NativeGuard: Protecting Android Applications from Third-Party Native Libraries

Mengtao Sun
Lehigh University
27 Memorial Drive West
Bethlehem, PA 18015, United States
mengtao.sun@lehigh.edu

Gang Tan
Lehigh University
27 Memorial Drive West
Bethlehem, PA 18015, United States
gtan@cse.lehigh.edu

ABSTRACT

Android applications often include third-party libraries written in native code. However, current native components are not well managed by Android's security architecture. We present NativeGuard, a security framework that isolates native libraries from other components in Android applications. Leveraging the process-based protection in Android, NativeGuard isolates native libraries of an Android application into a second application where unnecessary privileges are eliminated. NativeGuard requires neither modifications to Android nor access to the source code of an application. It addresses multiple technical issues to support various interfaces that Android provides to the native world. Experimental results demonstrate that our framework works well with a set of real-world applications, and incurs only modest overhead on benchmark programs.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; D.2.12 [**Software**]: Software Engineering—*Interoperability*

Keywords

Android; Java Native Interface; Privilege isolation

1. INTRODUCTION

Smartphones have been increasingly popular and widely adopted around the globe. Evolution and development of smartphones benefit from a large number of applications available in online stores such as Apple's App Store for iOS and Google's Google Play for Android. Apple adopts a confidential vetting process on all applications submitted to the App Store. On the contrary, Google has built a security framework for Android that in general keeps its ecosystem secure and at the same time open and flexible. Each Android application is assigned a unique Linux user ID and runs in a separate process. Furthermore, privileges of applications are controlled by a dedicated permission system.

Android applications are mostly written in Java. Similar to desktop Java programs, Android supports the Java Native Interface (JNI) and allows applications to incorporate native libraries. It is well known that in a conventional setting, native code defeats Java's security, as native code is not covered by Java's security model and has access to the entire address space. In Android, on the other hand, the story is different. First, previous studies (e.g., [6]) have shown low utilization of native code in Android applications. Second, process isolation, as the basis of security in Android, regulates all application components above the kernel, regardless of the programming language used in development. Therefore, it is generally believed that "native code is not pervasive", and that "native code is as secure as Java" in Android applications. Hence neither researchers in academia nor engineers in industry have paid much attention to native-code security in Android.

However, we believe Android's defense against native threats is not as strong as it appears. Our survey (detailed in Section 3) over the most popular Android applications shows that native libraries are used pervasively in popular Android applications. Nowadays, applications tend to provide diverse functionalities in order to take full advantage of cutting-edge hardware in modern Android phones. Tasks like 3D rendering and audio/video encoding have become extremely common in popular applications, where third-party native libraries are much more likely to appear. To make matters worse, these third-party native libraries enjoy all permissions the user grants to the whole application, which is often unnecessary and violates the principle of least privilege.

Given the prevalence of third-party native libraries in popular Android applications, the question is how to limit native libraries' privileges so that the damage caused by malicious or buggy third-party native libraries can be controlled. A natural idea is to design a framework in which native libraries are privilege separated from the rest of the application. Native libraries can have their own set of permissions, different from the permissions used in Java code.

For this purpose, we have designed NativeGuard, a framework that utilizes Android's process isolation to sandbox native libraries of an application. Generally speaking, NativeGuard improves Android's security in the following two aspects. First, NativeGuard separates native libraries contained within an Android application to another standalone application, where native code does not have full access to the entire application address space and the interaction be-

WiSec'14, July 23–25, 2014, Oxford, UK.

ACM 978-1-4503-2972-9/14/07 ...$15.00.
http://dx.doi.org/10.1145/2627393.2627396.

tween native and Java code is fulfilled via Android's interprocess communication (IPC) mechanism. Second, the native-library application is no longer granted all permissions the original application possesses; therefore, there would be much less damage if those libraries were exploited.

The main contributions of this paper are as follows:

- Our survey shows the prevalence of native libraries in popular Android applications, contrary to previous beliefs.
- To the best of our knowledge, NativeGuard is the first work on Android focusing on security threats of native libraries. We addressed multiple technical challenges in NativeGuard, involving the support of JNI function calls and the accommodation of Android's Native Development Kit (NDK) API. The framework takes advantage of the existing Android security architecture, does not need special hardware or system support, and can be easily deployed to the current Android system without much pain.
- We evaluated the prototype of NativeGuard over both the most prevalent Android applications today and industrial-strength benchmark programs, which fully demonstrated its practicality and efficiency.

We stress that, in NativeGuard's threat model, an application's Java components are trusted, while the application's native libaries are untrusted. We are mainly concerned with the scenario in which an Android developer incorporates a third-party native library into her application, treats it as a black box, and blindly assigns all permissions of the application to the library. Through NativeGuard, the developer can assign a much smaller permission set to the native libraries and as a result security is improved. Clearly, the Java components of an application could be malicious as well. But malicious applications are a well recognized threat and defenses have been proposed in many other studies. We instead focus on the security of native libraries. The threat model of NativeGuard will be detailed later in the paper.

The rest of the paper is organized as follows. We first introduce the background of Android security and the JNI interface in Section 2. In Section 3, we show the prevalence of native libraries in popular applications and explain how Android controls their security. Section 4 is an overview of NativeGuard, followed by Section 5, where we describe the isolation achieved by the framework, as well as several technical challenges and our solutions. We present in Section 6 our prototype implementation and experimental evaluation. In the end, we discuss related work, future work and conclude.

2. BACKGROUND: JNI AND ANDROID

In this section, we present a high-level overview of the Android architecture and its deployed security mechanisms. We also introduce the necessary background about the Java Native Interface.

2.1 Android Security Overview

Android is a software framework designed for mobile devices. It is built upon an adapted Linux kernel and supports Applications written in Java. Similar to desktop Java programs, they are also allowed to contain native modules. Android provides a set of security mechanisms to maintain the security of user data and system resources. Among them, application sandboxing and the permission system are two key features.

System-level security: application sandboxing. Android utilizes the Linux kernel as the basis of security and isolation. In Android, each application runs as a unique user with its own Linux user ID (UID). This design sets Android apart from the traditional Linux system, and provides natural kernel-level sandboxing among applications: each application stays within its own process boundary and does not have the privilege to interact with other applications. Thanks to the kernel-level isolation, protection covers both Java and native code.

Application-level security: the permission model. As described earlier, applications on Android are sandboxed and by default do not possess permissions to access security-critical information and devices. In order to be privileged, an application has to declare its necessary capabilities and gets user approval upon installation. Most permissions are checked when sensitive APIs are invoked, which is the only way an application could access corresponding protected resources. There are also a few permissions enforced by the Linux kernel. Moreover, applications may define custom permissions to limit interactions with other applications.

It is worth mentioning that under the current Android permission model, permissions obtained by an application apply to all of its components. It is not possible to grant permissions to only part of the application.

2.2 The Java Native Interface

The Java Native Interface (JNI) [13] is a framework allowing Java programs to interoperate with native libraries. In Java, the `native` keyword is used to declare native methods. The following code snippet of the `PlasmaView` class is extracted from a sample of the Android NDK, which declares a native method `renderPlasma`. Once declared, native methods can be invoked in the same way as ordinary Java methods. In the example, the `onDraw` Java method invokes `renderPlasma`.

```
public class PlasmaView {
  ...
  private Bitmap mBitmap;
  protected void onDraw (Canvas canvas)
      { long time_t = ...;
        renderPlasma(mBitmap, time_t); ...;}

  private static native void renderPlasma
    (Bitmap bitmap, long time_ms);

  static {System.loadLibrary(‘‘plasma’’);}
}
```

A native method is implemented in a native language, such as C, C++, or assembly. Native code may also use JNI functions to interact with Java. Through these JNI functions, native code can inspect, modify, and create Java objects, invoke Java methods, catch and throw Java exceptions, and so on.

In Android, the NDK includes a cross-compilation toolchain, which helps generate libraries from native code. It also provides a collection of APIs and system libraries that facilitate developers to perform various tasks from traditional libc function calls to OpenGL-based 3D graphics rendering.

Category	Apps	Apps with native libs	Percentage
Social	7	7	100%
Communication	5	4	80%
Gaming	19	18	95%
Entertainment	2	2	100%
Other	17	12	71%
Total	50	43	86%

Table 1: Top 50 Applications and Their Use of Native Libraries.

3. NATIVE CODE IN ANDROID: THE CURRENT SITUATION

In this section, we first present a small-scale study with a focus on the most popular applications on Google Play, which helps us better understand the use of native libraries in popular applications. Then we summarize how current security mechanisms in Android confine the behavior of native libraries.

3.1 Trends and Statistics

Native libraries are often treated as black boxes or simply trusted in past security research. In general, only a small portion of Android applications contain native libraries. A previous study showed that native libraries are found in less than 10% of the total applications inspected [6]. However, this reflects only part of the story.

More smartphone applications nowadays are about social networking, sharing, and entertainment. Social networking and gaming applications are among top categories that have been used by most users. Since native libraries are often used to perform CPU-intensive tasks such as image filtering, pixel rendering, audio/video encoding/decoding, which are common features provided by social applications, or are frequently performed in cellphone games, we expect native libraries to be much more likely to appear in popular applications.

To confirm this hypothesis, we investigated into those top applications: We downloaded the application package files (APKs) of the top 50 applications from the "Top Free in Android Apps" chart in Google Play in November 2013, which are the most popular Android applications at the time, regardless of their contents or categories. We then unpacked those APK packages and collected statistics on their usage of native libraries.

We identified 200 native libraries in total, showing an average of 4 native libraries per application. Table 1 presents the statistics. The first two columns show that according to the categorization in Google Play, applications in social, communication, gaming, or entertainment categories contribute to about two thirds of the applications in the Top 50 ranking. Among them, native libraries are pervasive: almost all applications falling in those categories carry at least one native library. Even in other categories such as music/audio or shopping, more than half of the applications inspected include native libraries. In fact, only 7 out of 50 applications surveyed are written solely in Java.

Our survey results and some previous studies (for example, [6, 31]) may seem contradictory. This can be explained by the following reasons. First, previous studies collected large numbers of applications without paying attention to their contents or popularity. As an example, Zhou *et al.* collected 204,040 applications from various online stores and reported native code deployment rate of 4.52% in total [31]. However, the majority of the applications may not be installed and used by most smartphone users: according to statistics shown in Google Play in November 2013, the No.1 application in the chart has accumulated almost a billion installs, while the application ranking 300 shows less than 500,000 installs in total, which is approximately only 0.05% of the top application. Second, the computing power in smartphones today has increased dramatically. Users are expecting more powerful applications and breathtaking games right in their palms. Hence, applications tend to be more sophisticated with more features and the possibility to incorporate native libraries increases.

3.2 Deployed Security Mechanisms

As introduced in Section 2.1, two core defenses are deployed in Android: the application sandbox and the permission model, which cover both Java and native code. We next take a closer look at each of them, with a focus on how the protection extends to native code.

First, the application sandbox builds upon the Linux user-based process protection, covering everthing running above the kernel, regardless of the language being used. Different from a traditional Java Virtual Machine, where native code is not covered by the Java security manager, native components in Android stay within the context of a particular application and cannot read/write other applications' data or files or perform privileged tasks, unless granted necessary permissions.

Second, the permission model ensures that only applications with necessary permissions may access resources that may harm users' privacy or lead to unexpected result if visited or used improperly. The permission model extends to native code in the following two aspects:

- Most security-sensitive system resources are accessed via Android API functions, which native code does not have direct access to. Native code may instead invoke Java methods through JNI's method-invocation functions, in which permissions are enforced.
- For those resources that are not wrapped by the API functions, they are controlled by Linux groups [7]. In these cases, the Linux kernel takes care of the permission checking when the underlying system calls are invoked.

4. DEFENSE OVERVIEW

In this section, we discuss the threat model of NativeGuard, as well as defenses it provides. Technical details are left to Section 5.

4.1 Threat Model

As shown in the survey, native libraries are prevalent in popular Android applications. Oftentimes, they are incorporated to fulfill some fixed and repeating tasks, such as file compression, 3D rendering, and audio/video stream decoding. Rather than implementing the functionalities from scratch, application developers often tend to utilize existing third-party native libraries, and connect them to their Java code via a small amount of glue code upon necessity. For instance, we have identified the same photo editing library (namely, `libaviary_native.so`) that appears in

several popular social and photography applications. The downside, on the other hand, is the potential security threats brought by third-party native components: although developers can improve the quality of their work through careful examination and thorough testing, they cannnot fully trust arbitrary third-party binaries.

Despite Android's security mechanisms discussed in the previous section, native libraries still pose serious threats to the application and system security. At a high level, our threat model puts an application's Java components into the Trusted Computing Base (TCB), together with the rest of the system, while focusing on the security of native code. The threat model is justified by the following reasons:

- In the context of Android, native libraries are inherently not as safe as Java components. First, developers have no control over third-party native libraries they incorporate: in most cases these third-party components appear as black boxes with only documented interfaces. Second, native libraries are the weak link of an application because native languages are more prone to various vulnerabilities due to their lack of basic safety mechanisms, such as type checking and bounds checking.
- A large amount of research work has been performed to either strengthen Android's security mechanisms or mitigate security threats (e.g., [5, 27, 29]). Unfortunately, they do not deal with the security of native code. By concentrating on native code security, our work complements related Android security research.

In NativeGuard, we focus on the following two ways through which buggy or malicious native libraries may compromise the safety and security of the application or the Android system.

- Native libraries in an Android application reside in the same address space as the rest of the application and therefore have access to the entire address space of the application sandbox. Native code, once exploited, may potentially access/modify any data within the process boundary, leading to possible privacy violation or application malfunction.
- As per Android's permission model, once granted, permissions are applied to the entire application. Native modules possess the same privileges and may possibly abuse system resources, resulting in leak of confidential information or unexpected results.

4.2 Defenses Provided

In order to defend against the listed threats, native code needs to be insulated from the rest of the application, and should run without unnecessary permissions. In NativeGuard, we utilize the natural process boundary provided by the underlying Linux kernel in Android to isolate untrusted native code. Figure 1 depicts the isolation at a high level. As the basis of isolation, native libraries are relocated to a second sandbox besides the original one. The native-code sandbox runs in a separate process, so native libraries no longer have direct access to the rest of the original application. To support ordinary native method calls and JNI function calls, a stub library and a set of service trampolines are introduced, which hide implementation details of interprocess communication (IPC) from the original Java and native code. Following this design, there is no need to retrofit application code: IPC is transparent to the application. In addition, if native libraries are separated from an application with permissions P, where P is the set of permissions granted to the application at install time, then native libraries are jailed in another sandbox with permissions P', where $P' \subseteq P$. That is, native code has only a subset of the original permission set. In fact for most applications, as is shown in Section 5 and Section 6, $P' = \emptyset$.

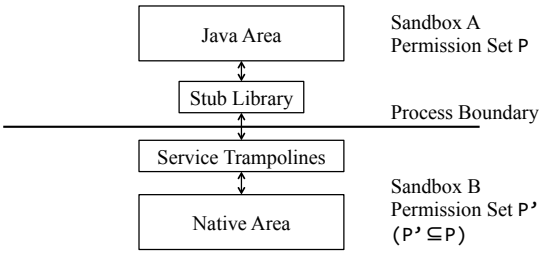

Figure 1: Overview of the isolation.

After separation, the process that holds native libraries can communicate with the Java process through the JNI interface, which unfortunately was not designed with security in mind. One worry is that malicious native code might misuse the JNI interface to cause integrity or confidentiality violations. For example, native code can issue JNI calls to modify fields of Java objects with values of incompatible types, which corrupts the integrity of the Java process. For another example, native code can forge a Java reference and issue a JNI call that uses the reference to read any memory in the Java process. This violates confidentiality. To prevent such violations, NativeGuard strengthens the JNI interface by performing runtime type checking when JNI functions are called. This guarantees that native code, even if malicious, can perform only type-preserving modification of the Java process and can read the Java process's memory through only the set of references passed from Java. This type safety guarantee is the same as what Robusta [21] provides to a conventional Java Virtual Machine.

One concern about process-based isolation is the performance overhead brought by context switches and interprocess communications. However, we believe it should work with most Android applications after a careful analysis of the Android NDK and usage patterns of most cellphone applications. First of all, based on the official document of the NDK [8], native code should be used only on tasks that perform CPU-intensive operations and that do not consume much memory, where context switches are not expected to occur frequently. Moreover, most applications today follow a user-driven model, where interactions with users are frequent and are the dominant factor in the application lifecycle. Hence, we consider that the idea of process isolation would not incur significant overhead most of the time. We present evaluation and performance data over benchmark programs in Section 6.

5. SANDBOXING NATIVE LIBRARIES

In NativeGuard, native libraries are put into another sandbox with limited permissions so that their security is controlled by Android's existing security mechanisms. We have different choices of implementing this sandbox.

(1) *Isolating native code in a second process within the same application.* By default, all components in one applica-

tion run in one process. Yet Android does allow one application to have multiple processes by supporting the `android:process` attribute in the manifest file of the application. It would be clean if native libraries run in another process but still stay in the same application as the rest of the code and resource files. However, this idea would not work with the current Android system, where permissions are enforced at the application level. Modifications to the underlying permission enforcement implementation would be inevitable, which would hinder NativeGuard from working directly with current Android smartphones.

(2) *Isolating native code in a second application.* In another approach, native libraries are moved to an entirely new *service* application running alongside the original one, the *client* application. The disadvantage is that the two applications have to be installed together, and there is no way to explicitly enforce any correlation between them. The benefit, on the other hand, is the compatibility with current Android systems. By putting native libraries into another application with less permissions, this approach sandboxes untrusted native libraries without requiring any modification to the system.

In the end, we chose the latter approach to keep NativeGuard a portable solution. Figure 2 shows the architecture of NativeGuard at a high level. An Android application is separated into two: the client application, which holds the Java side of the original application, and resource and user-interface files for interacting with users; and the service application, which takes the native libraries. The two applications interact via interfaces defined by the Android Interface Definition Language (AIDL) and the application holding native libraries acts as a service. It runs in the background and responds to requests from the client. Extra code is generated to support functionalities including native method calls, JNI function calls, NDK API invocation, and service application initialization.

We next present the detailed design of NativeGuard in four steps: (1) a brief overview of AIDL; (2) a closer look at the components and the workflow of NativeGuard; (3) how JNI function calls are handled across process boundaries; and (4) various technical issues concerning the NDK APIs.

5.1 AIDL Overview

AIDL is an interface definition language that defines the interface through which IPC is performed between a *client* and a *service*. AIDL follows Java's syntax. By defining the AIDL interface in an `.aidl` file and implementing necessary interface methods, Android application developers can easily utilize Android's IPC to marshall objects or perform remote method calls across processes. Specifically, the Android software development kit (SDK) takes the interface definition and generates a `stub` class in Java, which extends the `Binder` class, the core part of RPC support in Android. Developers then write code to extend the `stub` class and provide the actual implementation of interface methods. Then the interface is exposed to clients for IPC.

5.2 Native Code Separation

Given an Android application's APK package, NativeGuard separates native libraries in the application and the rest into two applications. In addition, it generates several AIDL interfaces and auxiliary native libraries, and slightly modifies the launcher class(es) (shown in dashed boxes in Figure 2) to achieve the separation. Next we elaborate several key components in the architecture.

Modified launcher activities. Similar to the `main` method of a Java program, the launcher activity is the starting execution point of an Android application. A callback method of the activity named `onCreate` is automatically invoked by the system when the application is launched. To ensure that the service application starts together with the client and the client is *bound* to it after launched, NativeGuard locates the `onCreate` callback method in the application and adds necessary code so that when launched, the client proactively binds to the service by sending out an *intent*. In order to keep NativeGuard working with arbitrary applications, where source code is not always available, we inject crafted Dalvik bytecode sequences with the help of apktool, a widely-used reverse engineering tool for Android applications. We leave the implementation details to Section 6.

Proxy native libraries. Since Java components and native libraries are put into two different applications, those Java components no longer have direct access to native libraries. NativeGuard's solution is to introduce a level of indirection through a layer of trusted proxy libraries. For each native library, there is one proxy library with the same name. Proxy libraries are put in the client application and provide implementations for exactly the same native methods as the original ones. The implementation of a native method in a proxy library invokes an appropriate AIDL interface method, which uses the IPC to invoke the corresponding native method in the service application. This level of indirection keeps changes transparent to Java code.

The AIDL interface for native method calls. An AIDL interface is defined for native methods and is exposed to the client application. After the client is bound to the service, native code in proxy libraries may invoke the corresponding IPC method through the JNI interface. For each native method, there is one AIDL interface method implemented on the service side. When the method is invoked, binder IPC code generated by the SDK copies the arguments to the service process and calls the implementation in the end.

Stub libraries. Calling an AIDL interface method transitions the control flow to the service application. Eventually the intended native method by Java code should be invoked. NativeGuards adds another layer of stub libraries for the purpose of maintaining a proxy table of JNI functions. The reason for this layer is the following. JNI passes object references to native methods as *opaque references*. Those references, however, no longer make sense when passed to another address space and need to be "properly translated" upon dereference. The proxy JNI function table redirects JNI calls back to the client application where opaque references could be correctly resolved. Details regarding stub libraries as well as the proxy JNI function table are presented in Section 5.3.

The AIDL interface for JNI and NDK API calls. As mentioned in the previous paragraph, JNI calls in sandboxed native libraries are redirected back to the client process, which requires another AIDL interface for the necessary IPC. Similarly, the implementations of some NDK API functions cache low-level pointers to objects and manipulate them directly in native code; they also call for the IPC to

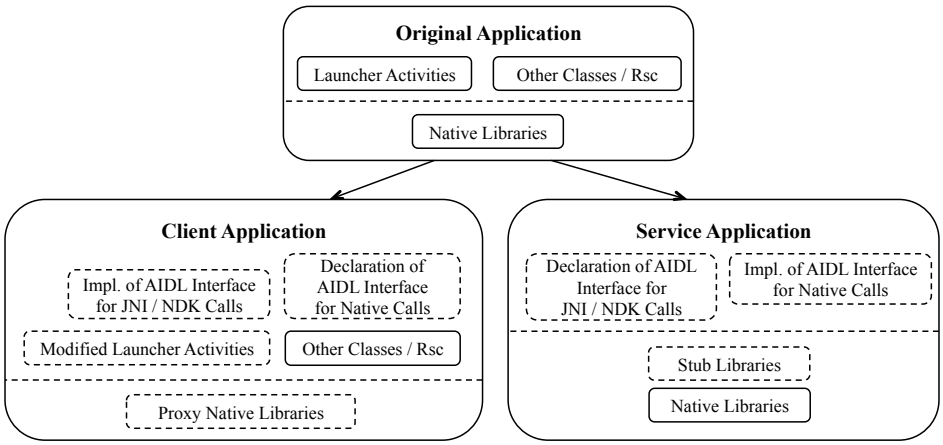

Figure 2: Architecture of NativeGuard.

interact with objects in the client process. Another set of AIDL interface methods are hence provided for this purpose. More details are discussed in Section 5.3 and Section 5.4, respectively.

We next walk through the basic process of native code isolation in NativeGuard using a native method example from `libjni_filtershow_filters.so`, a widely-deployed native library for photo editing. The example we present is `imageFilterVibrance_nativeApplyFilter`, a native method that applies the vibrance effect to a photo. The first step is to extract the original library and replace it with a proxy library. The proxy library contains a proxy function for each native-method implementation. The second step is to write/generate mandatory components shown in dashed boxes in Figure 2, put them in the two new applications, build and deploy them respectively. The following steps outline what happens when the user starts the instrumented application from the device.

(1) When the application is launched by the user, the client application is started first, which sends out an intent. The service application starts and responds to the intent. The client is then bound to the service.
(2) When the application loads the native library, it loads the proxy version of `libjni_filtershow_filters.so` on the client side.
(3) When the `imageFilterVibrance_nativeApplyFilter` native method is invoked, the control transfers to the proxy method. The proxy method performs a function call to the corresponding AIDL interface method.
(4) When the AIDL interface method for `imageFilterVibrance_nativeApplyFilter` is called, the control transfers to the service application. The service application loads the stub library if it is not yet loaded and performs a native method call to the corresponding stub function.
(5) The stub function takes over the control and (a) uses `dlopen` to load the real library in the service process if it has not been loaded; (b) uses `dlsym` to find the address of `imageFilterVibrance_nativeApplyFilter` in the real library; and (c) performs a function call to the real target function.

5.3 Supporting JNI Calls

Native code may use JNI functions to manipulate Java objects or call Java methods. For instance, the following native method `initIDs` extracted from class `java.util.zip.Inflater` in the Zlib library calls `GetFieldID` to obtain the JNI identifier of a field.

```
JNIEXPORT void JNICALL
Java_java_util_zip_Inflater_initIDs
  (JNIEnv *env, jclass cls){
  ...
  jfieldID strmID =
    (*env)->GetFieldID(env, cls, ''strm'', ''J'');
  ...
}
```

In the above code, `GetFieldID` returns the identifier of the field "strm" in the Inflater class, which can later be used to read from or write to the field. Note that the first two parameters of a native-method implementation are special: a JNI interface pointer, and a reference to the Inflater object or the Inflater class in the case of static native methods, similar to `this` pointer in C++ or Java. The `cls` bolded in the code above is the reference to the Inflater class.

After NativeGuard's isolation, the implementation of `initIDs` is migrated to the service application. Hence when invoked, `cls` in `initIDs` becomes the reference to the service class and is no longer the one that it is expecting.

As briefly discussed in the preceding section, NativeGuard's solution is to introduce a stub library between the Java code and the real native libraries in the service application. The stub library maintains a proxy JNI interface pointer structure in the service application, but with function pointers to the AIDL interface methods for JNI functions. When the stub function calls the real target function (the last step discussed in Section 5.2), it passes the proxy JNI interface pointer to the corresponding native library. Hence, when native code in the isolated library invokes a JNI function via the proxy interface pointer, the control is transferred to an AIDL interface method, which performs IPC and jumps back to the client application and the real JNI function is called. Note that opaque references must be retained so that correct objects could be retrieved when JNI function calls are redirected back to the client application. In the example above, the value of `cls` is mandatory to find the field ID and thus cannot be lost. In NativeGuard, one more argument is added to the AIDL interface methods for native method calls. When a proxy method calls to the AIDL interface (step (2) in Section 5.2), it passes in `this` reference value. The reference is kept all the way down to

the native code in the service application so that all JNI calls within this context could be properly handled back in the client application.

The benefit of this design is threefold. First, isolated native code still follows the same syntax for invoking JNI functions and thus need not be aware of the isolation. Second, only references are marshalled across processes, not Java objects. This brings down the overhead caused by IPC as object marshalling is very expensive. Also it makes the solution source-code free, as reference marshalling in AIDL does not depend on the implementation details of objects. Last, no global references are necessary in the proxy libraries on the client side, as JNI function invocations in the service application always stay within the context of a native-method call, where local references are valid during the period.

In addition, the proxy JNI interface provides a natural place for performing runtime type checking on the JNI. As we have discussed, misuse of the JNI interface by the native code can cause confidentiality and integrity violations. Therefore, NativeGuard performs checks in the proxy JNI interface to ensure type safety of JNI calls. The implementation of these checks follows previous JNI checking systems, including Arabica [22] and Jinn [12]. We omit a detailed description.

5.4 Supporting NDK API Function Calls

Ideally, native libraries interact only with Java code: they are not dependent upon each other albeit belonging to the same Android application. In this case, it is sufficient to support only the JNI interface in our framework as native libraries perform computation tasks and only communicate with the rest of the program via the JNI interface. The reality, however, is different when a few special NDK libraries are involved.

The NDK provides a set of API functions for native code to fulfill various tasks. Besides traditional libc support, the API also includes headers of OpenGL libraries for 3D rendering, libjnigraphics headers for bitmap pixel manipulation, and so on. As a simple example, native code may invoke `AndroidBitmap_lockPixels` implemented in `libjnigraphics` to grab the lock on a Java Bitmap object and acquire a pointer to the pixel buffer of the object, through which direct access to pixels is supported. In fact, many applications and libraries provide photo filter functionalities based on API functions in `libjnigraphics`, including `libjni_filtershow_filters.so`, the example library we introduced in Section 5.2.

It turns out that in some NDK libraries, manipulation on Java objects is directly performed via native pointers, rather than through the JNI interface. In these cases, Java objects are only "wrappers" of native data structures, which are allocated in native methods implemented in *system* native libraries. In the above example, the underlying implementation of `AndroidBitmap_lockPixels` first calls `GetIntField` to obtain the value of an integer field, specifically, `mNativeBitmap` in the Java Bitmap object, then casts the integer to a `SkBitmap` pointer, through which the Bitmap pixels can be directly modified. By further tracing down the calling sequence, we find that the Bitmap object is created by calling a native method `nativeCreate`, which (1) allocates a new `SkBitmap` object; (2) casts the pointer value to an integer; and (3) caches the integer to the `mNativeBitmap` field. `nativeCreate`, however, is implemented in `libandroid_runtime`, a system native library in Android. Since our purpose is to isolate untrusted, third-party native libraries, system libraries are considered trusted. Hence, when a native library is isolated in the service application and tries to modify a Bitmap object via the NDK API, its behavior cannot be predicted, because the pointer cached can only be interpreted in the client application and does not point to a valid object in the service application.

To support these NDK API function calls after isolation, NativeGuard intercepts an API invocation and marshalls related objects to the service application before the real API function is run. For instance, if a sandboxed native library invokes the `libjnigraphics` API, NativeGuard provides a fake `libjnigraphics` in the service application, which utilizes the dedicated AIDL interface to transport the Bitmap object from the client before calling the real API function. Correspondingly, updated Bitmap objects are marshalled back when native code finishes its work on the pixels, specifically, when it calls `AndroidBitmap_unlockPixels`, the API function that indicates the end of the native modification.

We next make a few clarifications. First, we can easily marshall Java objects like the Bitmaps as long as they support the `Parcelable` interface, which allows the system to decompose objects into primitives that can cross the process boundary. If not, then we have to implement the `Parcelable` protocol for the object on our own, which may or may not be difficult, depending on the composition of the object. Second, not all NDK libraries encounter the situation discussed in this section. Our prototype implementation provides support for `libjnigraphics`, the library we discussed earlier, and the OpenSL ES native audio library.

5.5 Managing Native Code Permissions

The basic idea of limiting privileges of native code in NativeGuard is to better use the Android permission mechanism. Following the principle of least privilege (POLP), native libraries are isolated in another application with only the minimum mandatory permission set granted. We next discuss how the POLP principle is enforced in NativeGuard.

Based on the guideline of the NDK and how native code is managed under current Android framework, we infer that native code *itself* seldom requires any permission. First of all, the NDK may only be beneficial when used with self-contained libraries that perform CPU-intensive operations [8], which are not likely to access system resources or devices protected by permissions. Second, native code is not allowed to interact directly with the system API and must call back to Java via the JNI interface to visit protected resources [7]. In this case, permissions are enforced on the API calls, not native libraries; JNI function calls do not require any permission either. The only circumstance under which native code does require permissions is when native code directly accesses system resources *not* protected by the system API. For instance, a native library may open and write to a file on the SD card without calling back to Java, and hence requires the `WRITE_EXTERNAL_STORAGE` permission itself.

To the best of our knowledge, there is no official document or previous research that sheds light on how native functions, library calls or NDK APIs are connected to the permission model. Intuitively, native code's access to protected resources is fulfilled by system calls and the permission check is performed in the Linux kernel. But it is hard to decide whether a system call requires an Android permission as it

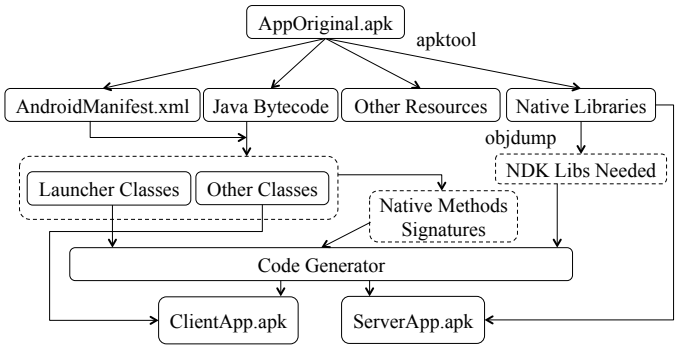

Figure 3: Workflow of the Application Separator.

may be dependent on its parameters. As a simple example, the system call `open` requires the `WRITE_EXTERNAL_STORAGE` permission when opening a file on the SD card with the write access, but requires no permission when opening a private file of the application or opening a file on the SD card with the read-only access. Furthermore, since NativeGuard aims to defend threats of untrusted native libraries in arbitrary Android applications from online stores, where source code of the native libraries is not available, it is even harder to find the minimum permission set efficiently solely from the binary.

Hence, we decide to follow a heuristic approach and by default grant no permission to the service application in NativeGuard. The approach is motivated by the observation that it is rare for legal native code to perform privileged operations, as it is a "bad practice" according to the NDK. In fact, as we will present in Section 6, the heuristic works with all real applications that NativeGuard has been tested on (around 30 applications). On the other hand, the drawback is that the approach does not support native code that requires privileged access to the devices. As a remedy, NativeGuard also supports a configuration file through which a developer can manually grant permissions to native libraries.

6. IMPLEMENTATION AND EVALUATION

In this section, we first present the prototype implementation of NativeGuard. Then we discuss its evaluation, on both benchmark programs and real-world applications.

6.1 Prototype Implementation

The thrust of this project, as we have discussed, is to design a framework that isolates untrusted native libraries in Android applications downloaded from online stores. Hence, we implemented NativeGuard as an "application separator", which takes as input an APK package and generates two APK files: the client and the service applications. As presented in Section 5, the client contains Java bytecode and other resources from the original application, the service isolates native libraries and by default does not require any permission. In addition, NativeGuard also supports a configuration file, where permissions can be manually granted to the service application upon separation.

Figure 3 shows the separation process in NativeGuard. At a high level, NativeGuard is composed of a code generator, and two third-party tools. First, we utilize apktool [1], an open-source tool for Android application reverse engineering, to extract native-method information and to make slight modifications to the launcher classes. Apktool can unpack an APK file into resources and `smali` code, a human-readable format for the Dalvik bytecode, where the basic information of the program, such as method names and native method signatures, is available. It can also rebuild them into an APK file after some modifications. Second, we incorporate the `objdump` from the NDK toolchain, which can dump an Android native library built by the toolchain to reveal information regarding library dependency. The main steps of the separation process are as follows.

(1) Apktool unpacks the input APK file.
(2) The `AndroidManifest.xml` file is parsed to locate launcher class(es).
(3) All `smali` files are analyzed to record signatures of native methods.
(4) Native libraries are dumped to record the dependency to NDK libraries.
(5) Based on information collected above, the code generator generates extra AIDL interfaces and library code. It also adds mandatory `smali` code to the launcher class(es) for service binding.
(6) In the end, two applications are built as the output.

NativeGuard is implemented in Java and is comprised of about 2,000 lines of Java code and about 20 template files for fast code generation. In addition, it incorporates apktool 1.5.2 and `objdump` from the NDK Revision 8c. We choose Java as the implementation language because the package installer in Android is written in Java. Therefore, NativeGuard can be incorporated into the package installer in the future, which provides users a completely transparent solution for native code isolation.

6.2 Evaluation

We carried out a three-stage evaluation to fully test NativeGuard's functionality and performance. First, we created an illegal native library that abuses granted permissions to test the effectiveness of NativeGuard's defense. Afterwards, NativeGuard was tested on dozens of applications in various categories that are among the most downloaded applications in Google Play. For performance testing, it was evaluated both on a representative benchmark suite and on a hand-crafted benchmark program. Experiments were conducted on a Nexus 4 smartphone running Android 4.3. All performance numbers were averaged over 10 runs.

Functionality testing. We have manually created a test native library, which *directly* accesses the location information of a phone without going through the Java side (assuming the application has been granted the location-access permission). In Android, most system resources are protected by privileged Java API methods, which are implemented in a trusted system process [7]. A typical native library has to invoke the corresponding API method through method-invocation functions in JNI in order to access the privileged resource. In contrast, our test native library directly talks to the system process and obtains the location information without going through the Java API.

The test library demonstrates that malicious native code can take advantage of permissions that are not really needed to cause security violations. NativeGuard can improve this situation. For the test native library, we used NativeGuard to isolate it into a service application with no permissions and thus its access to the location information was denied due to lack of permissions.

Benchmark	Size of Data Set	Overhead
jpeg	20 KB	17.16%
lame	177 KB	1.12%
	2585 KB	0.87%
tiff2bw	6662 KB	7.83%
	27873 KB	2.08%
tiff2rgba	6662 KB	3.14%
	27873 KB	0.54%
tiffdither	2223 KB	14.37%
	9303 KB	1.40%
tiffmedian	6662 KB	5.11%
	27873 KB	2.32%

Table 2: Runtime overheads on MiBench benchmarks.

Real-world applications testing. We collected a total of 30 applications from the official Google Play store to test NativeGuard's functionality on real applications. The applications are collected from different categories and are on the "top" charts of the store in November 2013. Note that tested applications are not strictly the overall top 30 ones, as we intended to exercise our framework on applications providing diverse functionalities and containing different native libraries. Our framework succeeds on 28 out of 30 applications. It fails on two applications because of the apktool, which fails either in the disassembling stage before separation or in the assembling stage after separation. Applications after separation are tested manually, as existing automatic testing tools that generate random inputs are not sufficient for our purpose. Since we want to make sure the native libraries are loaded and used during testing, it is much safer to just manually play with the instrumented applications and exercise various functionalities provided. For example, some applications require signing in before using any meaningful functions. In these cases, automatic testing frameworks are more likely to fail in exercising native code.

During our testing, we could perceive slightly longer response delays in some applications and use cases. For example, photo filters may take extra time to render a picture. But in general, NativeGuard introduces acceptable overhead and does not affect the functionalities delivered by the applications. Details regarding evaluated applications are presented in Appendix A.

Performance evaluation. For a security framework utilizing process isolation, the runtime overhead of NativeGuard depends greatly on the intensity of *context switches*, e.g. the frequency of IPC. If isolated native libraries are typical "good candidates" for the NDK and do not involve lots of context switches, the overhead caused by NativeGuard should be small.

We first evaluated NativeGuard on MiBench [9], a free and open-source benchmark suite for embedded systems. The suite provides six categories of benchmark programs for different purposes in real-world applications, and testing data sets of various sizes. MiBench is not Android-ready. We picked several benchmarks under the Consumer category (a category for consumer devices, like PDAs and smartphones) and ported them to Android. Table 2 presents the results.

Buffer Size	Overhead	Context Switches (per millisecond)
1KB	183.13%	31.89
2KB	107.49%	25.98
4KB	55.02%	18.51
8KB	34.36%	9.81
16KB	26.64%	3.96

Table 3: Extreme-case runtime overheads on the Zlib benchmark application.

In general, NativeGuard shows moderate overheads on MiBench programs. The benchmarks all utilize native libraries to perform CPU-intensive operations, such as image compression and conversion, and thus do not frequently make context switches. The result table also confirms the correlation between the overhead and the context switch intensity: programs on large data sets show less performance overheads, as they stay longer in the service application before switching back. Since native libraries in MiBench programs are representative candidates for the Android NDK, NativeGuard is promising to incur modest overhead on isolating a majority of native libraries in popular applications.

Another important factor to evaluate on mobile devices is the overhead posed on memory and storage, which are both limited resources on smartphones. Although for an application with native libraries, users with NativeGuard now have to keep two applications running simultaneously, the increase of memory utilization turns out to be moderate: NativeGuard introduces only 11.81% of memory overhead on the ported MiBench application, according to data reported by Android's `dumpsys` tool. Moreover, NativeGuard shows a tiny 130KB increase on the size of the application, being only 7.11% of the original.

But *what if the isolated library does make frequent context switches?* To further understand the performance of NativeGuard in extreme cases, we carried out another set of hand-picked benchmark programs. The programs compress a medium-sized file stored on the device using the popular Zlib library. When the user presses the start button on the user interface, the Java side of the application divides the file into data segments of smaller sizes and passes a data segment through a buffer to Zlib, which performs the compression and returns the result to the Java side. Then the Java side passes the next segment of data. Hence, when the Zlib library is sandboxed, the size of the buffer is strongly correlated to the performance overhead, as a smaller buffer results in more frequent context switches between Java and native code. We conducted experiments with different buffer sizes and the results are presented in Table 3.

As shown in the table, the runtime increase of NativeGuard on the Zlib benchmark demonstrates the similar trend: as the buffer size increases, the performance overhead decreases. The performance penalty can be as high as near two times when extremely intense context switch happens.

In summary, the experiments demonstrate that the approach of process isolation brings security to untrusted native code, and with modest overhead on most real-world applications where context switches between Java and native code are not frequent.

7. RELATED WORK

We next discuss related work in two categories: techniques for sandboxing untrusted components from a trusted environment, and previous studies on Android security.

Untrusted code isolation. It is always desirable to isolate untrusted code to prevent uncontrolled access or malicious compromise to the trusted environment and there have been various approaches. *Language-based isolation* ensures the security of untrusted code by utilizing static types [17] or object-capability models [16, 11], but it is tied to a specific language. *Isolation based on virtual machines* regulates untrusted code by building a safety-oriented platform (e.g., [3]), which is a clean solution but incurs severe performance penalty. In comparison, NativeGuard utilizes *hardware-based process isolation*, which has long been used in various operating systems to isolate untrusted components [4, 23]. Process isolation suffers from high performance overhead with frequent interprocess communication, but can provide flexible and robust isolation if used with clever optimizations. For instance, Codejail isolates untrusted libraries into a jailed process and incurs acceptable overhead on libraries that are tightly coupled with the main program [26].

With regard to sandboxing untrusted native code in foreign function interfaces, NativeGuard is similar to several previous frameworks. Klinkoff *et al.* designed a system that sandboxes unmanaged native code in the .NET framework [10], but relies on a kernel add-on module to control system calls in untrusted code. Robusta adopts software-based fault isolation (SFI) [25] and puts native libraries in Java programs into an SFI sandbox [21]. In terms of performance, it compares favorably to other sandboxing frameworks thanks to SFI, but relies on nontrivial modification to the internal of a Java Virtual Machine. Arabica improves Robusta and achieves JVM-portability through clever use of the Java Virtual Machine Tool Interface (JVMTI) [22], which, unfortunately, is not available for the Dalvik Virtual Machine on Android. On the contrary, NativeGuard reuses Android's permission model and does not need support for special interfaces or plug-ins, hence is ready to deploy on any Android system.

Android safety and security. As the most widely-adopted smartphone OS worldwide, Android has attracted much attention from academia in recent years. A couple of empirical studies provided more complete view of Android application security and its permission model. For example, Enck *et al.* designed a Dalvik decompiler `ded` and performed analysis on 1,100 Android applications [6]. Their work produced many findings, some of which might lead to ways of exploiting Andrioid. Felt *et al.* established a mapping between the Android API and the permissions, and shed light on the pervasive overprivilege problem in Android applications [7]. PScout utilizes static analysis to further improve the completeness of the mapping and reveals the state of the art in newer Android versions [2]. Their work is a strong motivation of NativeGuard as many applications are overprivileged with security-critical permissions, which are not needed by native libraries. Recent studies have presented various systems (e.g., TaintDroid [5] and VetDroid [29]) to detect user privacy and information leaks, which increased the overall security of Android, but left native libraries unmonitored.

Much work has been performed to address various aspects of Android security, for example, advertisements (e.g., [20, 19]), application repackaging and malware detection (e.g., [30, 24]), and privilege escalation attacks (e.g., [14, 15]). Though in a different context, AdSplit and AFrame are in spirit similar to our framework, as they isolate advertising libraries into separate processes [20, 28]. To display both the advertisement and the host application on the screen after separation, AdSplit follows an emulation approach to allow two activities to share the screen, while AFrame supports embedded activities. Both of them require changes to the system. In comparison, NativeGuard isolates native libraries into a service application that does not interact with the user, avoiding unnecessary modifications to the Android framework. Moreover, a number of systems have been designed to improve the permission system's granularity and flexibility (e.g., [18, 32]). These studies increase the power of the permission model, but requires changes to the system. They also risk overprompting users to make security-related decisions. By contrast, our framework reuses the current permission model, where users still receive the same information when installing applications.

8. FUTURE WORK

Some parts of NativeGuard can be improved. The next step is to incorporate NativeGuard into the Android package installer. NativeGuard is currently implemented as a command line tool, which genereates APK packages according to user commands. If integrated into the Android package installer, it would bring conveniece to end users; they can download, isolate native libraries, and perform installation in a streamlined process.

We also plan to explore techniques that efficiently decide the minimum permission requirement of native binaries. Currently our framework relies on a heuristic, which worked well in our multistage evaluation, but cannot support legitimate libraries that do require permissions. Since native code may directly access some system resources through system calls, it would be a good starting point to build a permission map to connect system calls to permission requirements.

9. CONCLUSIONS

Although the Android platform has a sophisticated security architecture for Java code, native libraries are uncontrolled. Given the increasing popularity of Android devices and insufficient research, native libraries pose pressing challenges to the security of the Android ecosystem. In this paper, we have proposed NativeGuard, a security framework that isolates native libraries into a non-privileged application. NativeGuard requires no change to the Android system, nor does it require access to an application's source code. It incurs modest runtime overhead on tested real-world applications, in which interprocess communication is not intensive. We believe that our study is a good starting point for future security research on native code in Android.

Acknowledgments

We thank anonymous referees of WiSec '14 for detailed comments on an earlier version of this paper. This research is supported by US NSF grants CCF-1217710, CCF-1149211, and a research award from Google.

10. REFERENCES

[1] android-apktool. `https://code.google.com/p/android-apktool/`.

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. pages 217–228, 2012.

[3] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy (S&P)*, pages 350–364, 2006.

[4] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 339–354, 2008.

[5] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[6] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *20th Usenix Security Symposium*, pages 21–21, 2011.

[7] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *18th CCS*, pages 627–638, 2011.

[8] Google. Android ndk. `http://developer.android.com/tools/sdk/ndk/index.html`.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, 2001.

[10] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna. Extending .NET security to unmanaged code. *Internation Journal of Information Security*, 6(6):417–428, 2007.

[11] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for web applications. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, pages 551–560, 2010.

[12] B. Lee, M. Hirzel, R. Grimm, B. Wiedermann, and K. S. McKinley. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *PLDI*, pages 36–49, 2010.

[13] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. pages 229–240, 2012.

[15] T. Markmann, D. Gessner, and D. Westhoff. Quantdroid: Quantitative approach towards mitigating privilege escalation on android. In *IEEE International Conference on Communication*, pages 2144–2149, 2013.

[16] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed System Security Symposium(NDSS)*, 2010.

[17] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[18] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[19] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[20] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating smartphone advertising from applications. In *21th Usenix Security Symposium*, 2012.

[21] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the JVM. In *17th CCS*, pages 201–211, 2010.

[22] M. Sun and G. Tan. JVM-portable sandboxing of Java's native libraries. In *17th European Symposium on Research in Computer Security (ESORICS)*, pages 842–858, 2012.

[23] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2004.

[24] T. Vidas and N. Christin. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 197–208, 2013.

[25] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 203–216, New York, 1993. ACM Press.

[26] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *17th European Symposium on Research in Computer Security (ESORICS)*, pages 859–876, 2012.

[27] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *20th CCS*, 2013.

[28] X. Zhang, A. Ahlawat, and W. Du. AFrame: Isolating advertisements from mobile applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[29] 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 *20th CCS*, 2013.

[30] W. Zhou, X. Zhang, and X. Jiang. Appink: Watermarking android apps for repackaging deterrence. In *8th ACM Symposium on Information, Computer and Communications Security*, pages 1–12, 2013.

[31] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Network and Distributed System Security Symposium(NDSS)*, 2012.

[32] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, pages 93–107, 2011.

APPENDIX

A. EVALUATION OF APPLICATIONS

Detailed information about applications evaluated in Section 6 is shown in Table 4. Notice that a "*" besides the version number or the size of an application indicates that the version or the size of that application varies with different devices, and the number shown in the table is the one for our testing device and system. There is no information for native libraries in `PlayKids`, because apktool fails to disassemble the application. For `DJ Studio 5`, we include its native libraries and dependencies in the table, but apktool fails to rebuild the application after disassembling. Further experiments show that the failure is not resulted from the instrumentation of NativeGuard, as apktool fails to build the application even without any change after disassembling.

Category	App	Version	Size	Native Libraries	Needed NDK Libraries
Photography	Snap Camera HDR	2.1.4	5.3M	libjni_eglfence.so libjni_filtershow_filters.so libjni_mosaic.so	libjnigraphics.so
	Photo Editor by Aviary	3.1.1	10M	libaviary_moalite.so libaviary_native.so libexif_extended.so	libjnigraphics.so
	Photo Editor	1.3.13	1.9M	libIUDeskImageFilter.so libIUDeskJpegCodec.so	libjnigraphics.so libjnigraphics.so
Social	Snapchat	4.0.20*	7.5M*	libphotoeffect.so	libjnigraphics.so
	ooVoo	2.0.4	21M	libovmedia-v7a.so	libjnigraphics.so libGLESv2.so
	Badoo - Meet New People	2.27.3*	19M	libScanPay.so	
Communication	WhatsApp Messenger	2.11.109	11M	libframeconv.so	
	AntiVirus Security	3.4.2.1*	9.5M*	libdeng.so	
	Handcent SMS	5.3	6.0M	libhccommon.so libmms2gif.so libspeex.so	
Tools	Brightest Flashlight Free	2.4.1	1.2M	libndkmoment.so	
	GO Keyboard Android Terminal Emulator	1.9.11 1.0.53	5.3M 456k	libMFtInput.so libjackpal-androidterm4.so	
Shopping	eBay	2.5.0.31*	10M*	libredlaser.so	
	Walgreens	4.1	18M	libaviary_moalite.so libaviary_native.so libexif_extended.so	libjnigraphics.so
	Out of Milk Shopping List	4.1.6*	8.9M*	libscanditsdk-android-3.4.0.so	
Games	Pou	1.4.8	16M	libsonic.so	
	Farm Story: Thanksgiving	1.9.6.3	16M	libs8.so	
	Cartoon Camera	1.99	1.0M	libgpuimage-library.so	
Business	Box	2.3.0*	11.3M*	libleveldb.so	
	Call Blocker	4.2.46.20	3.9M	libNqCrypto.so	
	Olive Office Premium (free)	1.0.89	19M	libchmjni.so libpdfjni.so	libjnigraphics.so
Books & Reference	Cool Reader	3.1.2-34	6.7M	libcr3engine-3-1-0.so	libGLESv1_CM.so
	Audible for Android	1.5.3*	9M*	libAAX_SDK.so	
	Ancestry	2.2.335*	5.7M*	libNativeTreeViewer.so	
Education	Mathway	1.0.3	13M	libmonodroid.so	
	PlayKids	1.1.1	17M	N/A	N/A
Music & Audio	iHeartRadio	4.10.0*	7.7M*	libaacarray.so	
	Bandsintown Concerts	4.3.2.1	10M	libdeezer.so	
	DJ Studio 5	5.0.8	11M	libaudio-jni.so	
Lifestyle	AroundMe	4.2.4	5.2M	libcountry-database.so	

Table 4: Separated applications and their information.