Control Flow Obfuscation for Android Applications

Vivek Balachandran, Sufatrio, Darell J. J. Tan, Vrizlynn L. L. Thing
{balav, sufatrio, jjdtan, vriz}@i2r.a-star.edu.sg
Institute for Infocomm Research, Singapore

Abstract

Android apps are vulnerable to reverse engineering, which makes app tampering and repackaging relatively easy. While obfuscation is widely known to make reverse engineering harder, there is still a lack of Android obfuscation solution that realizes effective control-flow obfuscation, and make the resulting obfuscated apps sufficiently more complex from the app execution flow’s viewpoint. This paper presents our control-flow obfuscation techniques for Android apps at the Dalvik bytecode level. Our three proposed schemes go beyond simple control-flow transformations employed by existing Android obfuscators, and make it difficult for static analysis to determine the actual app control flows. To realize this, we also address a previously-unsolved register-type conflict problem that can be raised by the verifier module of the Android runtime system by means of a type separation technique. Our analysis and experimentation show that the schemes can offer effective obfuscation with reasonable performance and size overheads. Combined with the existing data and layout obfuscation techniques, our schemes can offer attractive measures to hinder reverse engineering and code analysis on Android apps, and help safeguard Android app developers’ heavy investment in their apps.

Keywords: Android, software obfuscation, mobile security, reverse engineering, application security.

1. Introduction

Recent years have seen a widespread and pervasive use of smartphones and tablet devices. The number of smartphones sold in 2015 surpassed 1.4 billion units, with Android dominating the market with an 80.7% share in Q4 [1].

This extensive user base of Android devices and the alarming level of security threats targeting them make the security of Android applications (henceforth called apps) become a high priority [2]. One of the threats to Android apps is the relatively easy recovery of their app code by an attacker [3]. Android apps can be reverse engineered easily using off-the-shelf tools like apktool [4] or IDAPro [5], which results in rampant app tampering and repackaging [6]. This significantly threatens the fast growing mobile app market, which is estimated to value at $5.5 billions in 2015 and will reach $8.9 billions in 2018 [7].

One of the techniques used against reverse engineering attacks is code obfuscation [8]. Obfuscation is the process of obscuring a piece of code so that it is harder to understand the reverse engineered code. Yet, the transformed code will not change the semantics of the original code. Due to this property, obfuscation has been recognized as a cost-effective mechanism to help app developers protect their released apps [9]. Existing Android obfuscation tools, however, seem to still lack complex control-flow obfuscation techniques (see Section 2).

This paper presents three schemes of performing control-flow obfuscation on Android apps at the Dalvik bytecode level, which go beyond simple control-flow transformations employed by existing Android obfuscators (see Section 3). The schemes disturb the normal control flow of an app method by making use of packed-switch construct, try-catch construct, and the combination of the two constructs. They make it difficult for static analysis to determine the actual control flows of the apps since the scattered code-block continuations and exception-raising operations are hard to be statically determined. While similar notion of control-flow flattening and signaling exist in the obfuscation literature [10] [11] [12], applying such techniques to Android bytecode still requires one to solve a possi-
ble register-type conflict issue, which can be raised by the Android runtime system when an app’s Dalvik bytecode is split, relocated and/or linked. To address this problem, we propose a register-type separation technique, which prevents the Android runtime system from encountering type conflict situations in the obfuscated apps (see Section 4).

Our analysis and experimentation show that the proposed schemes offer effective obfuscation with acceptable overheads (see Section 5). Experiments using popular apps and known benchmark suites show that our obfuscation incurs a reasonable size-overhead factor of 1.29, and a performance-overhead factor of 1.19. Combined with other existing forms of data and layout obfuscation techniques, our proposed control-flow obfuscation schemes can therefore make reverse engineering attacks on Android apps much harder. Furthermore, our register-type separation technique will also enable other forms of control-flow obfuscation techniques to be applied on Android apps. As such, our proposed schemes will help hinder reverse engineering of Android apps, and safeguard app developers’ heavy investment in their apps.

The remainder of this paper is organized as follows. Section 2 provides some background on Android obfuscation and runtime environments, and also compares related work. Section 3 presents our control-flow obfuscation schemes, while Section 4 describes the register-type conflict problem and elaborates our solution to it. Section 5 gives evaluation results of our obfuscation schemes. Section 6 further discusses our schemes’ usability and limitations. Finally, Section 7 concludes this paper.

2. Background and Related Work

2.1. Background on Android Obfuscation

Software obfuscation is a well-known technique used for protecting software from reverse engineering attacks [3]. Although it is theoretically impossible to achieve perfect obfuscation [11], obfuscating a program, in practice, makes its reverse engineering and code analysis harder [9]. The work [15] investigates the effect of obfuscation on reverse engineering of Android apps. It concludes that obfuscation, even with basic techniques employed by Proguard [10], can significantly increase the effort required to understand the obfuscated app. The importance of Android app obfuscation can be seen with the integration of Proguard into the Android build system. The Android developers’ documentation recommends releasing apps after obfuscating with ProGuard.

Android apps are released as Dalvik bytecode, which is then executed by the Android runtime system (more on this in Section 2.2). To analyze or manipulate an app in the absence of its source code, an attacker transforms the app bytecode into an assembly or higher level code representation. The motivation for the transformation may vary, including understanding the logic of a function, removing any watermark, or illegal app repackaging. Dalvik bytecode is an easy target for reverse engineering with various available reverse engineering tools, such as androguard [17], baksmali [18], apktool [11], IDAPro [5], dexdump [19] and dex2jar [20].

Obfuscation techniques that have been applied to Android apps by existing obfuscation systems include the following (see also Table 1):

1. Identifier renaming: It renames meaningful class and method names with arbitrary names like ‘a’ and ‘b’.
2. Class repackaging: This technique flattens existing multi-level class hierarchy by moving all classes into a single-level hierarchy.
3. Excessive overloading: It aggressively overloads different methods and fields using the same name.
4. Encryption of literal strings: This makes it difficult for an attacker to search for a particular string.
5. Encryption of classes, native libraries and assets: This technique encrypts an entire class, library and asset; which are then decrypted at runtime for their execution.
6. Java reflection: It can statically hide the actual names of the accessed fields and invoked methods of a class.
7. Splitting .dex file: Splitting the bytecode and then performing a dynamic/runtime code loading can also hide, and thus obfuscate, app code.
8. Control-flow obfuscation using opaque predicates: Opaque predicates together with junk-code insertion (explained below) are employed by some obfuscators.
9. Removing debugging information: This will make code understanding harder.
One important type of obfuscation is control-flow obfuscation, which aims to hinder an attacker in understanding the logical flow of a program. Some control-flow obfuscators perform simple techniques, like inserting opaque predicates. An opaque predicate is a conditional instruction that always returns the same truth value, yet it is hard to be statically determined [8, 21]. Inserting an opaque predicate gives us two branches: one for the original code, and another for unexecuted junk instructions. Another basic control-flow obfuscation technique is the insertion of linked jump instructions. In this technique, a goto statement is inserted at the beginning of a method to jump to the end of the method. At the end of the method, another goto statement is inserted to return to the first original instruction of the method. The control-flow graph (CFG) is thus changed without affecting the semantics of the program.

2.2. Background on Dalvik Bytecode and Android Runtime Environments

An Android app is written mostly in Java, but allows the incorporation of native code through Java Native Interface (JNI). The Java source code of an app is compiled into Dalvik bytecode (.dex) [22], which is a register-based bytecode format specifically designed for Android. All files belonging to an app are ultimately packaged and signed as a single APK file. [23] shows the steps involved in the Android build process.

On the user’s device, the Dalvik bytecode of an app is run by a process virtual machine within the Android runtime environment. Two Android virtual machines are available, namely Dalvik Virtual Machine (DVM) and Android runtime (ART) [13]. The former had been used since the earliest Android version until it was replaced by the latter in Android 5.0 (Lollipop). ART improves its predecessor DVM with performance improvement techniques such as ahead-of-time (AOT) compilation and improved garbage collection, as well as better support for development and debugging. ART and DVM, nonetheless, are compatible runtime systems running Dex bytecode as stated in the Android’s official developer site [13], apart from a few specific techniques listed in [24]. That is, apps developed for DVM should generally work when running with ART since both virtual machines execute the same unmodified Dalvik bytecode file structure, instruction formats [22], and constraints [25]. Hence, our proposed obfuscation, including its register-type separation technique, applies to apps run by both DVM and ART. In the rest of the paper, we use the term Android runtime system when we want to refer to both DVM and ART. We will, however, highlight the differences between DVM and ART when necessary, such as when discussing the reported runtime error messages in Section 4.1.1.

2.3. Related Work

There are various Android obfuscation tools available in the market, such as Proguard [16], DexGuard [27], DashO [28], DexProtector [29], jarg [30], JODE [31], Allatori [32] and yGuard [33]. Table 1 shows the types of obfuscations performed by the various Android obfuscation tools according to the techniques listed in Section 2.1.

<table>
<thead>
<tr>
<th>Tools</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proguard [16]</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DexGuard [27]</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DashO [28]</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DexProtector [29]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td>jarg [30]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>JODE [31]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td>Allatori [32]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>yGuard [33]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✅</td>
</tr>
</tbody>
</table>

The current Android obfuscation tools seem to still lack complex control-flow obfuscation techniques. Only a few tools perform control-flow obfuscation, and they realize only simple techniques like inserting opaque predicates. Similarly, in the academic literature, we also have yet to see any proposed complex control-flow obfuscation of Android apps. The work [34, 35] applied only basic control-flow obfuscation techniques to Android malware in order to show how existing anti-malware systems lagged behind against malware transformation. Meanwhile, [36] proposes data and object-oriented design obfuscation, which can complement our control-flow obfuscation. Our control-flow obfuscation techniques, in contrast, are complex ones.

---

[2] Due to its OAT compilation, ART internally compiles an input Dalvik bytecode to native code in OAT files, which replace DVM’s odex (Optimized Dalvik EXecutable) files [26].
which scatter the code fragments in the program and increase execution flow diversion between the fragments. To the best of our knowledge, our work is the first that does so, including addressing the type-conflict problem raised by the Android runtime system as explained in Section 4. We believe that complex Android control-flow obfuscation like ours should be incorporated into existing Android obfuscation tools.

Most parts of Android apps are written in Java. Obfuscation techniques in Java are therefore relevant in the context of Android obfuscations. Various obfuscation techniques for Java programs have been proposed in the last decade. A taxonomy of obfuscations with numerous techniques on Java source code is given in [5]. A program control-flow obfuscation using opaque-predicate based conditional instructions is also discussed in [37]. One important drawback of performing control-flow obfuscation at the source code level is that code splitting can be done only at the boundaries of the basic blocks [10] due to variable scoping or data dependency issues. For Java source-code obfuscation, there exists another important challenge. In Java, higher-level loop constructs, such as while or for, cannot be easily transformed into if-then-goto constructs for subsequent code splitting [10]. Unlike other languages like C, Java has no unconditional goto jump. Instead, it allows only a limited form of jump to the end of a blocked statement. Our work obfuscates Android apps at the Dalvik bytecode level, where the splitting of the code is not limited by the boundaries of the basic blocks.

Obfuscations for Java bytecode exist in the literature. The most common technique is the renaming of classes, methods and identifiers as explained in [10]. The work [11] proposes a Java bytecode obfuscation technique where identifiers are reused excessively by means of identifier overloading. Meanwhile, [39] performs Java bytecode obfuscations by first translating Java bytecode into an intermediate representation (IR) called Jimple. Working on a low-level IR allows JBCO to insert goto instructions and implement its goto instruction augmentation technique. The technique randomly splits a method into two sequential parts, then spatially reorders the two parts, and inserts two goto instructions to direct the correct execution flow. While these obfuscators work on Java bytecode, which are executed by the stack-based JVM, our work performs control-flow obfuscation on Dalvik bytecode of Android apps, which are executed by the register-based DVM or ART. As such, we also need to address specific issues that are raised in the Android runtime environment, such as the register-type conflict problem elaborated in Section 4.

Wang et al. [10] propose an obfuscation technique using switch statement to flatten the control flow. While the obfuscator is similar to our packed-switch obfuscation described in Section 3.2, it is however done on C source code. The work [12] implements the same control-flow flattening on C++ source code. Since both [10] and [12] obfuscate a program’s source code, they can only split the code at the boundaries of the basic blocks. In contrast, our packed-switch obfuscation can arbitrarily select a cutting point, including within a basic block, when splitting a Dalvik bytecode sequence into two different code blocks. As a result, our scheme can form a higher number of code fragments (n). This distinction is important since the complexity of a switch-based obfuscation is $O(n!)$, which becomes intractable only if $n$ is sufficiently high. Moreover, their technique is not directly applicable to Dalvik bytecode without addressing the Dalvik register-type conflict problem, which we solve in our work.

Chow et al. [11] discuss a control-flow flattening technique by splitting basic blocks and employing a dispatcher module, which is modeled as a deterministic finite automaton, to reconstruct the correct control flow of the flattened code. To achieve better obfuscation, insertion of dummy blocks is also performed. In their analysis of the scheme, Chow et al. derive that the complexity of determining the execution flow of the dispatcher is in FSPACE-complete. The scheme, however, assumes a generic imperative-language based code with no runnability requirement, which exists for obfuscation in Android environment (see Section 3.4). Their scheme thus cannot be directly applied to Dalvik bytecode.

Sandmark tool [38] is capable of performing a class-splitting operation, which splits a class into multiple classes. Sandmark, however, does not provision a technique to arbitrarily split a basic block into smaller code fragments and shuffle them within the program. Rather, it can perform basic-block splitting only by inserting opaque predicates within the basic blocks, thus structurally breaking the blocks. Sandmark also employs primitive promoter technique, which wraps all primitives, as well as signature bludgeon technique, which converts all methods to take Object[] and return Object[]. These source-code level techniques, however, cannot guarantee that the type-conflict prob-
Table 2: Comparison of existing systems that perform control-flow obfuscation and our proposed Android obfuscator.

<table>
<thead>
<tr>
<th>Obfuscation System</th>
<th>Obfuscation Target</th>
<th>Allow Splitting of a Basic Block?</th>
<th>Work on Android APKs?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wang et al. [10]</td>
<td>C source code</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Collberg et al. [8]</td>
<td>Java source code</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Sandmark et al. [38]</td>
<td>Java source code</td>
<td>Limited</td>
<td>No</td>
</tr>
<tr>
<td>Chow et al. [11]</td>
<td>Generic imperative-language code</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Java ByteCode Obfuscator [39]</td>
<td>Java bytecode</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Our obfuscation schemes</td>
<td>Dalvik bytecode</td>
<td>Yes</td>
<td>(address Dalvik type-conflict problem explained in Sect. 3)</td>
</tr>
</tbody>
</table>

lem at the Dalvik bytecode level (see Section 3) will never occur. This is because, at the Dalvik bytecode level, fields of class instances are accessed using `iget` and `iput` instruction families, which make use of a register as the destination and source operand, respectively [22]. Since registers are involved, the same register-reuse technique can be applied by the compiler and, accordingly, the type-conflict problem may occur. In our obfuscation, the type-separation and obfuscation steps are performed on Dalvik bytecode after its conversion from the source code by the compiler. The type-conflict problem can thus be avoided.

The differences between the above-mentioned control-flow obfuscation systems and ours are summarized in Table 2. Notice that if a control-flow obfuscation system targets Android apps, then it needs to address the Dalvik type-conflict problem as explained in Section 3.

Lastly, we also mention another obfuscation technique that utilizes bytecode, namely virtualization obfuscation. In this technique, a target program is transformed into bytecode for a custom, and possibly randomized, virtual machine (VM) interpreter. The corresponding VM interpreter is attached and distributed along with the bytecode. Analysis of the executed bytecode would face a great challenge since only the structure and logic of the bytecode interpreter are directly visible [43]. Several works have been done to reconstruct the original logic of virtualized-obfuscated programs. Rolles [44] suggests reverse engineering the VM, detecting entry points into the VM, and then developing a disassembler that converts the bytecode into intermediate code and subsequently machine code. Coogan et al. [43] propose a method that combines bit-level taint analysis and semantic-preserving transformations. Kinder et al. [45] makes static analysis of the program more effective by sensitizing the value of the virtual program counter managed by the inserted interpreter. In our work, we do not transform a target Dalvik bytecode to work with a custom VM interpreter. In fact, the in-place Android runtime system enforces various constraints on Dalvik bytecode [25], which may cause it to terminate its execution of obfuscated Android apps. We address this constraint requirement with register-type separation technique.

3. Control Flow Obfuscations

Our proposed schemes obfuscate Android apps at the Dalvik bytecode level to achieve an expressive control-flow obfuscation. As also noted in [8], while Java language can express limited structured control flow, lower level virtual machine code like Dalvik bytecode can express arbitrary control flow. However, there exist several requirements that need to be met by control-flow obfuscation in Android environment, which are discussed next.

3.1. Control-Flow Obfuscation Objective and Correctness Requirements

Control-flow obfuscation aims to transform an app control flow so that the transformed control flow statically has an intractable number of possible basic-block ordering combinations, thus hindering the comprehension of the app. Generally, there are two ways in which we can disturb the control flow of an app method to make its CFG much more complex to analyze. One is by disturbing the linear ordering in which code blocks are written in a method. Another is by increasing the control-flow branches or diversions within the method.

An Android bytecode $B'$ that is generated from applying a control-flow obfuscation preserves the execution semantics of its corresponding original
bytecode $B$ if it satisfies the following obfuscation-correctness requirements:

$R_1$: Execution-flow preservation: $B'$ executes all operations of $B$ in their original order.

$R_2$: No computation interference with the added instructions: Any added instructions in $B'$ do not interfere with the computation, i.e., data processing, of $B$’s existing operations.

$R_3$: Runnability of the obfuscated code: $B'$ must be runnable by the Android runtime system.

The requirement $R_3$ is unique to Android due to the fact that Android bytecode is a form of managed code. Such a requirement does not apply to executables that run directly on top of an OS. Existing x86-based obfuscation work [46, 12], for instance, can apply advanced techniques like self-modifying code and even those that interfere with the way the OS works, such as execution-stack manipulation. In contrast, the Android runtime system imposes various constraints on a valid Android bytecode by performing static analysis based checks [25] prior to executing the bytecode.

We propose three obfuscation schemes to realize the above-mentioned obfuscation objective. Our obfuscation process, as part of the Android app build workflow by app developers, is depicted in Figure 1. The remainder of this section describes how our control-flow obfuscation schemes basically work, which fulfill the requirements $R_1$ and $R_2$. Section 3.2 explains how the schemes can satisfy the runnability requirement $R_3$ in the presence of a verifier module of the Android runtime system.

3.2. Packed-Switch Obfuscation

In this obfuscation technique, we split the code in a target method into $n$ equal code fragments and shuffle the code fragment locations within the method, thus disrupting the linear order of the method. The obfuscator inserts a packed-switch construct into the program that will help in executing the shuffled code fragments in the correct order during app runtime. Opaque variables [8, 21] are used to obscure the switching index so that static analysis becomes ineffective.

In Dalvik bytecode, packed-switch is used to represent a switch-case construct. It is similar to the tableswitch in Java bytecode. The following steps explain the obfuscation using packed-switch.

3.2.1. Splitting Method into Code Fragments

The first step of the obfuscation is splitting the target method equally into $n$ code fragments. As mentioned earlier, we are allowed to split a method in the middle of a basic block, similar to that in [11]. At the beginning of each code fragment, a label is inserted. For instance, the $i$-th code fragment starts with a label :obfpswitch$i$. These added labels help in reaching the formed code fragments.

3.2.2. Inserting Packed-Switch and Looping over the Code Fragments

For the correct execution of the method after splitting into code fragments, the code fragments should be executed one after another in the exact same order as it appears in the original code. We can increase the control-flow complexity and attain the same execution flow with the help of packed-switch within a loop construct as shown in Figure 2.

We use a new register as a switching register to switch between the code fragments. At the beginning of each code fragment, a label is inserted. For instance, the $i$-th code fragment starts with a label :obfpswitch$i$. These added labels help in reaching the formed code fragments.

For our prototype implementation, the obfuscator applies code transformation on the smali representation of an app’s Dalvik bytecode, which can be generated by apktool [4]. The smali representation of a method lists the text-based mnemonics that directly correspond to the bytecode of the method. Other alternative implementations are also possible, such as using dexlib2 library [47].

3.2.1. Splitting Method into Code Fragments

The payload of the packed-switch instruction actually takes a literal integer value that indicates the first (and lowest) switch case value [22]. In our example, we set it to 0, forming the line .packed-switch 0x0. As such, the value of our switching register directly corresponds to the index of the packed-switch’s target-address table.
the register used for switching is v10, which is initialized with 0. In this example, we used a direct assignment of the value to the register. However, an opaque variable generated using a hash function can be used to set the value of the switching register, thus making it difficult for static analysis.

A new code block containing :loop label and a packed-switch instruction with the switching register v10 is inserted at the beginning of the method. This packed-switch instruction redirects the execution control to the correct code fragment based on the value v10 holds.

We also modify each formed code fragment by adding two new instructions at the end of the fragment to update the switching register v10. This register is assigned the value of the next code fragment that has to be executed. Similar to the register’s initial value, the values of all next-code fragments are just shown in clear in this example. The last instruction added is a jump instruction to the packed-switch instruction.

To the end of the method, the packed-switch payload block is inserted. It consists of the identifier to the packed-switch, the lowest switch case value, and all the target addresses of the code fragments.

3.2.3. Shuffling the Code Fragment Locations

The next step is to shuffle the code fragment locations in the method so that the linear order of the code fragments in the method is disrupted. The code fragments can be shuffled in any order as the packed-switch construct works based on the target addresses. The code fragments are randomly shuffled so that an attacker cannot infer the order in which the code fragments are executed by looking at the spatial alignment of the code fragments.

3.2.4. Inserting Junk Code Fragments

Inserting new junk code fragments into the method further confuses the reverse engineer, which is also done in prior work [11]. The switching register will not be assigned the value needed for this code fragment to get executed. The injected junk code fragment thus never gets executed.

In the example shown in Figure 2, obf_pswitch is the label to a junk code fragment inserted into the method, which is marked by #CODE_FRAGMENT_JUNK. Register v10 should possess the value 1 for the junk code to get executed. However, none of the valid code fragments will ever assign value 1 to v10.
3.2.5. Inserting Junk Packed-Switch Target Addresses

The payload of the packed-switch instruction contains the target addresses of the code fragments to which the control flow is transferred based on the switching register’s value. At obfuscation time, the obfuscator knows the values that will be taken by the switching register. The target addresses that are mapped to unassigned values will not get executed as illustrated in Figure 3. They simply point to random points in the code. In the CFG of the reverse-engineered method, there will be edges to these junk target addresses from the switching condition. Yet, these branches will never be taken during the program execution.

3.3. Try-Catch Obfuscation

This second obfuscation scheme diverts the app execution flow between a try block and its corresponding catch block using exception-raising instructions. The exception-raising instructions are, however, difficult to be statically determined due to their use of opaque variables. The control-flow diversion due to an obscured exception allows us to hide a sequence of instructions from the code segment in the try block without affecting the semantics of the code. The relocated sequence of instructions, instead, appears unconnected within the catch block.

While theoretically we can hide any Dalvik instruction, our scheme is concerned with hiding control-flow instructions like jumps (e.g. goto) and method invocation calls (e.g. invoke-direct). Hiding these instructions modifies the control-flow logic of the entire target method. The scheme’s basic strategy is thus similar to the signal-based obfuscation for the x86 platform [12].

In what follow, we describe how we can realize a try-catch based obfuscation on Android apps.

3.3.1. Formatting the Try Block

Once the instructions in a method to be relocated are selected, a try block is defined to encompass all remaining instructions in the method. To have a potent obfuscation, we want to have multiple exception-raising instructions within a single try block. To facilitate this, before each exception-raising instruction, we need to initialize an integer flag to uniquely identify each exception-induced execution diversion to the catch block. As a standard practice, opaque variable(s) should be used to set this flag. Following each exception-raising operation, junk code can also be injected to further obscure the execution flow. The junk code may contain potential exception-raising instructions. Additionally, a number of junk flag-setting operations may also be injected throughout the try block.

Figure 4 illustrates a sample try block that camouflages some instructions. For easy illustration, we use a simple division-by-zero exception and with no opaque variables used for the flag setting and exception-raising operations.

3.3.2. Adding the Catch Block

The new catch block that reconstructs the original control flow of the program is inserted just after the try block. We purposely use a single catch block to handle the all-encompassing Java.lang.Exception class. The catch block can therefore be a common jump target of all potential exception-raising instructions in the try block.

Within this catch block, we can use a packed-switch construct to dispatch a diverted execution path. In the case when the relocated operation is an invoke-* or conditional jump, we need to insert a goto to jump to the continuing operation. Note that we can add spurious switch cases that will never be executed. Figure 5 shows the corresponding catch block for the try block in Figure 4.

```plaintext
:try_start # Start of the try block
const/4 v4, 0x0 # Set the flag register v4
div-int/lit8 v5, v5, 0x0 # Raise an exception
(added junk code after the exception)
:label_1
(continuing operations after a diversion to the catch block)

:label_2 # An added dummy label

:try_end # End of try block
.return-void
```

Figure 4: A try block example. The two div-int/lit8 instructions, which are shown in red, raise an exception by dividing their second operand by zero. Notice again that we do not show the use of opaque variables on the flag register v4 and div-int/lit8 operands.
3.4. Combining Packed-Switch and Try-Catch Obfuscations

This sub-section outlines our scheme that combines the packed-switch and try-catch obfuscation techniques as follows:

1. We first apply the packed-switch obfuscation (see Section 3.2) to split a method into code fragments, and then reshuffle the formed code fragments.

2. Next, we apply the try-catch obfuscation to the resulting code of Step 1 as follows. The entire packed-switch based obfuscated code is first put inside the try block. The packed-switch instruction is then moved from the try block to the catch block. As its replacement in the try block, we use an exception-raising instruction.

3. Lastly, other control-flow instructions within the method can additionally be subjected to the try-catch based instruction hiding as explained in Section 3.3.

Notice that the packed-switch instruction is also a form of control-flow instruction. It dispatches the execution flow according to its accompanying jump table and the value of the switch register. Figure 6 shows the combined obfuscation scheme with only the packed-switch camouflaged. It can be noticed that control-flow edges emanate from the catch block into different code fragments.

3.5. Satisfying the Control and Dataflow Preservation Requirements

As can be seen, our proposed obfuscation schemes preserve the execution order of the original instructions within each method of a target app. Hence, they satisfy the requirement $R_1$. Furthermore, our schemes add new instructions to direct the correct original instructions by using a disjoint set of registers so as to meet the requirement $R_2$.

4. Type-Conflict Problem and Solution

This section elaborates an important problem caused by the Android runtime system, which we encountered while implementing the described obfuscation techniques. It also explains how we modified target apps prior to applying the obfuscation...
techniques so as to prevent the problem and satisfy the runnability requirement $R_3$.

4.1. Type Conflict Problem Issued by Android Runtime System

When performing complex control-flow obfuscation on Android apps, such as the schemes described in Section 3, one also needs to address a technical obstacle posed by the verifier component of the Android runtime system [48, 49, 50]. In DVM, this verifier component is often commonly referred to by its internal name VFY. It ensures that an executed app's Dalvik bytecode complies with various static and structural constraints [25].

Dalvik structural constraints are enforced upon relationships between different elements of the bytecode [25]. Prior to executing a method, the verifier seems to perform control and data flow analyses [49, 50] on the method to verify if it observes the constraints. This verification includes ensuring the absence of a register with potentially conflicting data types within a method. If a type conflict is detected, the Android runtime system will terminate the app execution, i.e the running app will crash.

A complex control-flow obfuscation on Dalvik bytecode may split method instructions into different code segments, relocate and link them. Such actions will generally cause the verifier to report a register-type conflict. A detailed documentation of how the verifier exactly works is, however, unavailable. Hence, we need to resort to experimentation to determine what an Android obfuscation must observe in order to avoid triggering a type-conflict verification error, which is explained next.

4.1.1. A Motivating Example

Let us consider a small Java method, which is shown in Figure 7 as our motivating example. This method performs a simple arithmetic calculation to print the absolute value of an integer variable $a$. In addition, it prints whether $a$ is a negative or non-negative integer. To derive a compact smali method, the method simply invokes two auxiliary methods: printString() and printInt(). They print out the value of the passed string and integer argument, respectively.

The produced smali code of our sample Java method is shown in Figure 8. Notice how the smali code reuses a register for different data types. One example is register v2, which is assigned with integer 0x5, as well as strings "Negative" and "Non-negative".

Figure 7: A Java method as our motivating example.

Figure 8: The smali code representation of our sample Java method. This code is converted from the Dalvik bytecode that is generated by the Android SDK.

Figure 9 shows our sample method that has been subject to a packed-switch obfuscation as explained in Section 3.2. For our clearer explanation, the obfuscated smali code are annotated with comments, use descriptive labels, and employs assignments using plain constants instead of opaque ones.

One would expect that the obfuscated code should run without raising any register-type conflict. The applied obfuscation does split the app statements into three code blocks. Nevertheless, the actual program execution flow on the original statements across the code blocks remains the same. Yet, when we run the obfuscated app, the DVM terminated the app, and outputed the following type-conflict error messages to logcat:

VFY: register1 v2 type 2, wanted ref
VFY: bad arg 0 (into Ljava/lang/String;)
VFY: rejecting call to
Lcom/example/helloworld/
   MainActivity;.printString
   (Ljava/lang/String;)V
VFY: rejecting opcode 0x71 at 0x003f

ART, meanwhile, reports the following error messages:
E/AndroidRuntime( 1502):
   java.lang.VerifyError: Rejecting class com.example.sampleApp.MainActivity because it failed compile-time verification ...

From these error messages, especially the ones reported by DVM, we can observe that the verifier performs a static analysis to ensure the absence of type-conflicting registers in a conservative manner. In this particular example, it cannot correctly infer that the value of v2 right after the label :obf_pswitch_2, at the beginning of block 2, will be a string reference following its last assignment towards the end of block 1. Instead, the verifier concludes a possibility of v2 holding an integer type due to an integer definition point in block 0.

This type-conflict problem due to control-flow obfuscation is also noted in [51, “Merging or Splitting Code”, pp. 41–43]. However, no effective general solution to the problem is proposed in the work.

4.1.2. Problem Definition

We now give a succinct problem definition based on our motivating example above. Dalvik bytecode allows for register reuse in that a register may hold different, often incompatible, data types within a single method. In unobfuscated apps, such practice never raises any type conflict problem. This is because a register is reused in such a way that it will not reach any program point where two incompatible data types are in scope at the same time. The compiler generating the Dalvik bytecode ensures this property by solving the register allocation problem known in the compiler theory. It applies a variable liveness analysis so that a register is reused for multiple variables only if the variables are never simultaneously live.

Applying control-flow obfuscation techniques, however, can violate the type-safety checks by the Android runtime verifier. Added goto operations, which are never actually executed due to the use of opaque predicates, are taken into account by the

```
.method public static sampleMethod()V
   .locals 7
   # Register initializations with neutral 0
   const v0, 0
   const v1, 0
   const v2, 0
   const v3, 0
   const v4, 0 # The added block counter
   const v5, 0 # An added temporary register
   const v6, 0 # The added number of blocks (3)
   ### Beginning of the for loop ###
   const v4, 0x0
   const v6, 0x3
   :obf_goto_0
   if-ge v4, v6, :obf_cond_0 # Exit if v4 = 3
   packed-switch v4, :obf_pswitch_data_0
   :obf_goto_1
   add-int/lit8 v4, v4, 0x1 # Increment v4
   goto :obf_goto_0
   ### Block 0 ###
   :obf_pswitch_0
   const/4 v2, 0x5
   move v0, v2
   move v2, v0
   if-gez v2, :cond_0
   const v4, 0 # Set block counter to 0
   goto :obf_goto_1
   ### Block 1 ###
   :obf_pswitch_1
   const-string v2, "Negative"
   move-object v1, v2
   move v2, v0
   const/4 v3, -0x1
   mul-int/lit8 v2, v2, -0x1
   move v0, v2
   :goto_0
   move-object v2, v1
   const v4, 1 # Set block counter to 1
   goto :obf_goto_1
   ### Block 2 ###
   :obf_pswitch_2
   invoke-static v2, Lcom/example/helloworld/MainActivity;->printString(Ljava/lang/String;)V
   move v2, v0
   invoke-static v2, Lcom/example/helloworld/MainActivity;->printInt(I)V
   return-void
   :cond_0
   const-string v2, "Non-negative"
   move-object v1, v2
   const v4, 2 # Set block counter to 2
   goto :obf_goto_1
   :obf_cond_0
   goto :goto_0
   :obf_pswitch_data_0
   .packed-switch 0x0
   :obf_pswitch_0
   :obf_pswitch_1
   :obf_pswitch_2
   .end packed-switch
   .end method
```

Figure 9: An obfuscated smali code of our sample method. The shaded statements are the original ones present in the unobfuscated version.
verifier in its analysis. The proposed packed-switch obfuscation may raise type-conflict problems due to the inability of the verifier to statically resolve the correct code-block continuation. The verifier may find a register’s excepted type at the beginning of a particular code block to be in conflict with another in-scope type at the end of another code block that will never actually be executed consecutively. Similarly, in the try-catch based obfuscation, with a catch block as shown in Figure 6, the verifier insists that all added \texttt{goto} operations from the catch block to the try block will not cause any type conflict on all reused registers.

More formally, we can view this register-type conflict problem from the static program analysis' viewpoint as follows. The verifier module seems to perform a register liveness analysis in each method prior to running it, and ensure that there is no type conflict among live registers at any program point. To do this, at each program point, it keeps track of live registers together with the data type they hold. Should it find a register with two different types at the same program point, it will then report the register-type conflict problem as shown above.

Given this issue, we thus need to find a general solution that will prevent the verifier from deducing any false type conflict in otherwise correctly obfuscated apps.

### 4.2. Proposed Register-Type Separation Technique

To address the problem defined above, we propose the register-type separation technique to be applied before performing the control-flow obfuscations. The technique first checks if a register may hold different data types within a method. If that is the case, it then allocates a new register for all occurrences of the reused register holding a particular data type. One numeric register is sufficient to hold all compatible numeric-based data types, such as \texttt{integer}, \texttt{float} and even \texttt{Boolean}. This is because the Dalvik-level instructions treats numeric-based register operands uniformly.\footnote{Dalvik bytecode makes use of a separate instruction set to operate on Java’s wide numeric types \texttt{long} and \texttt{double}, each of which requires two consecutive registers.} We, however, need to allocate one register for each encountered object-type. Given this register-type separation, the Android runtime verifier will no longer deduce any program point where two incompatible data types are in scope at the same time.

Referring to our earlier program analysis’ definition of the type-conflict problem, our type-separation technique thus first applies a type determination analysis on registers in each method. Subsequently, it performs a register renaming by assigning one register for each of the held data types. As a result, the register liveness analysis performed by the verifier will never encounter a register with different types at any program point in each method.

To realize this register-type separation technique, we need to address the following two important sub-problems. First, the number of registers within each method will increase. One problem with Dalvik operations is that some operations, such as \texttt{move vA, vB} and \texttt{array-length vA, vB}, can only accept the first 16 registers, i.e. \texttt{v0–v15}. Some of these operations have a corresponding generalized operation that can take high registers. We can thus simply replace them with their respective generalized operation. For example, \texttt{move} can simply be replaced with \texttt{move/from16}.

Some of these instructions, however, have no generalized equivalences. On these instructions, we need to first assign any used high register to a free low register using either \texttt{move/from16} for single-width numeric, \texttt{move-wide/from16} for double-width numeric, or \texttt{move-object/from16} for object register. All utilized low registers are then passed as operands to the instructions.

Some instructions that take only low registers also set one of their operand registers. For instance, the instruction \texttt{array-length vA, vB} sets \texttt{vA} with the length of the array referenced by \texttt{vB}. Before these instructions are executed, their high registers are copied to low registers as above. After the execution of the instructions, the low register that is set by the instructions is copied back to the corresponding high register. This produces the same effect as a direct value setting by such instructions.

In our implementation, we set aside five low registers to function as these auxiliary registers. This is because Dalvik instructions take at most five low registers in the case of \texttt{invoke-\*} and \texttt{filled-new-array} operations.\footnote{To prevent possible type conflict on these five registers, during the obfuscation process, we always put together any added assignment operations with the original instruction operation. This ensures that the auxiliary registers are always defined first and subsequently used with the same data type. Besides these five registers, we also need to reserve several other low registers for implementing control-flow obfuscation.} To prevent possible type conflict on these five registers, during the obfuscation process, we always put together any added assignment operations with the original instruction operation. This ensures that the auxiliary registers are always defined first and subsequently used with the same data type. Besides these five registers, we also need to reserve several other low registers for implementing control-flow obfuscation.
i.e. the loop counter, flag and exception-raising registers, as well as for generating opaque predicates and variables. Hence, we decided to shift all existing registers by 16. That is, all operations in the obfuscated code take registers numbered 16 to 255. In this way, the instructions can be uniformly treated during their rewriting process.

The second sub-problem is that we need to determine the type of each operand so that the register separation process can decide which new register to replace an operand. Based on analyzing the Dalvik operations, in most cases, we can safely determine the data type based on the operation’s operand requirements [22]. For instance, the operation `array-length vA, vB` always takes a numeric first operand `vA` and an object second operand `vB`. A problem arises, however, since there are several operations whose operand types cannot be determined simply by analyzing the operand requirement. These operations are `const/16`, `const`, `const/high16`, `if-eq`, `if-ne`, `if-eqz` and `if-nez`. The first three instructions lead to ambiguous operand type when they take `0x0` as the assigned value because Dalvik does not distinguish `0` and `null`. Likewise, the four `if-` operations can compare both numeric and reference types due to the dual type-possibility of a zero-valued register.

To solve this type-ambiguity problem, a type inference solution is known in the literature [52]. It basically determines a register’s type information at an ambiguous operation by looking at how it is used in another connected operation where the operand type is known. That is, for each ambiguous register declaration, the algorithm performs a depth-first search on its method’s CFG to find any register use points that can expose the register type. Conversely, known type information is forward propagated from each register declaration to all ambiguous register use points.

We remark that our type separation technique may allow one to more easily inspect the data type held by a register. However, the same information can be obtained using a simple data-flow analysis on the inspectable Dalvik bytecode.

4.3. Satisfying the Runnability Requirement

Our register-type separation technique preserves the semantics of a target method, and satisfies the runnability requirement $R_3$ by observing all Dalvik bytecode constraints [25], when performing its following code transformations:

1. Register-index shifting.
2. Separation of a register used for multiple data types, which assigns one unused register for each held data type.
3. Replacement of instructions with their variants that can operate on higher registers.
4. Operation expansion on instructions that take only low registers. This step adds assignment operation(s) to unused low registers prior to the instructions, and also assigns any set low register to its corresponding high register following the instructions. To avoid possible type-conflict problem on the used low registers, our technique puts the preceding and succeeding added assignment(s) together with the original instruction as an atomic unit. That is, our subsequent obfuscation process will not split them. Because of this, all auxiliary low registers are always defined and used with the same type within each contiguous instruction sequence. As a result, there is no type-conflict problem reported on them.

5. Evaluation and Analysis

In this section, we discuss the overhead and efficacy of our obfuscation techniques. We performed our evaluation on an Android Virtual Device, which is part of the Android SDK, emulating Google Nexus 4 phone with 2GiB of RAM. It runs Android 4.2.2 with DVM as the Android virtual machine.

5.1. Size Overhead

One of the problems of applying our obfuscation techniques is that new instructions are added into the code. As a result, the size of a target app increases. We thus aim to observe the impact of obfuscation on the size of an Android APK file. To this end, we applied the combined obfuscation to several popular apps downloaded from the Google Play store. For each app, we split each method selected for obfuscation into three equal code blocks. We then measured the size of the original app and the obfuscated one.

Table 3 shows the increase in the APK file size due to this obfuscation. The average size-increase factor of the obfuscated apps is 1.29. We also found that, in all tested apps, the average size increase of each obfuscated method is about the same at 1.65 KiB.
Table 3: Size overhead on several popular test apps. App sizes are expressed in MiB, which is the IEC standard acronym for mebibyte. 1 MiB = $2^{20}$ bytes.

<table>
<thead>
<tr>
<th>App</th>
<th>Number of obfuscated methods (N)</th>
<th>App size before obfuscation (Sb) in MiB</th>
<th>App size after obfuscation (Sa) in MiB</th>
<th>App increase factor (Sa/Sb)</th>
<th>Average app size increase per method (Sa – Sb) / N in MiB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Winamp</td>
<td>3,330</td>
<td>21.8</td>
<td>30.3</td>
<td>1.22</td>
<td>0.001651</td>
</tr>
<tr>
<td>Wechat</td>
<td>14,501</td>
<td>77.1</td>
<td>101.4</td>
<td>1.31</td>
<td>0.001675</td>
</tr>
<tr>
<td>Line</td>
<td>14,468</td>
<td>68.0</td>
<td>91.7</td>
<td>1.34</td>
<td>0.001638</td>
</tr>
<tr>
<td>Sound Cloud</td>
<td>6,740</td>
<td>43.2</td>
<td>54.1</td>
<td>1.25</td>
<td>0.001617</td>
</tr>
<tr>
<td>Photo Editor</td>
<td>2,397</td>
<td>16.8</td>
<td>20.7</td>
<td>1.23</td>
<td>0.001627</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>1.29</strong></td>
<td><strong>0.001647</strong></td>
</tr>
</tbody>
</table>

Table 4: Performance overhead on a number of test apps.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Elapsed time of unobfuscated method (in ms)</th>
<th>Elapsed time of obfuscated method (in ms)</th>
<th>Time increase factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOlden</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BiSort</td>
<td>1,878</td>
<td>2,324</td>
<td>1.24</td>
</tr>
<tr>
<td>TreeAdd</td>
<td>1,494</td>
<td>1,795</td>
<td>1.20</td>
</tr>
<tr>
<td>Perimeter</td>
<td>2,240</td>
<td>2,546</td>
<td>1.137</td>
</tr>
<tr>
<td>Ashes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fahrenheit</td>
<td>94</td>
<td>135</td>
<td>1.44</td>
</tr>
<tr>
<td>Factorial</td>
<td>16</td>
<td>21</td>
<td>1.31</td>
</tr>
<tr>
<td>HashTabletest1</td>
<td>28</td>
<td>34</td>
<td>1.21</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>18</td>
<td>22</td>
<td>1.22</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>1.19</strong></td>
</tr>
</tbody>
</table>

5.2. Performance Overhead

We now evaluate the performance overhead due to our combined obfuscation technique. For this, we obfuscated several modules of two freely-available Java benchmark suites: Ashes Suite Collection [53] and JOlden [54]. Since the modules are written as stand-alone Java classes, we needed to build a small Android app and embed them. We utilize these pure Java-based benchmarks so as to provide a more accurate performance measurement free from any effects of UI interaction with the user. The two benchmark suites meet our Java benchmark requirements with the following properties:

- Freely-available benchmarks with self-contained Java source code that are runnable within Android apps: for allowing a porting into Android app;
- Non-interactive and windowless: for factoring out any user interaction effects;
- Deterministic: for replicability of the observation.

We executed the ported benchmark modules, and measured their execution time. All the benchmark modules were then obfuscated, and their execution time were also measured. The benchmark modules of Ashes test suite were executed 100 times to calculate the elapsed time.

Table 4 shows the performance overhead caused by our obfuscation technique. We can see that, on average, the performance overhead increases by a factor of 1.19. This shows the performance penalty incurred by our combined obfuscation scheme is similar to those achieved by other widely known control-flow based obfuscation techniques in the x86 environment such as [12, 55, 46].

Another measure of time complexity is variance in execution time of the obfuscated program as the input size varies. For this, we built a small app that performs a bubble sort operation on an integer array to sort the numbers in ascending order. The input array is purposely arranged in descending order, thus simulating a worst case scenario for bubble sort. We recorded the elapsed times of the original and the obfuscated apps for array sizes of 100, 200, 300, 400 and 500. We found that, as the input size of the array increases, the execution time of the obfuscated program increases correspondingly to that of the original program. Figure 10 shows that the time complexity of the obfuscated application is bounded between the time complexity of the original app and twice the time complexity of the original app.
5.3. Potency of Obfuscation

Let us now examine the complexity of our proposed obfuscation schemes. In [8], the potency of an obfuscator is defined as a metric that measures how much more difficult the resulting obfuscated code is to understand (for a human) than the original code.

We derive the potency of our schemes with respect to the attack model we consider below.

5.3.1. Attack Model on Obfuscated Apps

It is important to make clearer of the adverse environment in which an obfuscated app is going to be reverse engineered. We assume that the attacker is the owner of the mobile device on which the app is running, and thus has a complete access to the app bytecode that he/she is interested to analyze. Moreover, we assume that the attacker knows that the app is obfuscated, and he/she aims to reconstruct the actual app control flow using static reverse engineering tools. Hence, we define static analysis attack on obfuscated apps in the context of reconstruction of the original control flow of the apps using available reverse engineering tools.

In our schemes, the difficulty of statically reconstructing the original app control flow depends on successfully deobfuscating the opaque variables that are used in assigning values to the switching variable. [8, 21] discuss about generating strong opaque predicates that may take exponential time (with respect to the size of the program) to break, but can be constructed in polynomial time. Such opaque predicates can be constructed based on the intractability of precise static analysis of pointer-based structures and parallel regions [9].

In what follows, we quantify obfuscation potency in terms of the complexity of reordering a method back to its original condition by assuming the difficulty of deobfuscating opaque variables.

5.3.2. Packed-Switch Obfuscation

The packed-switch obfuscation splits the original code block into n code fragments.

At the end of every code fragment, a value is assigned to a register that decides the next code fragment to execute. Since we assume the usage of opaque variable, the subsequent fragment determination is unresolved by static analysis. As such, there are n! combinations of the shuffled code fragments. An attacker trying to determine the correct order thus has to inspect these n! combinations in the worst case, and hence it is the upper bound to reconstruct the original control flow of the program.

Figure 13 in the Appendix illustrates the CFG of a sample method without any obfuscation. Figure 14 in the Appendix is the CFG of the same method that is obfuscated with packed-switch obfuscation technique. For easy illustration, we split the method into three code fragments.

5.3.3. Try-Catch Obfuscation

For the try-catch obfuscation, we presume that an exception event can only be ascertained at runtime due to the use of opaque variables. Thus, statically analyzing an obfuscated method can only list x potential exception-raising operations (within the try block) and y switch case branches (within the catch block). The complexity of deriving the correct execution flows between the two blocks is thus x · y. Recall again that, within the try block, we can induce any number of exception-raising operations that will never be executed due to the use of opaque predicates. Likewise, we can induce junk switch cases in the catch block.

Figure 15 in the Appendix shows the CFG of the method shown in Figure 13, but obfuscated with the try-catch obfuscation technique. Three instructions within the method are camouflaged with three exception-raising instructions. In the catch block, the original instructions are executed within the three switch-case branches.

5.3.4. Combined Obfuscation

As described in Section 3.4, the combined obfuscation is conducted by first applying the packed-switch obfuscation, and then moving the packed-switch instruction and all other control-flow instructions into the catch block of the try-catch obfuscation. All the shuffled code fragments will be completely inside the try block.

Let us assume that no extra junk switch-case branches are injected. That is, the number of the
Table 5: Size overhead on two tested apps with varying number of code fragments.

<table>
<thead>
<tr>
<th>App</th>
<th>Measurement</th>
<th>Number of formed code fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Unobfuscated</td>
</tr>
<tr>
<td>Winamp</td>
<td>App size (in MiB)</td>
<td>24.8</td>
</tr>
<tr>
<td></td>
<td>No. of obfuscated methods</td>
<td>3,330</td>
</tr>
<tr>
<td></td>
<td>Average size increase per obfuscated method (in KiB)</td>
<td>1.65</td>
</tr>
<tr>
<td>Photo editor</td>
<td>App size (in MiB)</td>
<td>16.8</td>
</tr>
<tr>
<td></td>
<td>No. of obfuscated methods</td>
<td>2,397</td>
</tr>
<tr>
<td></td>
<td>Average size increase per obfuscated method (in KiB)</td>
<td>1.90</td>
</tr>
</tbody>
</table>

switch-case branches for the try-catch obfuscation is the same as the number of potential exception-raising operations ($x$). Since only a single catch block is employed, we now have $n + x$ switch-case branches, each of which corresponds to a code block to be executed. We now have a higher number of executable code blocks than the original packed-switch’s code fragments since a camouflaged instruction within a code fragment has an effect of splitting the fragment into two smaller code blocks. Given the formed $n + x$ code blocks, which would look indistinguishable to an attacker, the upper bound complexity of recovering the actual code-block ordering in the combined obfuscation scheme is thus $(n + x)!$.

Figure 16 in the Appendix depicts the CFG of the sample method obfuscated with the combined obfuscation scheme.

5.4. Cost vs Potency

The strength of the obfuscation and the overhead due to the obfuscation are proportional to each other. As we increase the number of code fragments ($n$) in the obfuscated code, more new instructions are added to redirect the control flow to the packed-switch instruction. Therefore, the size and performance overhead of the program will also increase. We now examine the size and performance overhead of increasing $n$.

We varied $n$ from 3 to 15 when obfuscating Winamp and Photo Editor apps. One problem with increasing $n$ is that, at certain value, some short methods of the apps cannot be split to form the required number of code fragments. For instance, it is not possible to split a method with 6 instructions into 9 equal code fragments. Since our goal is to measure the exact overhead of increasing $n$, app methods that are too short for a specific value of $n$ are exempted from obfuscation. We instead measure all the successfully obfuscated methods, and inspect the overhead trends on these methods.

Table 5 shows how the size of the apps increases as we vary the number of code fragments $n$ in the obfuscation. It can be seen that the number of obfuscated methods is lower when $n$ is higher as explained earlier. Yet, we can see a stable increasing pattern of the average size increase per obfuscated method as we increase $n$. Figure 11 graphically shows this pattern on the two tested apps. It shows that the average size of the obfuscated methods increases linearly as we increase $n$.

As we increase the number of code fragments $n$ in the obfuscation, the incurred performance over-
head will also increase. To calculate the effect of varying \( n \) to the performance overhead, we write a small app that performs a micro benchmark and observe the elapsed times. The app invokes a method `function_time()` for 1,000 times. This method contains hundred lines of arithmetic operations employing Java’s `Math` library calls. Figure 12 shows that the performance overhead increases linearly with \( n \) in the obfuscated method.

6. Discussions

We discuss in this section several aspects related to the applicability and potential limitations of our obfuscation process.

In general, our obfuscation can split a target method at any program point. Nevertheless, there are certain program points that are prohibited for splitting due to the imposed Dalvik bytecode’s structural constraints. For instance, the instruction `move-result vAA` must be immediately preceded by a method invocation instruction `invoke-*` [25]. Hence, our implementation makes sure that `invoke-*` and `move-result*` instructions are not split into two different code fragments.

Another splitting restriction is pertinent to Dalvik pseudo-instructions. Rather than performing an operation, these pseudo-instructions define supplemental data of their respective actual instruction. These pseudo-instructions thus cannot be split. There are three pseudo-instructions in Dalvik bytecode, namely `packed-switch-payload`, `sparse-switch-payload` and `fill-array-data-payload`, which carry data for `packed-switch`, `sparse-switch` and `fill-array-data`, respectively [22]. The `packed-switch-payload`, for instance, lists all jump addresses of a `packed-switch` instruction. Our obfuscation thus does not split the content of pseudo-instructions.

The Dalvik bytecode’s structural constraints [25] additionally stipulates that the three pseudo-instructions must not be reachable by control flow. Hence, when we add a dummy target address, it should not point to the address of a pseudo-instruction within a target method.

The register-type conflict separation is a major contribution of our work. A downside of this technique is that it will lead to a register bloat if the number of registers in a method is already high. This may also affect the runtime performance of the pertinent apps. We do notice, however, that in practice methods are usually written in manageable size with an acceptable number of registers used.

Nevertheless, there may be cases where the number of registers in a method go beyond 256 following the application of our type-separation technique. This poses an issue since several Dalvik instructions, such as `move-result*` and `return*` instruction families, take registers in the range of \( v0 \) – \( v255 \) as operands. To deal with it, we can implement the following enhancement to our type-separation technique. First, we need to reserve three registers in the range of \( v16 \) – \( v255 \) when performing the steps outlined in [22]. Three registers are sufficient since the maximum number of registers taken by such instructions is three, which is in the case of `cmp*`, `aget*`, `aput*`, `binop` and `binop/lit8` instruction families [22]. For each of instructions that cannot take registers above \( v255 \), we then add register assignments prior to (following) the instruction into (from) the reserved high registers. The registers above \( v255 \) in the instruction are then replaced with the assigned reserved registers. As before, we put the instruction and its added assignments as a single unit during the obfuscation process.

This outlined solution works since, for each data type, Dalvik bytecode has a `move` operation that takes registers up to \( v65535 \). The reserved high registers will not raise any type-conflict problem as they are always set before being used. Our current implementation, however, does not apply this technique yet, which can constitute our future work.

Lastly, we run our performance overhead evaluation on an Android Virtual Device, which is available as part of the Android SDK. Our main objective is to quantify the relative variation in the time complexity when an app is run on the same platform in its original and obfuscated forms. This performance overhead measurement will vary according to devices used for running the apps and their underlying hardware platforms.

7. Conclusions

We have analyzed the need for potent control-flow based obfuscation as to help protect Android apps. We also have described the design and implementation of three control-flow obfuscations for Android apps at the Dalvik bytecode level, which go beyond simple control-flow transformations used by existing Android obfuscators. The register-type conflict problem raised by the Android runtime system has also been addressed by means
of our type separation technique. Our experimentation shows that the schemes increase the complexity of app control flow and make it difficult to reverse engineer. Meanwhile, the incurred performance and size overheads are comparable to those achieved by several widely-known obfuscation techniques in the x86 environment. The schemes thus can work together with other known Android obfuscation and app tamper-proofing techniques to help safeguard Android apps against the increasingly rampant tampering and repackaging.

Appendix A: Control Flow Graphs of a Sample Method and Its Obfuscated Versions.

To give a better illustration on the effects of applying our obfuscation schemes, this Appendix gives the CFG of a sample method to be contrasted with those of the obfuscated versions. We used Androguard tool [17] to generate the graphs.

Figure 13: The CFG of the original sample method.

Figure 13 shows the CFG of the original (non-obfuscated) sample method. Figure 14 is the CFG of the sample method that is obfuscated with packed-switch obfuscation technique. For easy illustration, we split the function into three code fragments. Figure 15 shows the CFG of the method that is obfuscated with the try-catch obfuscation technique. Finally, Figure 16 depicts the CFG of the sample method that is obfuscated with the combined obfuscation scheme.

Acknowledgment

This material is based on research work supported by the Singapore National Research Foundation under NCR Award No. NRF2014NCRNCR001-034.

References

[24] Android Open Source Project, ART and Dalvik - Con-
Figure 14: The CFG of the sample method that is obfuscated with the packed-switch obfuscation.

Figure 15: The CFG of the sample method that is obfuscated with the try-catch obfuscation.


Figure 16: The CFG of the sample method that is obfuscated with the combined obfuscation scheme.

60–72.


[50] class_linker.cc, https://android.googlesource.com/platform/art/*/android-5.0.1_r1/runtime/class_linker.cc


