletingInstructions="" name=""
      savingInstructions=""/>
    ......
  </service>
  ......
  <serviceDetectionCost></serviceDetectionCost>
  <rewritingCost></rewritingCost>
  <residueDetectionCost></residueDetectionCost>
</image>
```

Fig. 5. ANRED Final Report Template

through corresponding call graphs. Ideally, only the mismatches indicate data residue instances. However, as shown in Figure 1 and previous work [40], the data cleanup logic may be flawed. As a result, ANRED quantifies the likelihood of each data residue instance with respect to the complexity of data deleting logic. A variant [11] of the standard Cyclomatic complexity [5] is used in ANRED against each deleting function. When the deleting logic is missing, we assign the largest complexity value, indicating the highest risk for this instance to be a real data residue vulnerability. At the image level, ANRED further aggregates all data residue instances together into a well-formatted XML report as shown in Figure 5.

4 Implementation

The implementation of ANRED consists of around 7,500 **L**ines **O**f Java **C**ode (LOC) and 5,000 lines of Python code with comments included. We further break down the entire code base into four modules: `Jar` extraction, `Jar` decompilation, residue detection and result analysis. The code composition and library dependency for individual module is shown in Table 4.1. To handle different situations, our code base contains functions with small deviations. For instance, we have slightly different Python scripts to extract `jar`s from Nexus images and Samsung images. We emphasize that, those functions are counted twice in the presented statistics, i.e., the count of effective LOC will be smaller. The code base of ANRED, as well as our image collection and evaluation results from Section 5, are publicly available on [2].

4.1 Bridging Broken Links

ANRED leverages WALA's `shrike` utility to bridge broken links caused by the `Handler`, `Thread`, `AsyncTask` and `ServiceConnection` classes in Android. We fit `jar` files into `shrike` individually, and traverse through each embedded class, functions in each class and bytecode instructions in each function. For each instruction, we check it against the candidate invocations that can cause broken links. In the `Handler` case, such invocations include `send*Message()` and

Module	#LOC	Language	Dependency
`Jar` Extraction	1438	Python	`ext4utils` [8]
`Jar` Decompilation	2638	Python	`dex2jar` [6], `apktool` [3], `dextra` [7] `deodex-lollipop` [13], `baksmali/smali` [15]
Residue Detection	7568	Java	WALA [16]
Result Analysis	857	Python	-

Table 1: ANRED Code Base Breakdown

`sendToTarget()` APIs, while in the `AsyncTask` case, `execute*()` typed APIs will be our candidates. For functions that do not contain candidate invocations, they stay untouched in the fixed `jar` file.

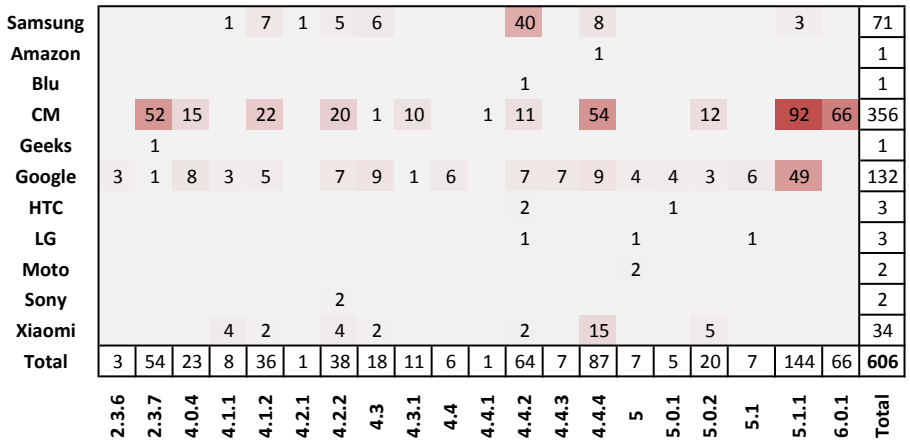
For each target instruction identified, ANRED records its position within the method body and inserts a static invocation to the callee function. Such a function invocation requires totally four types of instructions. `NewInstruction`s are used to create generic java objects to be fed as function arguments, followed by `StoreInstruction`s and `LoadInstruction`s, which store and load the generated variables, respectively. Eventually, an `InvocationInstruction` initiates the function call. In `shrike`, each `InvocationInstruction` consists of four segments: function descriptor, object type, function name and dispatch mode. As explained in Section 3.4, ANRED dispatches this instruction as a static invocation, and searches from the outermost class for the object type. While function name is self-explanatory, an accurate function descriptor could be challenging to obtain. In most cases, we can manually craft the descriptor string according to the Android documentation. However, in the `AsyncTask` case, all callee functions contain `Params...`, known as Java Varargs [12] that takes an arbitrary number of values with arbitrary types upon invocation. ANRED overcomes the challenge by directly resolving the function descriptor from the object class implementation. Specifically, we search the entire class hierarchy for the class that implements the object type, and then obtain descriptor for each function inside.

4.2 Building Class Hierarchy and Call Graph

ANRED totally constructs three call graphs: one for system service collection, one for the saving logic analysis and the last one for the deleting logic analysis. Since these three graphs serve for different purposes, they demand different level of precision. In particular, we use RTA [18,19] algorithm in constructing the call graph for system service collection, as it is relatively simple and fast. All we need from this call graph is the identification of certain APIs. In contrast, we employ 0-CFA [32,36] algorithm to construct call graphs for the saving and deleting logic analysis, since they both require an accurate control flow and data flow dependency.

4.3 Discussion

There are a few limitations resulting from ANRED's implementation. First of all, WALA can only perform static analysis on Java bytecode, excluding native



	2.3.6	2.3.7	4.0.4	4.1.1	4.1.2	4.2.1	4.2.2	4.3	4.3.1	4.4	4.4.1	4.4.2	4.4.3	4.4.4	5	5.0.1	5.0.2	5.1	5.1.1	6.0.1	Total
Samsung				1	7	1	5	6				40		8					3		71
Amazon														1							1
Blu												1									1
CM		52	15		22		20	1	10		1	11		54			12		92	66	356
Geeks		1																			1
Google	3	1	8	3	5		7	9	1	6		7	7	9	4	4	3	6	49		132
HTC												2				1					3
LG												1			1			1			3
Moto															2						2
Sony							2														2
Xiaomi				4	2		4	2				2		15			5				34
Total	3	54	23	8	36	1	38	18	11	6	1	64	7	87	7	5	20	7	144	66	**606**

Fig. 6. Distribution of Collected Android Images for ANRED Evaluation

Android system services from our study. From the previous study [40], the percentage of native services is relatively small. Secondly, the complexity of static analysis has long been a concern. In the data residue detection process, we make the best effort to guarantee the code coverage, but there will be inaccuracy introduced in various stages, such as in the code decompilation and broken link re-connection. At last, human effort is still needed to validate possible data residue instances reported by ANRED. However, as we show in Section 5.2, the manual involvement has been greatly reduced with the help of ANRED, yet more data residue instances are captured.

5 Evaluation

We have collected a total of 606 Android images from various sources [10,9,14,4,1]. Figure 5 depicts the distribution of our image collection with respect to the diversity of vendors as well as coverage of Android versions. In particular, we have covered all major vendors, such as Samsung, HTC, CyanogenMod (CM, in short), Google, Xiaomi, etc, and all major Android versions from Gingerbread up to the newest Marshmallow. Throughout the experiment, we consider the OS provider, instead of the device manufacturer, as the real vendor of the image. For instance, we have labeled Google as the vendor for all Nexus images. CM provides Android OS for various device models from different manufacturers, and thus, it becomes the vendor for all of them. We have conducted our experiments on `Dell PowerEdge T620` with `Intel Xeon CPU E5-2660 v2 @ 2.20GHz` running `Ubuntu 14.04.1 LTS`. The `Jar` extraction and decompilation stages took around one week to finish for all images, and the processing outputs are available on [2]. After that, we ran ANRED against each image with 16G heap memory size allocated for the Java Virtual Machine (JVM) and with the default timeout value (60s) for each system service.

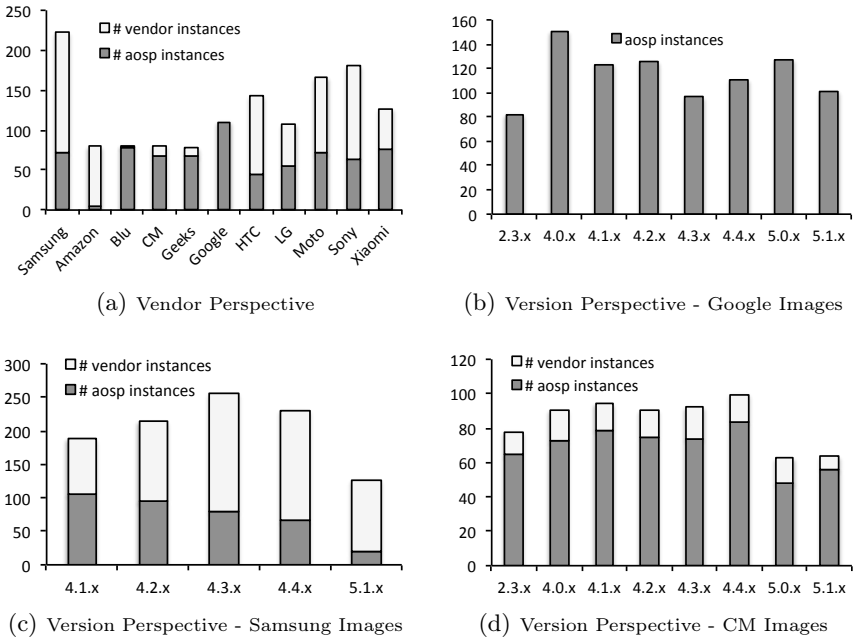
5.1 Panorama of Android Data Residue

Overview On average, we have identified 191 likely data residue instances on each image, with 106 (55.5%) of them being labeled as missing data removal logic. Not all of them will necessarily lead to the data residue vulnerability, depending on whether the data leftover is security-critical and exploitable by adversaries. To separate the data residue instances introduced by vendor customization and the ones inherited from the AOSP code base, we consider Google images as the reference. Given one data residue instance on a vendor image, we label it as AOSP instance if such instance has also been identified on Google images with the same version number. Otherwise, we consider it as vendor introduced instance. For images that do not have a corresponding Google image in our collection, we exclude them from our data analysis. In following sections, we further break down the data residue situation on Android from four perspectives: vendor, version, service category and residue type.

Vendor-wise View In our image collection, the version distribution varies greatly from vendor to vendor. Thus, a direct comparison of the average residue count for each vendor will not accurately reflect the effect of vendor customization on the data residue vulnerability. To remove the version bias, we compare the percentage of vendor introduced data residue instances instead. The result is shown in Figure 7(a). We have observed that, vendor customization is indeed a big factor contributing to the data residue vulnerability. In particular, vendors like Samsung, Amazon, HTC, and Sony are responsible for more than 65% of data residue instances identified on their images. One extreme case is the Amazon image running 4.4.4. Our analysis result indicates that, 95% of data residue instances identified on this image are due to the heavy customization. Even for vendors like Moto, LG and Xiaomi, the percentage of vendor introduced data residue instances is higher than 40%. In comparison, Blu, Geeks and CM make fewer changes to the Android OS, and thus, only introduce a small portion of data residue instances to their images.

Version-wise View Apart from vendor customization, we would like to further evaluate the effect of Android's frequent version upgrade on the data residue vulnerability. The data residue trend across different Android versions is meaningful only if all selected images are belonging to the same vendor. We have chosen three vendors, i.e., Google, Samsung and CM, since our image collection contains different versions of these vendors. The results are depicted in Figure 7(b), Figure 7(c) and Figure 7(d), respectively. As Figure 7(b) shows, the average count of AOSP data residue instances fluctuates around 100 across version upgrade. We have observed that, there is always an increment when new Android branches, like Ice Cream Sandwich (4.0.x) and Lollipop (5.0.x), are released. We suspect that, new branches tend to come with new features, and thus, introduce new system services with possible data residue instances.

The version trends on Samsung images and CM images have quite different characteristics, as shown in Figure 7(c) and Figure 7(d). For Samsung, as the



(a) Vendor Perspective

(b) Version Perspective - Google Images

(c) Version Perspective - Samsung Images

(d) Version Perspective - CM Images

Fig. 7. Effect of Vendor Customization and Version Upgrade on Data Residue

version increases, we have observed a steady decrease of data residue instances inherited from AOSP code base. In comparison, the percentage of vendor introduced ones rises from 44% to 84%. The result indicates that, Samsung's customization on Android OS has grown heavier as Android evolves, resulting in fewer similarities with AOSP images. Thus, the data residue situation on upcoming Samsung images will be mostly determined by the level of its own customization. On the other hand, the count of data residue instances introduced by CM remains the same at around 15 from Gingerbread to Lollipop. Clearly, the customization performed by CM is quite consistent across version upgrade. In this case, the data residue situation on upcoming CM images depends highly on the corresponding AOSP version release.

Service-wise View ANRED has considered two categories of system services in the static analysis stage, i.e., preloaded app services and framework services. We have separated their effects and concluded that the majority (65%) of identified data residue instances are from framework services. This is consistent with the manual analysis result from previous work [40], where only 2 (download and print) out of 12 instances are within preloaded apps. However, we argue that, the framework developers and preloaded app developers should work together to remove all the data residue instances.

Residue-wise View As mentioned in Section 3.5, we have included five types of data residue instances in our analysis, i.e., `SQLiteDatabase`, `SharedPreference`,

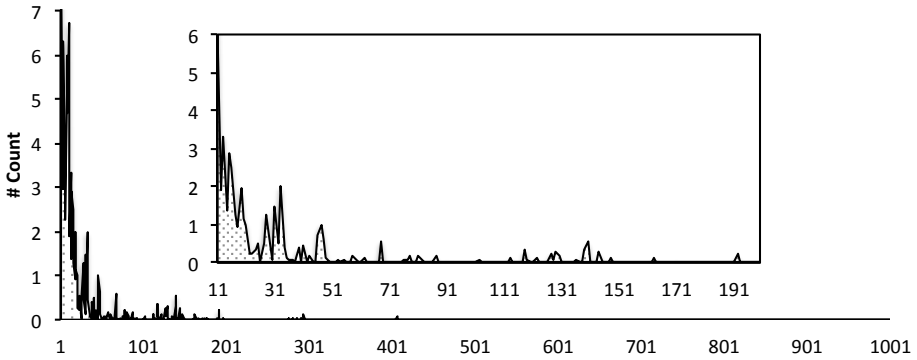


Fig. 8. Data Residue Instance Distribution based on the Complexity of Deleting Logic

`Settings`, `Java_Util` and `XML`. With a total of 116K data residue instances identified on all images from our collection, 73% of them are in memory data structures (`Java_Util`) and configuration entries (`Settings`). In comparison, previous work [40] has also identified 8 (2 capability instances, 5 Settings instances and 1 permission instance) out of 12 instances belonging to those two types. It is worth mentioning that, among all data residue instances, 1,629 (1.4%) unique ones are found. In this process, we have used a combination of service name, residue type and entry name as the key to remove duplicates. Moreover, only 312 (19%) of the unique ones are from AOSP images, while the rest (81%) are all introduced by vendor customization.

Risk Quantification In our analysis, we have identified an average of 85 instances on each image that have data deletion operations in place upon app uninstallation. However, the complexity of each one's deleting logic varies greatly. We further calculate the average count of data residue instances for each complexity value. The overall distribution is shown in Figure 8. A total of 42 (50%) instances have deleting complexity value less than 10, and thus, should be considered as safe. However, the deleting logic of the other half is overcomplicated and may lead to security flaws. To understand how bad the situation is, we zoom in to the region with complexity value between 11 and 200, as shown in Figure 8. Surprisingly, a significant portion of functions that handle data removal upon app uninstallation even have a complexity value larger than 30. Based on our analysis, we suggest system service developers to follow clear guidelines to remove app data upon its uninstallation.

5.2 ANRED Effectiveness

To demonstrate the effectiveness of ANRED, we have evaluated it against AOSP 5.0.1 image, which was manually examined in previous work [40]. We use their results as the basis for comparison. The analysis takes 11 minutes in ANRED with the default 60s timeout value, and covers a total of 133 system services. Among them, 123 (%92.5) services are finished within timeout limit. Totally,

Instances	Category	Damage	Frequency
backup_transport	Settings	data leakage	359 (%59)
app restrictions	XML	privilege escalation	396 (%65)
notification_policy	XML	privilege escalation	200 (%33)
app-ops	XML	-	-
media_store	Content Provider	-	-

Table 2: New Data Residue Instances Identified by ANRED on AOSP 5.0.1

253 likely data residue instances are identified on this image. More importantly, 205 (%81) of them have a complexity value larger than 10, which are considered as highly risky [11]. We have further validated all 205 risky instances manually. This process was completed by one single Android security analyst within a day, in comparison with 6 person-month in the previous work [40]. It is made possible because of ANRED's detailed report, which not only presents each likely data residue instance, but also pinpoints its saving instruction and risky deleting instruction. We envision that, Android vendors will utilize ANRED in a similar manner to detect and remove data residue instances from their images before the final release.

In addition to the significantly reduced human involvement, we are able to capture 10 out of 12 real data residue instances presented in [40]. The only two missing instances are `Keystore` and `Download`, which have been resolved on Android Lollipop. The previous work reproduces the vulnerability on KitKat and prior versions. One representative data residue vulnerability is on Android printing framework. ANRED actually identifies two related instances: one from the `XML` category corresponding to the print record residue and the other one from the `File` category mapping to the print content residue. Other than that, ANRED has detected five new data residue instances. Three of them are actually exploitable, leading to data leakage and privilege escalation attacks. We have further measured the frequency of those three vulnerabilities within our image collection. As summarized in Table 2, these vulnerabilities are pervasive, with a total of 571 (%94) unique images containing at least one of them.

Backup Mis-transport Android backup framework helps to copy a user's persistent app data to remote cloud storage. Internally, there is a system service called `BackupManagerService`, which forwards all requests from registered client apps to the current enabled backup service. Android backup service serves as a backup transport between the device and the remote storage. Once enabled, the package name and service name will be jointly saved into Android Settings with entry name `backup_transport`. However, this configuration entry will remain effective even after the referring app has been uninstalled. As a result, an adversary can impersonate the uninstalled app and mis-transport the user's private data to malicious servers. This is not a big concern right now, as only vendor issued backup services can be installed on the device. But in the future, if Android decides to open this feature to 3rd-party apps, such a residue instance demands great attention.

App No-restrictions The multi-user feature on Android has been a viable approach for creating another restricted environment on the same device. On tablet devices, the restricted profile feature has been used mainly for parental control, while on Android phones, such restricted environment, namely managed profile, is favored by enterprise companies to control the environment where company-specific apps and data are running. In both cases, an administrator app is responsible for specifying the restrictions of individual apps. Such configurations will be saved into `user_id.xml` by a system service, `UserManagerService`, with attribute `restrictions`. However, these access control policies will not be removed, even after the targeted app has been uninstalled. In return, an adversary can inherit all privilege from the uninstalled app.

Notification Flooding Android supports three levels of notification restrictions on each app, i.e., allow, block and priority. By default, any notifications from the app will be allowed. However, a user can flip the configuration from the Settings app or choose to show priority ones only. The corresponding specifications will be saved in `notification_policy.xml` by a system service, namely `NotificationManagerService`, with attribute `notification-policy`. We have found that, when an app is removed from the device, its notification policy configured by the user will be left over. Thus, an adversary can impersonate this app and flood the user with annoying notifications.

Two Harmless Residue Instances Another two data residue instances have been found in our manual verification, but with limited implications. For instance, Android system service, `AppOpsService`, saves restrictions on app operations into `appops.xml`, but fails to immediately clean them up upon app uninstallation. Actually, the cleanup task is scheduled periodically for every 30 minutes. The second instance is related to the `mediastore` provider on Android, which contains meta data for all available media on both internal and external storage. An app can send a request to `MediaScannerService` for adding a media file into the `mediastore`. When the app is uninstalled, although the file is deleted from the file system, its meta data remains in the `mediastore` until Android scans the system for new media, which typically happens when the system first boots up or can be called explicitly from apps. Noting that the meta data includes a URI referring to the actual file, we further evaluated it against the capability intruding attack as in the Clipboard residue case [40]. However, as it turns out, URIs saved in `mediastore` do not possess any capabilities and immune from this attack. In both residue cases, although the attacking time window is quite small, we argue that, a timely data cleanup approach, like `BroadcastReceiver`, should be employed to provide better security guarantee.

5.3 ANRED Performance

Time Consumption On average, it takes ANRED 43.5 minutes to analyze one image, with 1.6 minutes (3.6%) on system service extraction, 6.5 minutes (15%) on `jar` patching and 35.4 minutes (81.4%) on static analysis.

System Service Analysis - Success Rate With an average of 205 system services being analyzed for each image in ANRED, 82.67% of them are finished in our experiment. In particular, framework services (86.53%) have a higher success rate than preloaded app services (80.63%). A closer look at the `jar` decompilation results reveals that, certain preloaded apps have applied code obfuscation techniques in the final packaging process. Another factor affecting the success rate is the timeout value. In our experiment, we have used the default 60s timeout value. With more time allowed for each service, the success rate of ANRED's system service analysis can be improved.

Broken Link Patching Stats In our experiment, ANRED totally patched around 3 million broken links. In particular, `Thread` and `Handler` are the two most commonly used classes in Android framework and preloaded apps. They account for a total of 92% of the overall patched broken links. As mentioned in Section 3.4, ANRED patches broken links with static invocations inserted at the end, while the object type is resolved based on a heuristic. We have further measured the accurate connection rate of ANRED for different categories of broken links. The result demonstrates the reliability of ANRED with 86% accurate connection rate on average. The patching accuracy for `AsyncTask` and `ServiceConnection` is slightly lower, indicating that multiple implementations of them may coexist in certain system services. Even for those cases, ANRED is still able to patch broken links conservatively without causing any exceptions.

6 Related Work

Android Vulnerability Exploration Earlier and recent research work have identified several worrisome security vulnerabilities in the Android apps. Prominent examples include the re-delegation problem [28], content providers leak and pollution [31], crypto-misuse in Android apps [25] and vulnerabilities in the Android's WebView component [34]. Other studies revealed security risks in the Android system itself. PileUp [37] uncovers a flaw in the `PackagemMangerService` that targets system update and allows a malicious app installed on a lower version Android and claiming specific capabilities to actually acquire these capabilities after system update. Two other works [27,39] reveal exploits on the `ClipboardService` that enables an unprivileged attacker to gain access to important private data. Our work aims to automatically detect the data residue vulnerability [40] scattering through a wider range of system services.

Static Analysis Several systems have been accordingly proposed to mitigate these discovered vulnerabilities. A great deal of previous studies aims to mitigate the confused deputy problem and permission leaks by either checking IPC call chains or by monitoring the communication between apps at run time [20,21,24,28,29]. To better understand the scope of the attacks discovered, several other static analysis frameworks have been proposed, including ComDroid [23], CHEX [33], FlowDroid [17], DroidSafe [30], Epicc [35], etc. These tools, however, analyze Android apps and cannot handle framework-level code,

thus, cannot be adopted to automatically discover the data residue vulnerability in system services within the Android framework. Our efforts in this work aim to fill this gap by introducing ANRED, which automatically examines each system service's bytecode to accurately uncover any data residue instances.

Dynamic Analysis Few Other dynamic approaches have been proposed to detect and defeat the discovered vulnerabilities. A prominent example in the literature is TaintDroid [26], a framework that allows to perform taint analysis on Android to track data flow across apps and the OS. Another prominent tool is DroidScope [38], which runs the whole Android platform on an emulator to reconstruct both Dalvik and OS level semantics. Dynamic solutions could possibly enable us to uncover data residue vulnerabilities at app uninstallation time. However, the triggering conditions leading to the data residue problem (such as device reboot, app installation and uninstallation) are difficult to fully emulate using dynamic approaches. Besides, even if taint tracking might look like a possible solution for detecting the data residue problem, the process is quite complicated because the data creation points might not necessarily appear in apps. ANRED aims to overcome these challenges that cannot be solved through a dynamic solution, through a purely static solution that performs the detection of data residue problems automatically and efficiently.

7 Conclusion

In this project, we propose ANRED to automatically detect data residue instances on a large scale of Android images with minimal human involvement. The evaluation results against 606 images have again brought questions over the extensive vendor customization and frequent version upgrade on Android. We hope that, vendors can use ANRED to check their images against the data residue vulnerability before shipping with new devices. More importantly, Google should take the lead to provide a clear guideline in reacting to the event of app uninstallation. Additional efforts are also required from the research committee to propose a runtime solution to eliminate the data residue vulnerability.

ACKNOWLEDGMENT

We greatly appreciate the insightful comments and constructive feedback from the anonymous reviewers. This project was supported in part by the NSF grant 1318814. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Android Revolution. `http://goo.gl/MVigfq`.
2. ANRED: Android Residue Detection Framework. `https://goo.gl/Q0d5qH`.

3. Apktool: A tool for reverse engineering Android apk files. http://goo.gl/LdB4V7.
4. Cyanogenmod Downloads. http://download.cyanogenmod.org/.
5. Cyclomatic complexity. https://goo.gl/1VqYUj.
6. dex2jar. https://goo.gl/5fzsd5.
7. dextra - A tool for DEX and OAT dumping, decompilation, and fuzzing. http://goo.gl/NPG0Kz.
8. ext4_utils. https://goo.gl/1nyYfM.
9. Factory Images for Nexus Devices. https://goo.gl/i0RJnN.
10. Huawei ROMs. http://goo.gl/dYPTE5.
11. Java: Computing Cyclomatic Complexity. http://goo.gl/tduqlP.
12. Java Varargs. http://goo.gl/TEMrjk.
13. Lollipop deodexing. https://goo.gl/uw2KmR.
14. Official Oxygen OS ROMs and OTA updates. https://goo.gl/cBTF1w.
15. smali and baksmali. https://goo.gl/JS7Mgw.
16. WALA. http://wala.sourceforge.net/.
17. 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 *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, 2014.
18. D. F. Bacon. Fast and effective optimization of statically typed object-oriented. Technical report, Berkeley, CA, USA, 1998.
19. D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.
20. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technische Universität Darmstadt, Technical Report TR-2011-04, 2011.
21. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. NDSS, 2012.
22. 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 *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.
23. 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*, MobiSys '11, New York, NY, USA, 2011. ACM.
24. M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.
25. M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013.
26. 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

27. S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, you, get off of my clipboard. In *proceeding of 17th International Conference on Financial Cryptography and Data Security*, 2013.
28. A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, pages 22–37, 2011.
29. E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing androids permission system. 17th European Symposium on Research in Computer Security, 2012.
30. 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.
31. M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.
32. D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM.
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*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.
34. T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. ACSAC, 2011.
35. D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.
36. O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.
37. L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 393–408, Washington, DC, USA, 2014. IEEE Computer Society.
38. L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.
39. X. Zhang and W. Du. Attacks on android clipboard. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2014.
40. X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.