# On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android

Hao Hao, Vicky Singh and Wenliang Du
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA
{hahao,vsignh02,wedu}@syr.edu

## ABSTRACT

Bytecode rewriting on Android applications has been widely adopted to implement fine-grained access control. It endows more flexibility and convenience without modifying the Android platform. Bytecode rewriting uses static analysis to identify the usage of security-sensitive API methods, before it instruments the bytecode to control the access to these API calls. Due to the significance of this technique, the effectiveness of its performance in providing fine-grained access control is crucial. We have provided a systematic evaluation to assess the effectiveness of API-level access control using bytecode rewriting on Android Operating System. In our evaluation, we have identified a number of potential attacks targeted at incomplete implementations of bytecode rewriting on Android OS, which can be applied to bypass access control imposed by bytecode rewriter. These attacks can either bypass the API-level access control or make such access control difficult to implement, exposing weak links in the bytecode rewriting process. Recommendations on engineering secure bytecode rewriting tools are presented based on the identified attacks. This work is the first systematic study on the effectiveness of using bytecode rewriting for API-level access control.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

## Keywords

Android, Access Control, Bytecode Rewriting

## 1. INTRODUCTION

In the Android Operating System, application is the main unit that interacts with users. Android applications are implemented in Java, which is then converted into Dalvik bytecode that resides within DEX (Dalvik Executable) files after compilation. During execution, Dalvik bytecode is interpreted by register-based Dalvik Virtual Machine (DVM) before it is executed. Android provides rich Java API methods for applications to access privileged resources as shown in Figure 1. In addition to application isolation enforced by Linux Operating System, Android also supplies a permission system [17] to restrict operations, which an application can perform, on privileged resources. Permissions required by applications are declared in the `AndroidManifest.xml` files. During application installation, users are notified of permissions that are required by the applications. The users have the option to grant these permissions if they are satisfied; otherwise, they could choose to cancel the installation.


**Figure 1: Android API Usage**

The current Android permission system is coarse-grained, which causes the applications to be over-privileged. For example, when an application uses the `loadUrl` method of class `android.webkit.WebView` to load `www.facebook.com`, `INTERNET` permission is required. However, because `INTERNET` permission is coarse-grained which does not restrict which domain the application can access, the application is provided with more privilege than it truly needs.

Several methods have been proposed to provide fine-grained access control for Android. The first method is the bytecode rewriting technique [11, 14, 19, 22]. By instrumenting applications' bytecode, security policies could be enforced upon security-sensitive Android API methods. Bytecode rewriting can be done on both Java bytecode and Dalvik bytecode. Some tools actually convert Dalvik bytecode to Java bytecode, and rewrite on Java bytecode, because of the maturity of Java bytecode rewriting tools. Our evaluation could be applied to both Java and Dalvik bytecode rewriting. The second method is native library rewriting, which imposes fine-grained access control by intercepting the native calls to Bionic library [26]. A third way to implement fine-grained access control can be achieved by modifying the Android Operating System [15, 18, 20, 28]. Each of these methods has its own advantages as well as disadvantages. The purpose of this paper is not to compare these methods; instead, we would like to focus on one of the methods, the bytecode rewriting technique. Compared to the other methods, the bytecode rewriting endows more flexibility and convenience, and has been used in a number of existing works.

The existing work point out several places where bytecode rewriting can be attacked if it is not implemented cor-


*ASIA CCS'13,* May 8–10, 2013, Hangzhou, China.

rectly [14], but no work has systematically studied the potential attacks on the implementation of bytecode rewriting. It is unclear whether there are other attacks beyond what are described in the existing work. Because of the ever-increasing importance of such a technique, having a full understanding of its effectiveness in providing access control is crucial. This paper is the first to conduct such a study.

The purpose of this paper is three-fold: (1) We would like to study all possible attacks against bytecode rewriting, beyond what have been mentioned by the existing work. In fact, we did find some interesting attacks that were not documented in the literature. Without the awareness of these new attacks, the rewritten applications can be compromised by adversary. (2) For all the attacks, we would like to understand why such attacks work, and the fundamental cause for these attacks. Some attacks may appear to be different, but we found them to be quite similar upon close examination. Such observations are described in the paper to give readers a deeper insight into the attacks. (3) Based on our understanding of the attacks, we make recommendations on how the bytecode rewriting can defend against these attacks. These recommendations could help better the design of bytecode rewriting tools.

# 2. API-LEVEL ACCESS CONTROL USING BYTECODE REWRITING

The key idea with the API-level access control using bytecode rewriting is to provide modified behaviors, which allow fine-grained access control at the Java API level. The modified behaviors are introduced by secure wrappers which function as replacements of sensitive Android APIs. Thus, before invocations of those APIs, specified security policies are enforced. Figure 2 illustrates the approach.

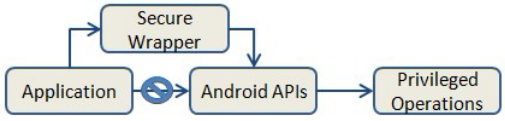



**Figure 2: API-Level Access Control**

The process of API-level access control using bytecode rewriting involves three steps. After the bytecode file is extracted from the APK [1] and decompiled [2, 8], static analysis is performed to identify the occurrences of the security-sensitive APIs in the code. These occurrences are modified, so customized access control logic is added before these APIs are invoked during the runtime. Finally, the bytecode file and all the other resources are repackaged into a single APK file, with a new digital signature generated for the file.

## 2.1 Dalvik Bytecode Analysis

To identify usages of Android APIs, method-invocation instructions need to be identified in the DEX file. The instructions are listed below [4]:

- `invoke-virtual` (`invoke-virtual/range`): used to invoke a normal virtual method.
- `invoke-direct` (`invoke-direct/range`): used to invoke either a private instance method or a constructor.
- `invoke-static` (`invoke-static/range`): used to invoke a static method.
- `invoke-interface` (`invoke-interface/range`): used to invoke an interface method on an object whose concrete class is not known.
- `invoke-super` (`invoke-super/range`): used to invoke the closest superclass' virtual method.

All of the aforementioned instructions have an argument called *method index*, which represents the method to be invoked. From this index, the fully-qualified method signature can be resolved, including parameters types, return type, as well as package, class, and method name. From this method signature, simple pattern matching can be used to identify the usages of the API methods that need to be restricted.

## 2.2 Dalvik Bytecode Rewrite

The existing work includes several rewriting mechanisms to enforce more fine-grained access control policies, e.g., placing the reference monitor in another service or within the application. Both of these techniques revise Dalvik bytecode to inject dynamic data checking, based on the policies specified by users. For the first approach [19], i.e., encapsulating sensitive APIs in a separate service, the application's `AndroidManifest.xml` file is modified so that permissions are removed and replaced with new permissions to access the secure service. Within the secure service, more fine-grained access control to the privileged resources are provided. During the process of bytecode rewriting, the calls to sensitive API methods are substituted with a completely new set of calls to the secure service. Because of its fail-safe default property, we did not evaluate this type of code rewriting, but our evaluation may be useful for this technique.

Instead, we focused on the second approach, i.e., the reference monitor on API methods is directly added to the application. After recognizing the utilization of API calls, the rewriting tool places the reference monitor, which uses secure wrappers, on sensitive APIs within the application. Access control policies can be placed on any public interfaces. Therefore, all the Java methods on the call chains would be restricted by the policies. The API method on which to interpose policy could be a method either in a class that can be extended, or in a class that is final and cannot be extended. We are going to discuss these two scenarios in more details below. Note that all the modifications are performed on the bytecode, but for a clearer presentation, we used Java code and not the bytecode in our examples.

**Bytecode rewriting for a `non-final` class.** Let us use an example to illustrate bytecode rewriting in this scenario. In Android, `WebView` is a `non-final` class. Assume that we want to put some restrictions on its `loadUrl` method, so we only allow the application to load certain URLs. To achieve this goal, we can define a wrapper class for `WebView`, and it is named `SecureWebView`. In this wrapper class, we perform some access control on the URL string before passing it to `WebView`'s `loadUrl`. See the following code:

[1] Android Applications are distributed as a single archive file called APK (Android Application Package) , which holds a DEX bytecode file, `AndroidManifest.xml`, XML resources, and any other resources the applications need.

```
public class SecureWebview extends WebView {
   public void loadUrl(String str) {
      //access control implementation
      ...
      super.loadUrl(str); }}
```

Once we have this wrapper, we replace all the occurrences of `WebView` in the bytecode with `SecureWebView`. See the following examples:

```
"WebView wv;"  is replaced by
"SecureWebView wv;"

"public class MyWebView extends WebView"
    is replaced by
"public class MyWebView extends SecureWebview"
```

**Bytecode rewriting for a `final` class.** Unlike in the previous method, there are cases in which the API methods belong to a `final` class. In these cases, we cannot define a subclass or use it as the wrapper. Another way to write the wrapper is thus needed. For the purpose of illustration, let us assume that `WebView` is a `final` class, and the following code shows how its `loadUrl` API is used.

```
class UserClass {
public void navigatePage(String url) {
      WebView w = new Webview();
      w.loadUrl(url); }}
```

We would like to put a restriction on the URL string when `WebView`'s `loadUrl` API is called. To achieve this goal, we can create another class `SecureWebView` with a static method `loadUrl`. In this static method, instances of `WebView` are passed as a parameter. All the invocations of `WebView.loadUrl` are replaced with `SecureWebView.loadUrl`, which implements access control. See the following code:

```
public class SecureWebView {
public static void loadUrl(WebView w, String s){
      //access control implementation
      ...
      w.loadUrl(s); }}

public class userClass {
public void navigatePage(String url) {
      WebView w = new Webview();
      /* w.loadUrl(url); is replaced by the
          following */
      SecureWebView.loadUrl(w, url); }}
```

## 2.3 Bytecode Rewrite Assumption

Android application can introduce its own native code mainly for performance reasons. With the existence of native code, the applications can directly invoke any native library functions of Android framework. Because app introduced native code is running in the same process space as DVM. The applications also have the ability to tamper the integrity of DVM state making method signature information unreliable. Hence, bytecode rewriting can be easily circumvented. Current bytecode rewriter assumes that applications either do not have native code or their native code is blocked from being executed [14]. This assumption is reasonable. From previous work [27], only 4.52% apps have native code. Hence, in our study we evaluated bytecode rewriting technique without considering app introduced native code.

# 3. EFFECTIVENESS OF API LEVEL ACCESS CONTROL

To measure the effectiveness of the API-level access control using the methods described above, we need to understand what it is trying to protect and how these protected resources are accessed under the hood. As depicted in Figure 3, resources that are out of process boundaries require privileged operations for access. These resources could be hardware devices, kernel data, or data in another process space. Hardware accesses are performed through instructions only executable in the kernel. Data from another process space is retrieved through inter-process communication, which is implemented by the binder driver in the kernel [3, 10, 13]. Access to kernel data and privileged instructions within the kernel are achieved through system calls.

To initiate system calls into the kernel, Android provides standard native C libraries called Bionic libc. These native libraries are bridged to Android Java APIs, so that they can be invoked directly from Java via the Java Native Interface (JNI). For example, `loadUrl` is a Java API in `android.webkit.WebView`. It invokes the native method `nativeLoadUrl` of the shared library `libwebcore.so` via JNI, and the shared library communicates with the kernel via socket system calls in order to access internet resources.

To acquire privileged resources protected by system services, such as locations, the `LocationManager` API interacts with the service called `LocationManagerService` using inter-process communications. The service obtains the location from the GPS device and then returns the result.

As discussed in the previous section, the Android install-time permission system protects these privileged resources, but at a level that is quite coarse-grained. The API-level access control using bytecode rewriting can intercept the calls from applications' own Java code to APIs, and place fine-grained access control before the invocation of those APIs. However, the way in which applications access privileged resources are quite complicated; therefore, it is very natural to question whether the bytecode rewriting is complete enough to prevent applications from directly accessing privileged resources, bypassing the secure wrapper. More specifically, we have the following questions: (1) Can applications directly talk to kernel? (2) Can applications directly invoke native libraries? (3) Can applications directly talk to system services? (4) Can applications hide the usage of APIs to deceive the bytecode rewriting?

These four questions represent four potential paths, as indicated by the numbers in Figure 3, to access privileged resources. If an application takes advantage of any of these paths, it would lead to an attack on API-level access control using bytecode rewriting.

**Access Kernel from Application Directly.** The first attack depicted in Figure 3 is not possible, because for applications to interact with the kernel, they must be able to invoke system calls. To be specific, they have to be able to execute special instructions provided by the CPU, such as the SWI instruction in ARM processors or the SYSENTER, SYSEXIT instructions in Intel processors [9]. DVM does not provide opcode to directly execute these instruc-

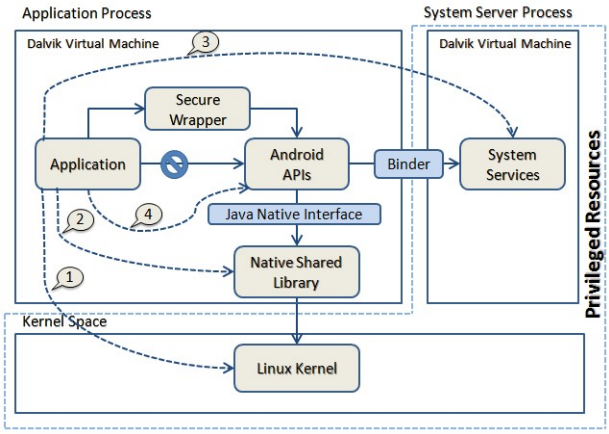



**Figure 3: Potential Vulnerabilities of API-Level Access Control**

tions. In other words, Java code must invoke the native code, such as the Bionic libraries, to communicate with the kernel. Therefore, the path marked with 1 in Figure 3 is not a feasible attacking path.

**Disguise Usage of APIs.** Because of Java language flexibility, the fourth attack which is concealing APIs usage is feasible. The most well-known case is Java *reflection*, whose troublesome aspect to bytecode rewriting has been documented by the existing work [14]. Using APIs `java.lang.reflect` package, any fields or methods of classes can be accessed or modified despite public or private at runtime. Thus, besides normal method invocation, reflection provides another way to invoke methods or change the behavior of methods. Reflection even can be used recursively (self-reflection). Hacker may make use of these powerful capabilities to disguise APIs usage. However, current bytecode rewriter [14] proposed defense mechanism against these attacks by detecting and preventing methods invocation of `java.lang.reflect` package. Other than Java reflection, other techniques such as polymorphism may also result in incomplete detection of APIs usages. Dynamic binding [21] enables methods to be bound during runtime but compilation. Any methods that adopts dynamic binding cannot be determined at compile time, and hence can escape from detection. However, researchers have discussed how to perform type inference on Dalvik bytecode effectively [23].

**Invoke Native Methods & System Services Directly.** Because the first attack path is infeasible and the fourth attack path has been actively discussed, our main focus is second and third attack path. In the following sections, we present our successful attacks with structure of attack mechanism, attack evaluation and recommendation on countermeasure. Attack mechanism illustrates techniques required to make attack successful. Attack evaluation demonstrates how applications can bypass current bytecode rewriter using our attacks. Our evaluation is performed based on Android 4.0.3 [1] SDK. All case studies are tested on a Android 4.0.3 ARM emulator. To help readers understand our attacks, we first present some background materials on Java Native Interface (JNI).

## 3.1 JNI Native Methods Resolution

To let Java code invoke native code and vice verse, there needs to be an interface. This interface is called Java Native Interface (JNI) [6]. In Android, JNI is the only interface that enables the two-way communication. Native code written in C/C++ and assembly is made available via shared objects (.so files) or shared library. For Java code to make a call into native code in a shared library, first, DVM needs to identify that the Java code tries to invoke native code. This is done through the definition of `native` method in Java, and `native` methods are the entries to native code. Second, DVM needs to identify which functions in the loaded shared library should be invoked, and then execute them. We call these functions *native library functions* to distinguish with the *native Java methods*. The following code snippet demonstrates the usage of native code in Java.

```
1 package edu.com;
2 public class MyClass {
3    native public long myFunc();
4    static {
5       System.loadLibrary("myLib"); }}
```

In the code above, the `MyClass` class from the package `edu.com` declares the `myFunc` method using the **native** modifier, which indicates that this method is actually an entry point to native library function. At line 6, the shared library `myLib` which contains the native functions is loaded into the DVM. Once the library is loaded, DVM will attempt to link the native Java method `myFunc()` to its corresponding native library function. The linking can be conducted in two different ways.

**Dynamic Name Resolution.** When the first invocation of the `myFunc` method happens, DVM searches in the loaded native libraries to find the corresponding native library function. The searching is performed based on names. DVM deduces the name of the corresponding native library function using the following convention:

1. The name uses `Java/` as the prefix.
2. The package name and class name are appended to the prefix. Since the package name is `edu.com` and the class name is `MyClass`, the resulting string is `Java/edu/com/MyClass`.
3. The name of the native Java method is appended to the above string. Since the method is `myFunc`, the resulting string is `Java/edu/com/MyClass/myFunc`.
4. Any `"/"` in the above string is replaced with `"_"`. This results in the final string `Java_edu_com_MyClass_myFunc`.

The DVM will then search the loaded native libraries for a function called `Java_edu_com_MyClass_myFunc`. If it is found, the DVM will set up the internal data structure to direct all future invocations of `myFunc` to this native library function. In the native library, the function corresponding to `myFunc` must be defined in the following way:

```
1 JNIEXPORT jlong Java_edu_com_MyClass_myFunc(
      JNIEnv* env, jobject thiz);
```

**Static Methods Association** In the method mentioned above, linking to native functions is decided by DVM and conducted when the first invocation occurs. There is another approach, in which the linking is decided by the native library through registration, not DVM. This registration typically happens when the native library is loaded, at which time a JNI function called `JNI_onLoad` in the native library is invoked. This function registers native functions to Java methods using the `RegisterNatives`. The following code snippet from a native library illustrates the process.

```
static JNINativeMethod method_table [] = {{
"myFunc", "(J)J",
          (void *) myFunc_Implementation }};

extern "C" jint JNI_OnLoad(JavaVM* vm, ... ) {
  jclass c = env->FindClass("edu/com/MyClass");
  env->RegisterNatives(c, method_table, 1); }
```

In the above example, within the `JNI_OnLoad` function, the `myFunc_Implementation` native library function is registered to the `myFunc` native Java method in the `edu/com/MyClass` class. Note that in this case, native function `myFunc_Implementation` does not need to follow the JNI naming convention as that in the previous dynamic name resolution case. When a shared library is loaded using the JNI method `System.loadLibrary`, DVM searches for `JNI_OnLoad` in the shared library. If the function is found, it will be invoked, so DVM can link the future invocation of `myFunc` to `myFunc_Implementation`.

Another technique for static association is to delegate this responsibility to another native function within the library and invoke this function after the library is successfully loaded. This is commonly used in Android framework. The following code snippet from `android_runtime` library demonstrates this technique:

```
static JNINativeMethod camMethods[] = {
{ "native_takePicture", "(I)V",
  (void*)android_hardware_Camera_takePicture},
   ... };
int register_android_hardware_Camera(JNIEnv* env)
{ return AndroidRuntime::registerNativeMethods
  (env, "andriod/hardware/Camera", camMethods,
   ...);}
extern "C" jint
Java_com_android_internal_util_WithFramework_
  registerNatives(JNIEnv* env, ...) {
  return register_android_hardware_Camera(env); }
```

In the above example, `Java_com_android_internal_util_WithFramework_registerNatives` native library function can be invoked by `registerNatives` in `com.android.internal.util.WithFramework` class through the dynamic JNI naming resolution. This function registers other native functions to Java methods of `Camera` class.

# 4. INVOKE NATIVE METHOD DIRECTLY

From the discussions above, it is clear that to access privileged resources at the kernel, applications have to utilize shared libraries. Apparently, the shared libraries could be either provided by the Android platform or introduced by applications. As we have stated earlier, in this paper, we follow current bytecode rewriter assumption that app introduced native code is not considered.

Bytecode rewriting enforces access control at the API-level in Java bytecode, not in native code. Therefore, if an application can directly invoke the native code without going through those restricted APIs, the API-level access control can be bypassed. This path is illustrated in Figure 3 as path 2 and Figure 4. In this section, we analyzed whether it is possible for application's Java code to directly invoke the native code in the shared libraries provided by the Android platform. We did identify two possible ways. Attack 1 is presented in this section, and attack 2 is presented in the section 5.

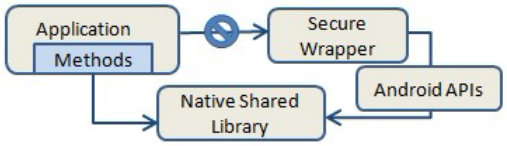



**Figure 4: Invoke Native Library Directly**

## 4.1 Exploit JNI Naming Convention

The way how native library functions are bound to native Java methods, either through the JNI naming convention or through registration, ensures that a native library function is only bound to a particular Java API, i.e., that API is the only path leading to the native library function. Therefore, if the bytecode rewriting puts a secure wrapper on that Java API, all the invocation of the corresponding native library function has to go through the wrapper, hence going through the intended access control. If we can find a way to invoke a native library function without going through its corresponding Java API, we can evade the restriction enforced by the wrapper.

**Objective and Approach.** The objective of our study is to evaluate whether there is a way to invoke native library functions through an unintended Java method, instead of from the one under the protection. In this subsection, we focus on the JNI naming convention, i.e., the dynamic naming resolution. This naming resolution attempts to link a native library function to a unique Java method, achieving a one-to-one mapping. Our approach is to study whether there is a loophole in the naming resolution that allows us to invoke the same native library function from two different Java methods, breaking the one-to-one mapping. Our attack strategy is illustrated in Figure 5.

**Attempt 1 (Failed):** As demonstrated in the last example, when DVM deduces the name of a native library function, `"/"` is replaced with `"_"`. One hypothesis is *whether using "_" in the package, class, or method name can cause ambiguity in the naming convention, and thus break the one-to-one mapping.* Let us look the following two classes:

```
//Example 1:
package edu.com;
public class MyClass {
   native public long my_Func(); }

//Example 2:
package edu.com.MyClass;
public class my {
   native public long Func(); }
```

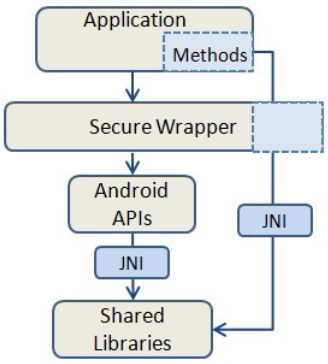



**Figure 5: Exploit JNI Naming Convention**

It appears that both native Java methods will map to `Java_edu_com__MyClass_my_Func`. Unfortunately (to attackers), Android has already thought about such a potential ambiguity, and its naming resolution replaces any `"_"` in the name with `"_1"`. Therefore, the first method maps to `Java_edu_com_MyClass_my_1Func`, while the second maps to `Java_edu_com_MyClass_my_Func`. Other unicode characters, such as `"&"`, `";"`, `"["`, etc., are also replaced in the same fashion. Our attempt failed.

**Attempt 2 (Succeeded)** Our failed attempt does lead to another hypothesis: *what if we put the number "1" in front of a package, class, or method name, can we cause ambiguity?*

The reason why the JNI naming convention can use `"_1"` to avoid the ambiguity caused by `"_"` is that in Java, no package, class, or method name can start with a digit [7]. Any attempts to do so will encounter compile-time errors. After digging into this, we realize that this naming violation is only detected at the compile time, not during the run time. Therefore, an attacker can add the digit to the beginning of a name at the bytecode level, i.e., through his/her own bytecode rewriting. We tried this, and have successfully invoked a native library function from an unintended Java API. We will show an example in the following.

Based on the JNI naming convention, the native library function `Java_edu_com_MyClass_my_1Func` is supposed to be mapped only to the `my_Func` function defined in the first example in our previous attempt. However, we can successful call this same native library function through a different Java method, i.e., the `1Func` method. It should be noted that in our Java code, we used `Func()` to pass the compiler, and then changed `Func()` to `1Func()` directly on the bytecode. Here is what the program looks like if we convert the modified bytecode back to Java code:

```
1 package edu.com.MyClass;
2 public class my {
3    native long 1Func();
4    static { System.loadLibrary('myLib'); }}
```

Because now two methods can invoke the same native library function, if the API-level access control only block the intended API, i.e., `my_Func`, in order to prevent applications from accessing privileged operations through the corresponding native library function, the attacker can bypass this blocking and invoke the native library function through another API.

## 4.2 Case Study

Our attack works on the native functions that have `"_1"` in their names. We searched in the shared libraries provided by Android, and found some cases. For example, the `sqlite_jni` library contains a function called `Java_SQLite_Database_error_1string`, with the `"_1"` pattern in the name. This function maps to the `error_string` method in the `Database` class of the `SQLite` package. We evaluated our attack based on this case. The attack objective is to invoke native library function `Java_SQLite_Database_error_1string` of `sqlite_jni` library from a different Java method `1string` of class `SQLite.Database.error`.

```
1 package SQLite.Database;
2 public class error {
3    public static native String 1string(...);
4    static { System.loadLibrary('sqlite_jni'); }}
```

We had to rename Java method `string` to `1string` directly on bytecode after compilation to deceive compiler. Figure 6 taken from logcat output demonstrates that by invoking `SQLite.Database.error.1string` we successfully invoked `Java_SQLite_Database_error_1string`. Because of different method signatures, any access control policies placed on the `SQLite.Database.error_string` method by current bytecode rewriter would not affect invocation of method `SQLite.Database.error.1string`. Restrictions on the application become ineffective.

| Time | PID | Application | Tag | Text |
|---|---|---|---|---|
| 12-03... | 364 | | dalvikvm | Debugger has |
| 12-03... | 373 | com.data | Log SQLite.Databasr.error.1string Method output | unkown error |

**Figure 6: Successful Attack on SQLite app**

Actually, if `"_"` appears in the name of a package, class, or native Java method, then the corresponding native library function must have `"_1"` in its name, creating an opportunity for our attacks. Having `"_"` in these names is not a common practice but at the same time is not totally absent. Although today, luckily, there are not many native library functions with this pattern, the existence of this loophole can cause many problems when more native libraries are added to Android. To ensure the security of the API-level access control, this loophole needs to be fixed.

## 4.3 Recommendations

To circumvent exploiting JNI name convention, bytecode rewriter can perform checking on naming convention of type descriptor and method. If any of them starts with numbers, bytecode rewriter should remove the digit as it is illegal.

# 5. EXPLOIT JAVA CLASS RELOADING

Another way to defeat the secure wrapper is to somehow modify the implementation of the APIs that the wrapper is trying to protect. If we can do this, the unauthorized access can be launched from the modified implementation, which is "behind the defense line" protected by the wrapper.

**Objective and Approach.** To achieve the above goal, we need to find ways to replace an existing trusted Java class (provided by Android) with another Java class provided by the application (untrusted). The application's Java class (and its methods) will have the same name as the existing Java class (and its methods). This way, the application's class can invoke all the native functions that the existing Java class can invoke. These invocations cannot be restricted by the API-level access control placed in the secure wrapper. We would like to exploit problems with the class loading mechanism in Android to achieve our goal. Figure 7 depicts our strategy.

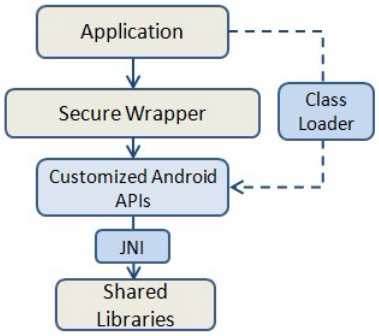



**Figure 7: Exploit Reload Java Classes**

In this study, we use `android.hardware.Camera` class as an example. This class is provided by Android, and is considered as trusted. A malicious application wants to redefine this class, so when this name is used, the redefined class is used, not the one provided by Android. Here is the code snippet of the redefined class:

```
1 package android.hardware;
2 public class Camera{
3    final public void someFunc() {
4        // Calling the privileged function
5        privilegedFunc(); }
6    native void privilegedFunc(); }
```

In the above code, the native function `privilegedFunc` is one of the APIs protected by the secure wrapper in the bytecode rewriting technique, but `someFunc` is not restricted (the name of this method can be arbitrary). If we use the original `android.hardware.Camera` class, our access to the `privilegedFunc` is restricted because of the wrapper placed on this class. However, if we can get DVM to load our redefined class, instead of the original Android class, and invoke `privilegedFunc` through `someFunc`, the access of the privileged function becomes unrestricted.

**Attempt 1 (Failed):** We attempt to load our redefined class (stored in the `Camera.apk` file) in DVM. Android provides `dalvik.system.DexClassLoader` to allow applications to dynamically load classes. We did the following:

```
1 DexClassLoader classLoader = new DexClassLoader(
2    "Camera.apk", ..., getClassLoader());
3 Class mClass = classLoader.loadClass("android.
     hardware.Camera");
4 android.hardware.Camera c = (android.hardware.
     Camera)mClass.newInstance();
5 //Access the privileged native code through
     someFunc()
6 c.someFunc();
```

Unfortunately, the above attack does not work. Apparently, our redefined class was not loaded. We looked into the code and find out where the problem is. In the class `DexClassLoader`, we found that using `DexClassLoader`, a Java class cannot be loaded again if it has already been loaded by this class loader or by its parent class loaders [2]. This loading policy is implemented in `DexClassLoader`.

**Attempt 2 (Succeeded):** To make our previous attempt successful, we need to change the loading policy, which means we need to change `DexClassLoader`. This is impossible without modifying the Android operating system. Fortunately, DVM allows us to write our own customized class loader. Our idea is to use this customized class loader to load our redefined `android.hardware.Camera` class, and then somehow use this class in the application, instead of the one provided by Android. DVM does allow two classes with the same name to coexist, as long as they are in separate class loaders.

In DVM, the `DexClassLoader` class is a subclass of `BaseDexClassLoader`, and the loading policy is implemented in the `loadClass` method. We can write another subclass of `BaseDexClassLoader`, override its `loadClass` method, but we skip the loading policy enforcement logic in this method, i.e., we load the class without checking whether it is already loaded by itself or by its parent class loaders. Here is the implementation of our class loader:

```
1  public class MyDexLoader extends
2              BaseDexClassLoader {
3    // Constructor omitted
4    @Override
5    public Class<?> loadClass(String s) {
6       Class c;
7       try { c = super.findClass(s);
8          return c;
9       } catch (ClassNotFoundException e) {
10         // handling the exceptions
11      }  return null; } }
```

In our customized class loader, we use `super.findClass` to load the class. Unlike the implementation in the class `DexClassLoader`, we do not check whether the class is already loaded by its parent class loaders. Now, we can use `MyDexLoader` to load our redefined `android.hardware.Camera` class without any problem.

There is another challenge. The `privilegedFunc` native function in our redefined `Camera` class needs to be linked to its corresponding native library function. Here is the problem: In Android, a Java method can only be linked to a native library function if they are both associated with the same class loader. In our case, the redefined `Camera` class is associated with our class loader `MyDexLoader`, but the native library will be associated with another class loader if it loaded the library [3].

We tried to resolve the challenge from association of native library and class loader, but without much success, mostly

[2]Each class loader needs to have a parent class loader, unless it is the first one.
[3]A native library is associated with the class loader who loads it.

because the association policy is enforced by DVM, which cannot be overwritten by Java code. However, we investigated that if a class loader is not updated with the list of libraries that loaded in DVM, then it is possible to reload those libraries.

Our investigation revealed that the default class loader (bootstrap) of DVM is not aware of `android_runtime` library being loaded. Hence making Java classes associated with this library a candidate for attack. `registerNatives` native function in this library would link plenty of internal native functions with Android Java classes.

## 5.1 Case Study

**Attack Background.** We performed our attack on a camera application [5]. Its main functionalities are taking pictures and then save the taken photos under the external SD card directory.

```
1 public boolean onOptionsItemSelected(MenuItem
      item) {
2 /* normal method invocation */
3 camera.takePicture(...);
4
5 /* equivalent method invocation using reflection
      */
6 Class c=Class.forName("android.hardware.Camera");
7 Method m=c.getDeclaredMethod("takePicture", ...);
8 m.invoke(camera,...); */  }
```

Suppose current bytecode rewriter enforced finer access control on method `Camera.takePicture` which specifies that pictures can only be taken at daytime between 8am to 6pm.

```
1 public class SecureCamera{
2 public static void takePicture(Camera camera,
      ...){
3     Time now = new Time();
4     now.setToNow();
5     if(now.hour > 8 && now.hour < 18) {
6        camera.takePicture(...); }}}
```

Current bytecode rewriter would also put some restrictions on *reflection* method `java.lang.reflect.Method.invoke` to prevent method `android.hardware.Camera.takePicture` from being invoked through `invoke`, as discussed in section 3.

```
1 public class SecureMethod{
2 public static void invoke(Method m, ...){
3     String name = m.getDeclaringClass().getName()
          + "." + m.getName());
4     if(!"android.hardware.Camera.takePicture".
          equals(name)) { m.invoke(); }}
```

**Attack Detail.** By exploiting Java class reloading, malicious developer can still take photos despite the access control policy. The attack mainly involves three steps.

First reload redefined `android.hardware.Camera` class into a new class loader. Without the needs of registration to native functions through name mapping, non native Java method can be renamed in the user-defined class. Method `takePicture` is a non native Java method so that we can rename it to `takeMyPicture`.

```
1 package android.hardware;
2 public class Camera {
3  public void takeMyPicture(...) {...}}
```

However, `takePicture` invokes `native_takePicture` native Java methods which contains actual implementation of taking pictures. Hence, we need to associate native Java methods of `Camera` class with corresponding native library functions. Refer to the code snippet in section 3.1, Java method `com.android.internal.util.WithFramework.registerNatives` in `android_runtime` library registers native library functions to `Camera` native Java methods. By reloading `android_runtime` library and invoking method `registerNatives`, native library functions are linked to redefined `Camera` class.

Then `takeMyPicture` can be invoked using reflection.

```
1  // Create a customized class loader
2  MyDexLoader ld = new MyDexLoader(...);
3
4  // Load redefined Camera class
5  Class c = l.loadClass("android.hardware.Camera");
6  Class util = l.loadClass("com.android.internal.
       util.WithFramework");
7  Method m = util.getDeclaredMethod("
       registerNatives", ...);
8  m.invoke(...);
9
10 // Invoke takeMyPicture method using reflection
11 m = c.getDeclaredMethod("takeMyPicture", ...);
12 m.invoke(...);  ... }
```

Because the method name `android.hardware.Camera.takePicture` has changed to `android.hardware.Camera.takeMyPicture`, invocation of `takeMyPicture` cannot be restricted. Figure 8 demonstrates the application can still take photos even though current bytecode rewriter enforced access control policy that pictures cannot be taken during nighttime.

To estimate the potential impact of this attack, we have identified all the native libraries that are not associated with default class loader in Android. The full list can be found in Appendix A. The functions in these native libraries are mapped to the Java methods in various Java classes. In the Appendix B, we present all Android Java classes that can be registered to internal native functions in `android_runtime` library. Several Java classes on this list seem security sensitive. Some examples are `shutdown` method of class `android.os.Power` and `getWifiState` method of class `androd.net.wifi.WifiManager`. If the API-level access control wants to restrict the access to the APIs in this class, it needs to prevent our attacks; otherwise, its access control can be circumvented.

## 5.2 Recommendations

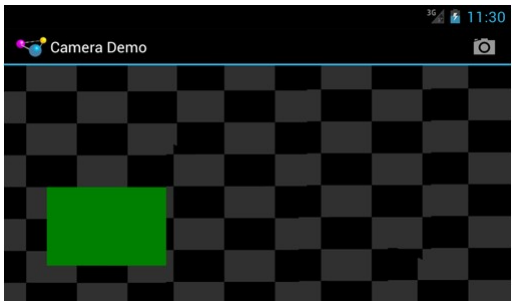
To cope with the problems caused by the class reloading, one possible way is to stop application's Java code from reloading preloaded Android core classes in any class loader instance. Digging deep into the call chain of the `findClass` method, which is used to perform class loading, we found that eventually `defineClass` native method in `DexFile` is invoked, and its corresponding native library function does the actual class loading task. Therefore, if API-level access control using bytecode rewriting is applied to prevent loading of Android classes, restriction on `BaseDexClassLoader`.

```
# ls -l photo.jpg
ls -l photo.jpg
photo.jpg: No such file or directory
# ls -l photo.jpg
ls -l photo.jpg
----rwxr-x system   sdcard_rw     7061 2012-11-28 23:30 photo.jpg
```

(a) Picture Saved

| Time | PID | TID | Application | Tag | Text |
|---|---|---|---|---|---|
| 11-28 23:30:33.589 | 588 | 588 | com.co... | Camera Class | Before invoking takePicture |
| 11-28 23:30:33.719 | 38 | 74 | | EmulatedCamera_Camera | takePicture |
| 11-28 23:30:33.859 | 588 | 588 | com.co... | Camera Class | After invoking takePicture |

(b) Logcat Output




(c) Screenshot

**Figure 8: Successful Attack on Camera App**

`loadClass` alone is not enough. All the invocations of methods within the call chain from `findClass` in the class `BaseDexClassLoader` to `loadClass` in `DexFile` should be restricted.

# 6. ACCESS SYSTEM SERVICE DIRECTLY

In Android, some of the implementation of the APIs are performed through the system process via inter-process communication. APIs invoke RPC stubs which initiates inter-process requests to ask system services to perform actual actions [17]. If applications can directly communicate with the system services without going through those APIs, the access control put on those APIs can be bypassed. The objective of this study is to find ways to achieve this. Figure 9 illustrates our strategy.

## 6.1 Exploit Customized RPC Stubs

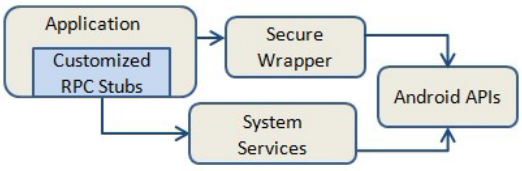



**Figure 9: Access System Services Directly**

Let us use location service as an example. Android provides an API `getLastKnownLocation` in the `android.location.LocationManager` class, and this API returns the last known location. The call to this API will be directed to `ILocationMangaer$Stub$Proxy`, which is the RPC stub to the system service `LocationManagerService`. If bytecode rewriting wants to put some fine-grained access control on the location access, it usually adds a layer of access control to this API. Unfortunately, there is a way to directly send a request to `LocationManagerService`, without going through the `getLastKnownLocation` API. To achieve this, the application can write its own RPC stub to communicate with `LocationManagerService`. We did a case study on geolocation app to demonstrate how exploit customized RPC stubs can bypass current bytecode rewriter.

## 6.2 Case Study

We evaluated our attack on a geolocation application which get last known location and display on the map.

```
1 import android.location.Location;
2 import android.location.LocationManger;
3
4 /* Return a handler to geolocation service */
5 LocationManager ser = (LocationManager)
      getSystemService(LOCATION_SERVICE);
6 /* Retrieve last know location */
7 Location loc = ser.getLastKnowLocation(...);
```

Suppose current bytecode rewriter enforced finer access control policy that the application can only retrieve location information when the location is within Alaska.

```
1 class SecureLocationManager extends
      LocationManager{
2 public Location getLastKnownLocation(...) {
3   Location loc = super.getLastKnownLocation(...);
4   if(loc.getLatitude()>60&&loc.getLatitude()<70&&
5      loc.getLongtitude()>140&&loc.getLongtitude()
          <160) {
6     return loc; }}
```

However, malicious app could introduce customized RPC with different method signature.

```
1 package my.location;
2 /* User-defined RPC stub class */
3 public interface LocMgr extends android.os.
      IInterface {
4     public static abstract class Stub extends
          android.os.Binder implements
5             my.location.LocMgr {...}}
```

```
1 import my.location.LocMgr;
2
3 IBinder b = android.os.ServiceManager.getService(
      LOCATION_SERVICE);
4 LocMgr sLocationManger = LocMgr.Stub.asInterface(
      b);
5 Location loc = sLocationManger
6         .getLastKnownLocation(...);
7 }
```

The above code snippet illustrates how customized RPC stub is used. Line 1 returns an IBinder instance, which is the gateway to the RPC interface. Line 2 establishes local clients by converting the IBinder instance to an instance of `LocMgr`, so that any invocation of inter-process calls are thereafter acted as regular method calls on `LocMgr`.

Note that `android.os.ServiceManager` is hidden in Android framework, which means the compiler would throw an error when an application tries to invoke any methods of it. At runtime `android.os.ServiceManager` is already loaded by DVM when application is started. However, one can cheat the compiler by writing a dummy `android.os.ServiceManager` to overcome compile time error.

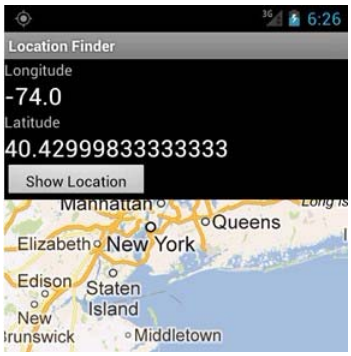



**Figure 10: Successful Attack on Geolocation App**

`my.location.LocMgr.getLastKnownLocation` has different method signature compared to that of `android.location.LocationManager.getLastKnownLocation`. Malicious application can escape access control check of current bytecode rewritier. Figure 10 demonstrates last known location is successfully retrieved by the application despite the location is at New York City.

## 6.3 Recommendations

The above problem is not difficult to fix; the easiest fix is to apply the API-level access control on `android.os.ServiceManager`'s `getService` API, so application's Java code cannot use this API to get system services. It is very rare that generic application would need to create custom client for system services, since rich android APIs are already provided to handle communications with system services. If we do not want to block this API, we can apply the bytecode rewriting technique to `android.os.Binder`.

# 7. RELATED WORK

**Fine-grained access control using bytecode rewriting.** Several researchers have proposed to implement fine-grained access control using bytecode rewriting on Android platform [11,14,19,22]. I-ARM embedded In-App Reference Monitors into Android application [14]. Via instrumenting the bytecode, security policies are interposed on all secure-sensitive API calls. Several other work [11,19,22] placed Reference Monitors in other Android services and substituted the calls to API methods with calls to other services. The work in [19, 22] removes all the assigned permissions from the applications, and reassigns the applications with fine-grained permissions. The executions of all secure-sensitive APIs will be delegated to the services. In-vivo uses another approach, it only uses the services to checks whether the application can invoke sensitive API calls or not [11], if it is approved, executions of APIs still take place inside the applications. Some researchers have proposed similar work on fine-grained access control using bytecode rewriting for Java platform [12, 16, 24]. Ajay et. al insert runtime checks into Java application code using rewriting [12]. Rudy and Wallach built three bytecode rewriting frameworks to add security semantics to the Java Virtual Machine [24]. Erlingsson has discussed practical applications of Java bytecode rewriting [16].

I-ARM pointed out some potential attacks on bytecode rewriting, i.e., usage of Java reflection and `ClassLoader` [14]. However, the main focus of the paper is on the implementation of a bytecode rewriting framework, analysis on potential attacks is quite brief. Our study is the first one presenting a systematical evaluation on API-level access control using bytecode rewriting. We did a comprehensive study on the attacking surfaces of the API-level access control using bytecode rewriting.

**Fine-grained access control through library interposition.** Instead of using bytecode rewriting, Aurasium accomplishes fine-grained access control via instrumenting Android Bionic libraries [26]. It encapsulates the original Bionic libraries functions inside its interposition routines, so fine-grained security policies can be enforced. These new routines will be invoked when applications invoke the routines in the Bionic libraries. This is achieved through the dynamic linking mechanism used by the native libraries. Compared to bytecode rewriting, this technique is more secure, as it does not worry about whether applications can directly jump to native library functions. However, by conducting access control at the lower level, it loses the rich context information that is available to bytecode rewriting. Such context is beneficial for enforcing more fine-grained security policies.

**Fine-grained access control through OS modification.** Other researches have modified the Android OS to monitor the data flow, detect data leakage and permission exploits [15, 18, 20, 28]. TISSA modified the Android OS to provide fine-grained protection on personal information [28]. TaintDroid performs taint analysis on applications and checks for privacy leakage [15]. Hornyack et al. modified the Android OS to replace private data with dummy data when the data are offered to applications [18]. It also blocks network transmissions if they contain data that are not supposed to be sent out. The main drawbacks of the OS-modification approach is the lack of flexibility and the need to update the phones.

**Bytecode rewriting for other purposes.** Several works use bytecode rewriting to instrument Android applications [23, 25], although not for enforcing API-level access control. Adsplit used bytecode rewriting to remove advertisement components to another advertisement service [25]. The work in [23] used bytecode rewriting to bind applications to fake Market app instead of the genuine one, so attacks can be launched in Google In-App Billing.

# 8. CONCLUSION AND FUTURE WORK

API-level access control using bytecode rewriting is a common technique used by existing work to provide fine-grained access control in Android. To fully understand how secure this technique is, we have conducted a systematic study on the effectiveness of the implementation of this technique. We have identified several new attacks. Although all the problems are fixable, our work manifests the need to perform more static analysis and dynamic checking should be performed to fulfill an effective API-level access control using bytecode rewriting. Our work can be beneficial to those who make use of bytecode rewriting or those who develop bytecode rewriting tools.

# 9. REFERENCES


[1] Android developer. http://www.developer.android.com/about/versions/android-4.0.3.html.

[2] Android reverse engineering honeynet project. http://www.honeynet.org/node/783.

[3] Binder. http://www.developer.android.com/reference/andrdoid/os/Binder.html.

[4] Bytecode for the dalvik vm. http://www.source.android.com/tech/dalvik-bytecode.html.

[5] Commonsware camera application. http://github.com/commonsguy/cw-omnibus.

[6] Jni tips. http://www.developers.android.com.guide/practices/jni.html.

[7] Naming a package. http://www.docs.oracle.com/javase/tutorial/package/namingpkgs.html.

[8] smali: An assembler/disassembler for android's dex format. http://www.code.google.com/p/smali.

[9] Swi handlers. http://www.infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/Cacdfeci.html.

[10] Android binder: Android interprocess communication. 2011.

[11] A. Bartel, J. Klein, K. Allix, Y. Traon, and M. Monperrus. Improving privacy on android smartphones through in-vivo bytecode instrumentation. *CoRR*, abs/1208.4536, 2012.

[12] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by java bytecode instrumentation. pages 1027–1040, 2001.

[13] E. Chin, A. P. Felt, K. Greenwood, and D. Wanger. Analyzing inter-application communication in android. *In Proceedings of the 9th International Conference on Mobile system, application and serivecs*, 2011.

[14] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *In IEEE Mobile Security Technologies*, 2012.

[15] W. Enck, P. Gilbert, B. 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, pp.1-6*, 2010.

[16] U. Erlingsson. The inlined reference monitor approach to security policy enforcement. 2004.

[17] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. *In Proceedings of the 18th ACM Conference on Computer and Communication Security*, 2011.

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

[19] J. Jeon, K. K. Micinski, and J. A. Vaughan. Dr. android and mr. hide: Fine-grained security policies on unmodified android. *Technical Report, Department of Computer Science, University of Maryland*, 2011.

[20] M. Nauman, S. Khan, and X. Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. *in Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[21] D. Poo, D. Kiong, and S. Ashok. Object-oriented programming and java, 2nd edition. 2007.

[22] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster. Application-centric security policies on unmodified android. *UCLA Computer Science Department, Technical Report*, 2011.

[23] D. Reynaud, E. C. R. Shin, T. R. Magrino, E. X. Wu, and D. Song. Freemarket: Shopping for free in android applications. *In Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[24] A. Rudys and D. S. Wallach. Enforceing java run-time properties using bytecode rewriting. 2002.

[25] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. *CoRR,abs/102.4030*, 2012.

[26] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. *In Proceedings of the 21st USENIX Security Symposium*, 2012.

[27] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detection malicious apps in official and alternative android markets. *In Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[28] Y. Zhou, X.Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications(on android). *Trust and Trustworthy Computing, pp.93-107*, 2011.


# APPENDIX

# A. LIBRARIES NOT PRELOADED

The following are shared native libraries provided by the Android platform. These libraries are stored under the directory /system/lib, and default class loader is not aware of their existence. Applications can reload them during runtime. If bytecode rewriting puts any access control on Java native APIs in order to restrict applications' access to the native library functions in these libraries, the access control can be bypassed using our attacks.

```
libEGL.so,  libETC1.so,  libFFTEm.so
libGLESv1_CM.so,  libGLESv1_enc.so
libGLESv2.so,  libGLESv2_dbg.so
libOpenSLES.so,  libOpenglSystemCommon.so
libRS.so,  libSR_AudioIn.so
libWnnEngDic.so, libWnnJpnDic.so
lib_renderControl_enc.so,  libandroid.so
libandroid_runtime.so,  libandroid_servers.so
libaudioeffect_jni.so,  libaudioflinger.so
libbcc.so,  libbcc.so.sha1
libbcinfo.so,  libbinder.so
libc.so,  libc_malloc_debug_leak.so
libc_malloc_debug_qemu.so,  libcamera_client.so
libcameraservice.so,  libclcore.bc
libcrypto.so,  libctest.so
libcutils.so,  libdefcontainer_jni.so
libdiskconfig.so,  libdl.so
libdrm1.so,  libdrm1_jni.so
```

libdrmframework.so, libdvm.so
libeffects.so, libemoji.so
libexif.so, libexpat.so
libext4_utils.so, libfilterfw.so
libfilterpack_imageproc.so
libgabi++.so, libgui.so
libhardware.so, libhardware_legacy.so
libharfbuzz.so, libhwui.so
libicui18n.so, libicuuc.so
libinput.so, libjni_latinime.so
libjni_mosaic.so, libjnigraphics.so
libjpeg.so, liblog.so
libm.so, libmedia.so
libmediaplayerservice.so
libmtp.so, libnativehelper.so
libnetutils.so, libnfc_ndef.so
libpagemap.so, libpixelflinger.so
libpower.so, libpowermanager.so
libreference-ril.so, libril.so
libsensorservice.so, libskia.so
libsonivox.so, libspeexresampler.so
libsqlite.so, libsqlite_jni.so
libsrec_jni.so, libssl.so
libstagefright.so
libstagefright_amrnb_common.so
libstagefright_foundation.so
libstagefright_omx.so
libstagefright_soft_aacdec.so
libstagefright_soft_amrdec.so
libstagefright_soft_g711dec.so
libstagefright_yuv.so
libstdc++.so, libstlport.so
libsurfaceflinger.so
libsurfaceflinger_client.so
libsystem_server.so
libsysutils.so, libthread_db.so
libttscompat.so, libttspico.so
libui.so, libusbhost.so
libutils.so, libvariablespeed.so
libvorbisidec.so
libwebrtc_audio_preprocessing.so
libwilhelm.so, libwnndict.so
libwpa_client.so, libz.so

## B. CLASSES LINKED TO ANDROID_RUNTIME

The following are Android framework Java classes that link to the native library functions in the android_runtime library. If an API-level access control is put upon these classes to restrict the applications' access of their linked native library functions, our attacks can bypass this access control by directly invoking those native library functions.

/android/debug/JNITest
/com/android/internal/os/RuntimeInit
/android/os/SystemClock
/android/util/EventLog
/android/util/Log
/android/util/FloatMath
/android/text/format/Time
/android/pim/EventRecurrence
/android/content/AssetManager
/android/security/Md5MessageDigest
/android/text/AndroidCharacter
/android/text/AndroidBidi
/android/text/KeyCharacterMap
/android/os/Process
/android/os/Binder
/android/view/Display
/android/nio/utils
/android/graphics/PixelFormat
/android/graphics/Graphics
/android/view/Surface
/android/view/ViewRoot
/com/google/android/gles/jni/EGLImpl
/com/google/android/gles/jni/GLImpl
/android/opengl/jni/GLES10
/android/opengl/jni/GLES20
/android/graphics/Bitmap
/android/graphics/BitmapFactory
/android/graphics/BitmapRegionDecoder
/android/graphics/Camera
/android/graphics/Canvas
/android/graphics/ColorFilter
/android/graphics/DrawFilter
/android/graphics/Interpolator
/android/graphics/LayerRasterizer
/android/graphics/MaskFilter
/android/graphics/Matrix
/android/graphics/Movie
/android/graphics/NinePatch
/android/graphics/Paint
/android/graphics/PorterDuff
/android/graphics/Rasterizer
/android/graphics/Region
/android/graphics/Shader
/android/graphics/Typeface
/android/graphics/Xfermode
/android/graphics/YuvImage
/com/android/internal/graphics/NativeUtils
/android/database/CursorWindow
/android/database/SQLiteCompiledSql
/android/database/SQLiteDatabase
/android/database/SQLiteDebug
/android/database/SQLiteProgram
/android/database/SQLiteQuery
/android/database/SQLiteStatement
/android/os/Debug
/android/os/FileObserver
/android/os/FileUtils
/android/os/MessageQueue
/android/os/ParcelFileDescriptor
/android/os/Power
/android/os/StatFs
/android/os/SystemProperties
/android/os/UEventObserver
/android/net/LocalSocketImpl
/android/net/NetworkUtils
/android/net/TrafficStats
/android/net/wifi/WifiManager
/android/nfc/NdefMessage
/android/nfc/NdefRecord
/android/os/MemoryFile
/com/android/internal/os/ZygoteInit
/android/hardware/Camera
/android/hardware/SensorManager
/android/media/AudioRecord
/android/media/AudioSystem
/android/media/AudioTrack
/android/media/ToneGenerator
/android/opengl/classes
/android/bluetooth/HeadsetBase
/android/bluetooth/BluetoothAudioGateway
/android/bluetooth/BluetoothSocket
/android/bluetooth/ScoSocket
/android/server/BluetoothService
/android/server/BluetoothEventLoop
/android/server/BluetoothA2dpService
/android/server/Watchdog
/android/message/digest/sha1
/android/ddm/DdmHandleNativeHeap
/android/backup/BackupDataOutput
/android/backup/FileBackupHelperBase
/android/backup/BackupHelperDispatcher
/android/content/res/ObbScanner
/android/content/res/Configuration