On the Effectiveness of Code-reuse-based Android Application Obfuscation

Xiaoxiao Tang[1], Yu Liang[2], Xinjie Ma[3], Yan Lin[1], and Debin Gao[1]

[1] Singapore Management University, Singapore, Singapore
{xxtang.2013, yanlin.2016, dbgao}@smu.edu.sg
[2] Wuhan University, Wuhan, China
liangyu@whu.edu.cn
[3] Nankai University, Tianjin, China
mxjnkcs@nankai.edu.cn

Abstract. Attackers use reverse engineering techniques to gain detailed understanding of executable for malicious purposes, such as re-packaging an Android app to inject malicious code or advertising components. To make reverse engineering more difficult, researchers have proposed various code obfuscation techniques to conceal purposes or logic of code segments. One interesting idea of code obfuscation is to apply code-reuse techniques (e.g., Return-Oriented Programming) to (re-)distribute essential code segments before they are reconstructed at runtime. Such techniques are well understood on x86 platform, but relatively less explored on Android. In this paper, we present an evaluation on the extent to which code-reuse-based techniques can be applied to obfuscate Android apps. Moreover, we extend code-reuse-based obfuscation to the Android platform by proposing an obfuscation mechanism for both Java and native code. Results show that 835 gadgets are found in the C standard library (`libc.so`) which cover the entire Turing complete set. Furthermore, we implement a semi-automatic tool named AndroidCubo and show that it protects both Java and native code with comparable security to those obfuscated with Java reflection at a small runtime overhead.

Keywords: Obfuscation, Android application, Code reuse, Java Native Interface

1 Introduction

Android is now the most popular mobile operating system with more than 80% market share. The number of apps available on Google Play has climbed to more than 2 million. The popularity of Android operating system and applications also invites lots of pirated apps. In order to produce pirated apps, adversaries typically analyze benign apps with reverse engineering tools [1–3], modify the app to bypass verification algorithms, if any, and then re-package the apps with injected malicious code or advertising components.

To make such attacks more difficult, app developers apply code obfuscation techniques. Google recommends developers to use Proguard [4] to obfuscate sensitive code in Android. However, this tool only obfuscates Java code and leaves native code as easy targets of attackers. Moreover, Proguard, like many traditional Java obfuscation techniques [5–7], only applies relatively simple obfuscation techniques, e.g., rename identifiers and remove debugging information. Although identifiers of classes and methods are no longer understandable after the obfuscation, names of the system APIs and the control flow of the program still enable reverse engineering to a great extent. For example, it is easy to figure out important functionality of an app by analyzing the system APIs invoked.

Return-Oriented Programming (ROP), which belongs to the bigger family of code-reuse-based techniques, was recently proposed as an attacking technique to exploit vulnerable programs [8–12]. It was subsequently used for code protection [13–15] and to provide program steganography, e.g., RopSteg [14]. The main idea of code-reuse-based obfuscation is to replace essential code with small code pieces distributed in the app and to reconstruct the essential code dynamically. These small code pieces, typically ending with return/return-like instructions, are called gadgets. Then, a payload, which contains addresses of the gadgets and parameters needed by them, is generated for code reusing. This payload is typically used to trigger some vulnerability (e.g., buffer overflow) and to invoke the hidden code by executing the selected gadgets one by one. With this technique, the semantics of the essential code in the original program are hidden in the payload. As part of the data in an app, payload is safer than the original code under the disclosure of reverse engineering tools. The hidden code can be further protected through dynamically downloading the payload from a trusted remote server. In addition to protecting benign code, this technique can also be used for hiding malicious behaviors by adversaries.

However, RopSteg and other code-reuse-based techniques cannot be directly applied to Android applications. First, Android apps are mainly developed in Jave, while code-reuse-based techniques are based on native binaries typically compiled from C/C++. Second, Android devices are built on ARM architecture on which registers are used for parsing function parameters and saving return addresses [16], as opposed to x86 which is more dependent on the stack.

In this paper, we present the first evaluation on the extent to which code-reuse-based techniques can be applied on Android application obfuscation. Moreover, we propose an effective code-reuse-based obfuscation mechanism for Android apps. This mechanism helps developers to obfuscate small pieces of sensitive code, including both Java and native code. We evaluate gadgets found in binaries of Android apps and calculate the amount of gadgets in several common native libraries used by Android apps. Results show that 835 gadgets in the C standard library (`libc.so`) cover a Turing complete gadget set. We implement this idea in a tool called AndroidCubo (Android Code-reuse Based Obfuscation) and successfully apply it on real examples to protect both Java and native code with a small overhead. We show that the security of our obfuscated code is comparable to that obfuscated with Java reflection.

2 Overview

Android app obfuscation focuses on preventing reverse engineering by adversaries. We assume a threat model in which an adversary reveals essential code in Android apps with reverse engineering tools, such as Apktool and APKstudio. These tools help adversaries decompile Android APK and disassemble the resources to Jave or assembly code. Then, adversaries can tamper the decompiled app and repackage it to perform malicious behaviors. Obviously, we assume that source code of the Android app is not available to the adversary.

An effective obfuscation technique has to achieve two goals when targeting Android applications. First, it should protect the compiled essential code from being reverse engineered to a human understandable format. Second, it should be generally applicable to any code segments to be hidden on any Android applications. In the context of code-reuse-based techniques, this means that a Turing complete gadget set that consists of frequently appeared gadgets is needed.

Fig 1 gives an overview of our code-reuse-based technique in obfuscating the essential code in an Android app. First, the essential code is replaced with a gadget sequence based on the Turing complete gadget set. The gadget sequence represents the semantics of the essential code and is also regarded as the code reuse program. Next, we prepare a payload according to the gadget sequence. After that, a segment of trigger code is embedded in the app to invoke the protected code at runtime. At last, when the protected app is running, the payload will be loaded into the memory of the app and passed to the trigger code for invoking the protected code.

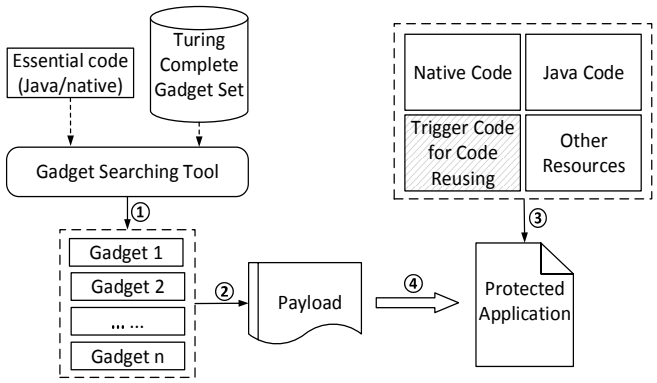

Fig. 1: An overview of our code-reuse-based obfuscation technique for Android apps.

The Turing complete gadget set is a fundamental requirement in this technique for providing enough gadgets to substitute the essential code. In the following sections, we first present our analysis of gadgets on ARM and then discuss the details of the code obfuscation mechanism.

3 Turing Complete Gadget Set

As we discuss in the earlier section, having a Turing complete gadget set is a necessary condition for a code-reuse-based obfuscation technique to be generally applicable to most Android applications. In this section, we present a Turing complete gadget set found available for code reuse obfuscation on ARM architecture. We also analyze the number of gadgets in each category. We focus our analysis on Android 4.4 on a Nexus 5 handset. In the following description, `Ra-Rd` and `Rx-Ry` denote different registers of ARM.

Previous studies [12, 17] applied gadgets ending with `BLX Ra` in their code-reuse techniques. `BLX Ra` is an indirect jump instruction whose jump destination is specified by register `Ra`. Unlike return instructions, `BLX` cannot fetch gadget addresses from memory. Thus, a specific kind of gadget, called update-load-branch (ULB) gadget, is used to sequentially fetch gadget addresses to registers and chain the gadgets together. However, the ULB gadget is very hard to find in native libraries [12]. Besides that, this strategy doubles the length of the gadget sequence, which makes code-reuse-based obfuscation techniques more complicated and slows down the program. Hence, we explore the possibility in using another type of gadgets that ends with `POP {Rx-Ry, PC}`. This `POP` instruction loads an address from the stack to the program counter register `PC` directly. It always appears in the epilogue of a function and is more commonly found in native libraries than the `BLX` instruction.

Our gadget searching strategy is to look for basic blocks (instruction sequences that do not contain branches) ending with a `POP {Rx-Ry, PC}` instruction to minimize the effort needed to handle branches in instruction sequences and payload generation. We implement this strategy into a gadget searching tool in python. This tool searches for all available gadgets and their relative addresses in native libraries. It also categorizes the available gadgets to different classes according to their functionality.

We apply our gadget searching tool on several commonly used native libraries used by Android apps and compare the number of gadgets in our gadgets set with that in the gadget set proposed by Davi et al. [12], see Table 1. The results show that number of gadgets in our gadget set is much larger than that used by Davi et al. [12]. This is because `POP {Rx-Ry, PC}` is more frequently used than `BLX Ra` in the native libraries. With the larger number of gadgets, the probability of finding all gadgets needed in the Turing complete gadget set is higher. Besides that, the more gadgets we find, the more flexibility we have for essential code replacement.

Table 1: Number of gadgets found in different gadget sets.

Native Libraries	`libc`	`libruntime`	`libunity`	`libvideo`	`libcocos2d`
# of Gadgets (Our Gadget Set)	835	2,244	21,483	317	12,913
# of Gadgets (Gadget Set in [12])	77	1,326	10,734	148	6,126

Upon our analysis, we realized that gadgets that implement basic operations, such as memory operations, arithmetic, and logic operations, can be easily found through searching the corresponding instructions. Other functionality, including control-flow transfers and function calls, need to be constructed carefully. We carefully analyzed the gadget sets found and managed to form a Turing complete gadget set for converting sensitive code into gadget sequences, see Table 2.

Table 2: Number of different types of gadgets in our gadget set.

Gadget Functionality	`libc`	`libruntime`	`libunity`	`libvideo`	`libcocos2d`
Load	127	151	2,484	60	1,607
Store	227	161	5,518	77	2,333
Add	20	3	878	23	204
Sub	30	1	78	3	35
Shift	12	8	20	2	689
And	6	8	137	3	60
Or	21	6	274	3	100
Xor	2	2	31	0	22
Unconditional Branch	226	753	12,063	84	3,035
Conditional Branch	28	15	1,107	29	29
Function Call	8	187	865	5	458

The results show that libraries contain sufficient gadgets in each category of the Turing complete gadget set, with the exception of `libvideo` where there is no gadgets to perform `xor` operation. However, also note that `xor` could be indirectly implemented with other logical operators. This shows that many commonly used libraries are sufficient for providing gadgets for code-reuse obfuscation.

4 Code Obfuscation

With the Turing complete gadget set found in various native libraries covering different functionality, we now present details of the obfuscation mechanism for protecting a piece of essential code in an Android application. The code protection process, as shown in Figure 1, consists of a few steps in 1) replacing the sensitive code with our gadget sequence; 2) generating code-reuse payload according to the gadget sequence; and 3) constructing trigger code to invoke the hidden code with payload in the app.

4.1 Essential Code Replacement

It is usually straightforward to replace the essential code to be obfuscated with gadget sequences. Most code-reuse techniques typically disassemble the essential code to instruction sequences first, and then substitute them with semantically

equivalent gadgets. However, dealing with Android applications makes this process more complicated as we want to be able to obfuscate both the native and Java code. This makes our code-reuse-based obfuscation tool different from most existing ones.

For Android apps, native code is always compiled to native libraries (`.so` file) by the building module of Android Native Development Kit (NDK). Reverse engineering tools, such as IDAPro, Hopper, or the GNU Project debugger (GDB) can be used to disassemble the native libraries and to obtain the instruction sequences for the essential code to be obfuscated. We can then substitute instructions in the essential code with gadgets in the native binaries of the app. Since most of these native libraries contain Turing Complete gadget sets as shown in Table 2, we will always be able to perform this substitution successfully.

Dealing with Java code in Android apps is more challenging, since existing code-reuse techniques only support native code. Although a subset of the language-independent functionality (e.g., concatenation of strings can be implemented in Java as `+` operator and native code as `strcat()` method) can be implemented in native code as well, other functionality that uses classes or methods specifically provided by Java or Android cannot be directly implemented in native code (e.g., enable bluetooth can only be implemented in Java as `BluetoothAdapter.enable()`).

Fortunately, the Java Native Interface (JNI) provides a flexible connection for the communication between Java and native code [18]. JNI provides several native methods for accessing object's field from native code as well as methods for converting Java classes to native classes, including `GetObjectClass()`, `GetMethodID()` and `CallVoidMethod()`. These methods allow native code to use Java class objects and to call Java methods by providing corresponding class names and method names. In addition, JNI also provides methods to convert Java objects to native variables. For example, `GetStringUTFChars()` can be used to convert a Java string to native chars.

Fig 2 shows an example of the corresponding native code that can be used to replace a sensitive Java API `sendTextMessage()`. In this example, The JNI function `CallVoidMethod()` will call the sensitive API in native code after retrieving the class and method names.

In addition to the proposed method of implementing Java functionality in native code via JNI and then subsequently obfuscating the resulting native code, here we propose another method using shell command. We notice that many Java operations can be represented with shell commands in Android apps, e.g., reading SMS can be implemented through shell command `content query --uri content://sms`. Therefore, we propose to obfuscate Java code by first replacing it with a call to `system()` with the corresponding shell command, and then subsequently obfuscating the calling of `system()` with our code-reuse program. This method only needs two gadgets — the first one to move the address of the corresponding command to register `R0`, and the second to invoke the system call function. The actual shell command appears as parameters to the system call.

```
void * sendSMS(JNIEnv *env)
{
    jclass smsclass = env->FindClass("android/telephony/SmsManager");
    jmethodID get = env->GetStaticMethodID(smsclass, "getDefault", "()Landroid/telephony/
        SmsManager;");
    jobject sms = env->NewObject(smsclass, get);
    //Obtaining sendTextMessage()
    jmethodID sendMethod = env->GetMethodID(smsclass, "sendTextMessage",
    "(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/PendingIntent;
        Landroid/app/PendingIntent;)V");
    jstring destAddress = env->NewStringUTF("1234567890"); //Phone number
    jstring text = env->NewStringUTF("native"); //SMS content

    //Sending SMS with sendTextMessage() in native code
    env->CallVoidMethod(sms, sendMethod, destAddress, NULL, text, NULL, NULL);
}
```

Fig. 2: The native code of calling sendTextMessage() with JNI.

Table 3 presents some common behaviors which can be represented by shell commands on Android. These commands are all feasible to be used on normal Android devices. The available shell commands can be found under the directory `/system/bin` in the corresponding Android devices. More complicated operations can be hidden in shell scripts written with available commands and be invoked through executing the scripts with `system()`. These shell commands include simple ones like file operations, process management, network configuration, as well as those provided by Android Debug Bridge (ADB) for activity management and package management.

Table 3: Examples of operations on Android and the corresponding shell commands.

Operations	**Shell Command**
Open Messenger	`am start --user 0 -a android.intent.action.SENDTO -d sms:PHONE_NUMBER --es sms_body MESSAGE`
Read SMS	`content query --uri content://sms`
Open Dialer	`am start --user 0 -a android.intent.action.DIAL -d tel:PHONE_NUMBER`
Start Browser	`am start --user 0 -a android.intent.action.VIEW -d URL`
Create Directory	`mkdir DIRECTORY_PATH`

4.2 Payload Generation

The main advantage of code-reuse-based obfuscation tools over other obfuscation techniques is that the hidden code exists in the form of data rather than instructions. To achieve this, we need to prepare a payload according to the gadget sequence. Payload is a segment of memory content that contains semantics

of the protected code and will be used for overwriting control data at runtime. A payload typically consists of three parts. The first part is the data that will be used to overwrite control data in memory to redirect control flow to the hidden code. The second part consists of the parameters and addresses for the gadget sequence which presents the semantics of the hidden program. The third part is a segment of buffer with data needed by the code reuse program and other padding data. Fig 3 is an example of the payload which has been loaded on the stack.

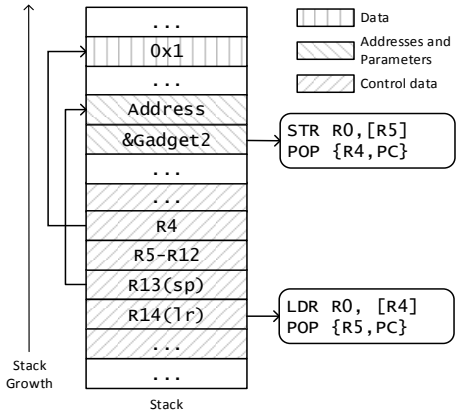

Fig. 3: Layout of the payload. The shadowed areas present different parts of the payload.

This payload is used for the gadget sequence that loads a number `0x1` from memory and stores it at another address. From bottom to top of the stack, the first part is the data that overwrites control data `jmp_buf` which is used to set register values of the execution environment. In the rewritten `jmp_buf`, `R4` is set to the address of `0x1` and stack pointer is set to the beginning of the second part of the payload. The second part contains the parameter needed by the first gadget and the address of the second gadget. The last part of the payload contains other data — the number `0x1` to be loaded from memory and stored to the address specified by `R5`. To generate the payload, the most essential steps are store the address of the first gadget in `lr` and addresses of following gadgets on the stack. Thus, by changing `sp`, the gadgets will be executed in proper order.

4.3 Code Triggering

After preparing the payload, extra code needs to be added to the app as an entry point of the hidden code. This part of the code fetches the payload at runtime and uses it to trigger the code-reuse program. Code-reuse programs are commonly triggered through overwriting control data, including return addresses,

function pointers, and jump buffer. The overwriting could be based on a set of vulnerable library functions that lack boundary checking, such as `gets()`, `fread()`, `strcpy()`, and `sprintf()`. As in some existing work [12], the control data we choose to overwrite is the `jmp_buf` structure that is used to restore the execution environment in exception handling. The `jmp_buf` structure contains data that will be used to set values of registers which are used for storing parameters and the return address of a function call. Thus, it is convenient to redirect the control flow through overwriting `jmp_buf` structure on ARM.

Fig 4 shows an example of overwriting `jmp_buf` [12]. In this piece of code, function `setjmp()` and `longjmp()` are used to store and restore the execution context in variable `jbuf`. Reading data from `sFile` to `buf` will overwrite `jbuf`. Thus, `longjmp()` will direct the program execution to somewhere specified by the overwritten `jbuf`.

```
1  typedef struct foo{
2      char buf[JP_BUFSIZE];
3      jmp_buf jbuf;
4  }FOO, *PFOO;
5  PFOO f;
6
7  void * overflow(char * filePath)
8  {
9      int i;
10     ... ...
11     i = setjmp(f->jbuf);
12     fread(f->buf, 1, BUFSIZE+256, sFile);
13     ... ...
14     longjmp(f->jbuf, 2);
15     ... ...
16 }
```

Fig. 4: Trigger code to be added to source code of the application.

4.4 Payload Protection

Since the semantics of the essential code are hidden in the code-reuse payload, it is important that our obfuscation tool provides protection on the payload to resist and reverse engineering attempts. To protect the payload, we propose three possible solutions.

- Instead of storing payload as static resources of the Android app, the payload can be embedded in the resources using information hiding techniques. For example, the payload can be hidden in a segment of normal code, e.g., as an image, using steganography [19].
- The payload can exist in an encrypted form of data in the Android app, and be decrypted at runtime.

– To completely remove the payload from the APK file of the Android app, we can dynamically download it from a trusted remote server [15]. Dynamically, the app will request and receive payload from the server based on a reliable protocol.

In this work, we use the last, and the most secure, method.

5 Implementation and Case Studies

We manage to implemente our idea of obfuscating Android application as a tool set, AndroidCubo. AndroidCubo takes as input the source code of an Android app and obfuscates selected native and Java code in it. We present some implementation details and applications of AndroidCubo on an app in this section. Experiments were performed on a Nexus 5 running Android 4.4.

5.1 Implementation details

Code-reuse programming is complicated since it involves a lot of low level operations on memory and registers. We implement AndroidCubo as a tool set for helping Android app developers to obfuscate sensitive code with code-reuse technique. It contains a source code template to be inserted into the Android source code and a payload maintainer to execute on a trusted server.

The source code template contains a Java class named `ObfuscateUtil` and a C program named `Hiding`. The class `ObfuscateUtil` provides native interfaces for calling native methods in `Hiding`. It also implements network communication with the trusted server which maintains the payload for the code-reuse program. The `Hiding` program has a method named `trigger()` that uses the payload (received from communication with the trusted server) to trigger the obfuscated code.

This source code template can be directly added to the Android project for obfuscating a segment of sensitive code. The only additional code a developer has to add is for preparing parameters if they are obfuscating API calls. To use this template for obfuscating multiple segments of sensitive code, the user needs to add trigger methods in `Hiding` and the corresponding interfaces in `ObfuscateUtil`.

The payload maintainer on the server side has two parts. The first part is a payload generator that works in the following manner.

– **Native code obfuscation** Our gadget searching tool lists available gadgets and their relative addresses for the developer to construct the gadget sequence. The developer can also use other existing tools, e.g. ROPgadget [20] or Q [21], to develop their code reuse program.
– **Java code obfuscation through shell commands** The generator automatically generates the payload with a command provided by the user.
– **Java API obfuscation** The developer specifies the addresses of the API and the corresponding parameters and our generator outputs the payload.

The second part is a program for sending payload to the app. This program is developed with PHP with which the server will handle the request of payload from the app, trigger the payload generator, and then send the payload over to the app.

5.2 Case study: Obfuscating Native Code

To demonstrate AndroidCubo in obfuscating native code, we hide a simple comparison algorithm as shown Fig 5(a)(b). This algorithm obtains and stores the larger one of the two input numbers. As described in Section 4, this simple algorithm needs to be converted to a sequence of gadgets first. AndroidCubo first executes the gadget searching tool and finds available gadgets and their relative addresses, and then generates a sequence of gadgets to substitute the original code as shown in Fig 5(c). In this sequence, gadgets 1-3 are used to load the first operand to register `R9`. Gadgets 4-6 are used to load the second operand to register `R3`. The last conditional gadget is used to find and store the larger number.

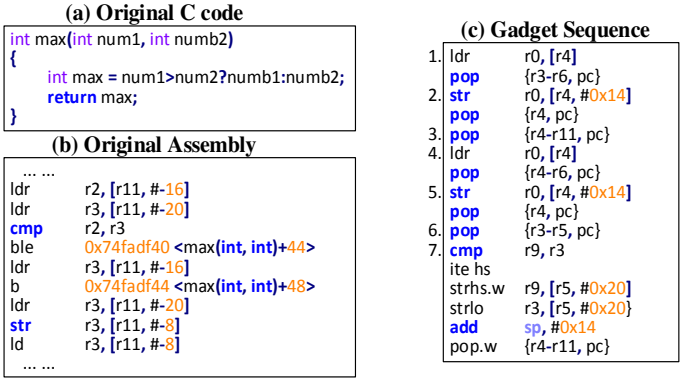

Fig. 5: Source code to be hidden and the corresponding gadget sequence. (a) Original C code; (b) Original assembly code; (c) Gadget sequence.

AndroidCubo then generates the payload based on the gadget sequence. In particular, the first part of the payload is the data used to overwrite the control data `jmp_buf`. `jmp_buf` directs the stack pointer to the beginning of the second part — the addresses and parameters of the gadgets. `LR` is then set to the address of the first gadget. The last part of the payload is a buffer containing junk data.

We recompile the Android app with outputs from AndroidCubo and execute the app with the corresponding payload. After executing the app and loading the payload to the stack, `longjmp()` successfully executes with the prepared `jmp_buf`, and the gadget pointed to by `LR` executes followed by other gadgets prepared in the payload.

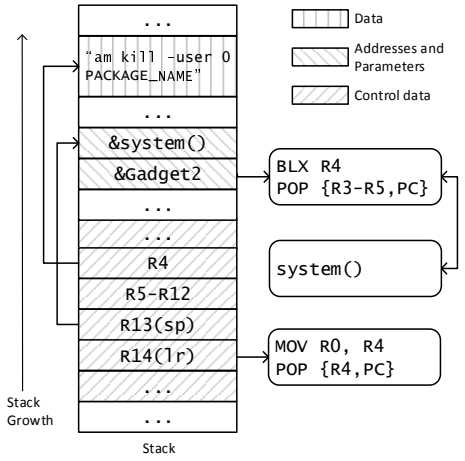

Fig. 6: Stack layout after loading the payload.

5.3 Case study: Obfuscating Java Code

We use another example to demonstrate using AndroidCubo to obfuscate Java code. In this example, we hide the Java code that kills a background process.

The operation of killing a background process is typically implemented by obtaining an `ActivityManager` object and killing the process by calling the method `killBackgroungProcess()` in Java. AndroidCubo hides this Java code through a shell command `am kill --user 0 PACKAGE_NAME` with two gadgets. The first gadget `MOV R0, R4; POP {R4, PC}` is used to prepare the shell command as a parameter for `system()`. The second gadget is a function call gadget `BLX R4; POP {R3-R5, PC}` to invoke the shell command. Fig 6 presents a view of the stack after our app loads the payload generated by AndroidCubo to overwrite a buffer.

From bottom to top of the stack, the three shadowed areas present the corresponding parts of the payload. The first part is the overwriting of control data `jmp_buf`. In `jmp_buf`, register `LR` is set to the address of the first gadget. Function pointer `SP` is set to the beginning of the second part of the payload. Register `R4` is set to the address of the command that will be assigned to `R0` as the parameter of `system()`. The second part is the gadget addresses and parameters. The most essential data on this part is the address of `system()` and the address of the second gadget. The last part includes the padding data and the command string needed by `system()`.

5.4 Overhead

In our experiments in applying AndroidCubo to the Android apps, it introduces around 150 LOC to native part and around 250 LOC to Java part of the Android application.

6 Comparison with other Obfuscation Techniques

There have been existing obfuscation techniques proposed, and in this section, we conduct a comparative test on sensitive API obfuscation among code-reuse-based method and other techniques, including control-flow obfuscation and Java-reflection-based obfuscation. Control-flow obfuscation techniques typically hides or protects the selected code by branching or looping garbage code. Java-reflection-based techniques typically hide sensitive API calls by using Java reflection to access the APIs through their names. We use these techniques to obfuscate an open source application named `OverFlow`. The sensitive API that we target to obfuscate is `sendTextMessage()`.

6.1 The Experiment

We obfuscate the target app with all three techniques and then build the signed APK file. We use Apktool [1], dex2jar [3], and JD-GUI [22] to reverse engineer the APk files obtained to see how much information of the sensitive API can be reconstructed. Apktool is used to unpack the APK file and obtain the dex file. dex2jar converts the dex file to jar files which contain the byte code of the app. After obtaining the jar file, we extract the class files in the jar and use JD-GUI to reverse engineer class files to readable Java code. The above constitutes the most commonly used methods for reverse engineering Android apps.

6.2 Reverse Engineering Results

Fig. 7 presents the reverse engineering output for the un-obfuscated app (Fig. 7(a)) and apps obfuscated by the three different techniques (Fig. 7(b)-(d)).

Although the control flow recovered in Fig. 7(b) seems opaque, it is easy to spot out the sensitive API call from the byte code at line 9. This shows that the control-flow obfuscation manages to introduce confusion in terms of how control transfers, but it fails to hide the existence of Java API call. From Fig. 7(c), we can also easily figure out the name of the API from the first parameter of `getMethod()`.

Fig. 7(d), on the other hand, substitutes the sensitive API call with a native function call whose functionality cannot be inferred from the name. That said, one could further analyze the native function `CallVoidMethod()` to see if it contains any hints of the API function to be called. We use IDAPro to reverse engineer the native function `CallVoidMethod()`, and find that the string `sendTextMessage` and `(Ljava/lang/String;...)V` can be recovered from the binaries.

6.3 Discussion

In our experiments of obfuscating the Android app with different obfuscation methods, AndroidCubo presents better security in hiding the sensitive API

```
private void sendMessage(String paramString1, String paramString2)
{
    try
    {
        SmsManager.getDefault().sendTextMessage(paramString1, null, paramString2, null,
        null);
        return;
    }
    catch (Exception paramString1)
    {
        paramString1.printStackTrace();
    }
}
```

(a) Decompiled code of un-obfuscated sendTextMessage()

```
... ...
// 131: goto -7 -> 124
// 134: aload 4
// 136: aload_1
// 137: aconst_null
// 138: aload_2
// 139: aconst_null
// 140: aconst_null
// 141: invokevirtual 105 android/telephony/SmsManager:sendTextMessage ...
// 144: return
// 145: astore_1
// 146: aload 5
// 148: astore_2
... ...
```

(b) Decompiled code of function call obfuscated by control-flow obfuscation

```
private void sendMessage(String paramString1, String paramString2)
{
    try
    {
        SmsManager localSmsManager = SmsManager.getDefault();
        localSmsManager.getClass().getMethod("sendTextMessage", new Class[] { String.class,
         String.class, String.class, PendingIntent.class, PendingIntent.class }).invoke(
        localSmsManager, new Object[] { paramString1, null, paramString2, null, null });
        return;
    }
    catch (Exception paramString1)
    {
        paramString1.printStackTrace();
    }
}
```

(c) Decompiled code of function call obfuscated by Java Reflection

```
private void sendMessage(String paramString1, String paramString2)
{
    try
    {
        nativeMethod(paramString1, null, paramString2, null, null);
        return;
    }
    catch (Exception paramString1)
    {
        paramString1.printStackTrace();
    }
}
```

(d) Decompiled code of function call obfuscated by AndroidCubo

Fig. 7: The decompiled code of calling sendTextMessage() and the decompiled code from obfuscated calling.

call from reverse engineering tools. At a high level, its idea is similar to Java-Reflection-based techniques in that both techniques replace the original Java call with another method call, and both techniques specify the underlying method to be called via a string. However, the replacement in Java-Reflection-based techniques is still a Java method call, which is relatively easy to analyze; on the other hand, AndroidCubo uses a replacement of native calls that are more difficult to analyze. Coupled with other string obfuscation techniques, we argue that AndroidCubo presents higher resilience in obfuscation compared to Java-Reflection-based techniques.

6.4 Limitations

Although applying the code-reuse-based obfuscation technique is feasible, there are a couple of limitations that are worth noting. First, AndroidCubo, in its current form, is a semi-automatic tool. Piecing together gadgets and writing long code-reuse programs are still a complicated process that requires the developer's attention and help. Second, applying code-reuse techniques for good, e.g., in obfuscating program logic, runs into the risk of being prohibited by code-reuse protection mechanisms. That side, current Android systems have no protection mechanisms to resist code-reuse programs, and advanced many techniques [23–26] are powerful enough to bypass most protection mechanisms.

7 Related Work

Traditionally, there have been three categories of obfuscation techniques proposed, including layout obfuscation [6], control-flow transformation [7, 27], and data obfuscation. Layout obfuscation [6] removes relevant information from the code without changing its behavior. Control-flow transformation [7, 27] alters the original flow of the application. Data obfuscation obfuscates data and data structures in the application. These techniques are certainly helpful in obfuscating Android apps; however, they are not specific to the Android platform and are especially weak in hiding code in Android apps.

There are also free or commercial obfuscation techniques specifically provided to Android developers. ProGuard [4] is a free and commonly used one that obfuscates the names of classes, fields, and methods. DexGuard [28] is a commercial optimizer and obfuscator. It provides advanced obfuscation techniques for Android development, including control-flow obfuscation, class encryption, and so on. DexProtector [29] is another commercial obfuscator that provides code obfuscation as well as resource obfuscation, such as the Android manifest file.

Code reuse techniques, including Return-into-lib(c) [30, 31], Return-oriented programming [8, 9] and Jump-oriented programming [10–12], are first proposed to exploit vulnerable apps by hijacking their control-flow transfers and constructing malicious code dynamically. Among these code-reuse techniques, only a few of them work on Android system or the ARM architecture. [12] proposes a systematic jump-oriented programming technique on ARM architecture. The

gadget set proposed in this work consists of gadgets ending with `BLX` instructions. In this paper, we use a different type of gadgets that are more commonly found in native libraries.

Recently, several code-reuse-based obfuscation techniques [13–15] have been proposed. One of the code-reuse-based obfuscation techniques is RopSteg — a steganography technique on x86 [14]. RopSteg protects binary code on x86 architecture, while our code-reuse-based obfuscation on Android platform works for both Java and native code on Android platform. Another work [15] proposes a malware named Jekyll which hides malicious code and reconstructs it at runtime. Our obfuscation mechanism can be used for protection of either malicious or benign code.

8 Conclusion

In this paper, we present a code-reuse-based technique for protecting Android applications. This technique enhances the concealment of both Java and native code in Android apps through hiding essential code. Our evaluation shows that the limited binary resources in Android apps are sufficient for applying code-reuse-based obfuscations. We further implement AndroidCubo semi-automate the process of obfuscating essential code. Examples present that it is practical to protect applications with AndroidCubo.

References

1. Winsniewski, R.: Apktool: A tool for reverse engineering android apk files. `http://ibotpeaches.github.io/Apktool/`
2. Vaibhavpandeyvpz: Apk studio. `http://www.vaibhavpandey.com/apkstudio/`
3. Alll, B., Tumbleson, C.: Dex2jar: Tools to work with android. dex and java. class files
4. Lafortune, E., et al.: Proguard. `http://proguard.sourceforge.net`
5. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand (1997)
6. Chan, J.T., Yang, W.: Advanced obfuscation techniques for java bytecode. Journal of Systems and Software **71**(1) (2004) 1–10
7. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1998) 184–196
8. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to risc. In: Proc. ACM CCS. (2008)
9. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. ACM CCS. (2007)
10. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proc. ACM ASIACCS. (2011)
11. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proc. ACM CCS. (2010)

12. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Return-oriented programming without returns on arm. System Security Lab-Ruhr University Bochum, Tech. Rep (2010)
13. Ma, H., Lu, K., Ma, X., Zhang, H., Jia, C., Gao, D.: Software watermarking using return-oriented programming. (2015)
14. Lu, K., Xiong, S., Gao, D.: Ropsteg: program steganography with return oriented programming. In: Proc. ACM CODASPY. (2014)
15. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on ios: When benign apps become evil. In: Proc. Usenix Security. (2013)
16. Seal, D.: ARM architecture reference manual. Pearson Education (2001)
17. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Information Security. Springer (2011)
18. Google: Jni tips. `http://developer.android.com/training/articles/perf-jni.html`
19. Morkel, T., Eloff, J.H., Olivier, M.S.: An overview of image steganography. In: ISSA. (2005)
20. Salwan, J., Wirth, A.: Ropgadget (2012)
21. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: USENIX Security Symposium. (2011) 25–41
22. Dupuy, E.: Jd-gui: Yet another fast java decompiler. URL http://java. decompiler. free. fr/? q= jdgui/accessed March (2012)
23. Carlini, N., Wagner, D.: Rop is still dangerous: Breaking modern defenses. In: USENIX Security Symposium. (2014)
24. Davi, L., Lehmann, D., Sadeghi, A.R., Monrose, F.: Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: USENIX Security Symposium. (2014)
25. Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G.: Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In: 23rd USENIX Security Symposium, San Diego, CA. (2014) 417–432
26. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. IEEE Symposium on Security and Privacy, IEEE (2013)
27. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. ACM CCS. (2012)
28. : Dexguard. `https://www.guardsquare.com/dexguard`
29. : Dexguard. `https://dexprotector.com/`
30. Nergal: The advanced return-into-lib(c) exploits (pax case study). Phrack magazine **4**(58) (1996)
31. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Recent Advances in Intrusion Detection, Springer (2011) 121–141