University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

12-2016

SEMEO: A SEMANTIC EQUIVALENCE ANALYSIS FRAMEWORK FOR OBFUSCATED ANDROID APPLICATIONS

Zhen Hu

University of Nebraska-Lincoln, hiji1232@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

🔵 Part of the Computer Engineering Commons, Information Security Commons, and the Software Engineering Commons

SEMEO: A SEMANTIC EQUIVALENCE ANALYSIS FRAMEWORK

FOR OBFUSCATED ANDROID APPLICATIONS

by

Zhen Hu

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professors Gregg Rothermel and Witawas Srisa-an

Lincoln, Nebraska

December, 2016

SEMEO: A SEMANTIC EQUIVALENCE ANALYSIS FRAMEWORK

FOR OBFUSCATED ANDROID APPLICATIONS

Zhen Hu, MS

University of Nebraska, 2016

Adviser: Gregg Rothermel, Witawas Srisa-an

Software repackaging is a common approach for creating malware. In this approach, malware authors inject malicious payloads into legitimate applications; then, to render security analysis more difficult, they obfuscate most or all of the code. This forces analysts to spend a large amount of effort filtering out benign obfuscated methods in order to locate potentially malicious methods for further analysis. If an effective mechanism for filtering out benign obfuscated methods were available, the number of methods that must be analyzed could be reduced, allowing analysts to be more productive. In this thesis, we introduce SEMEO, a highly effective and efficient filtering approach that can determine whether an obfuscated and an original version of a method are semantically equivalent. Our approach handles seven common, complex types of obfuscation and can be effective even when all types are compositely applied. In an empirical evaluation, we applied SEMEO to nine Android apps of varying complexity, and the approach provided over 76% recall and 100% precision in identifying semantically equivalent methods. We then performed three additional studies, that showed that: (1) SEMEO is much more effective at identifying semantically equivalent methods than FSQUADRA, an existing technique; (2) SEMEO is also effective for identifying repackaged apps that have been previously obfuscated by ProGuard, a popular obfuscation tool; and (3) SEMEO is effective at identifying semantically equivalent methods in a repackaged, malicious version of *Pokémon Go.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code obfuscation is a technique commonly used to render code difficult for humans or analysis software to comprehend. Traditionally, developers used obfuscation to protect code in its role as intellectual property. For example, most apps that can be downloaded from Google Play are obfuscated by ProGuard [17]. However, increasingly, obfuscation is also being used by malware authors to hide malicious payloads through a process known as "repackaging" [42]. In repackaging, a legitimate application is modified by adding code that performs malicious behavior. The application is then obfuscated to make it difficult to locate the malicious code [6]. Currently, repackaging (including its use of obfuscation) is the leading approach employed to create Android malware [18]. In 2015, 2.3 million new malicious apps were uncovered – a rate of one new malware every 13 seconds [14]. To combat malware effectively, security analysts need techniques that can efficiently and effectively detect repackaged malware. Because both benign and malicious apps are now commonly obfuscated, these techniques must be able to cope with code obfuscation. In addition, analysts need techniques that can effectively identify the locations of malicious code within repackaged malware so that they can perform additional analysis to understand that

code's behavior and mitigate its effects. To accomplish these two goals, such techniques must be able to handle sophisticated classes of obfuscation techniques including code injection, code reordering, function indirection, function inlining and function outlining [34].

Unfortunately, the most recent approaches presented for detecting repackaged malware are not effective at identifying locations of malicious code. For example, DROIDMOSS is effective at detecting repackaged malware. It has been used to investigate multiple market places for repackaged apps. The approach is based on fuzzy hashing [41]. However, it is not effective when obfuscation techniques are used. Zhang et al. [39] handle obfuscation by relying on high-level interconnections among *user interfaces* (*UIs*), and Zhauniarovich et al. [40] use resource usage within the analyzed apps to detect differences between an original app and a suspected repackaged app. Through reliance on UI interconnections and resource usage, these approaches cope successfully with code obfuscation because they do not rely on code-level *birthmarks* [28, 35] to perform detection. Instead, they relies on other birthmarks to detect behavioral differences at the UI or resource usage levels. As such, they are effective at detecting repackaged apps that are resource, UI, and event intensive. They are ineffective, however, if the malicious code affects only functionalities and does not cause UI or resource usage to change.

One way to render the process of analyzing repackaged malicious code more effective and efficient is to identify, from among all obfuscated code segments, those that are semantically equivalent to the original, unobfuscated code. The code thus identified does *not* need to be further analyzed, so this allows analysts to focus on the code that remains. To accomplish this goal, researchers and practitioners have attempted to apply deobfuscation techniques to obfuscated methods, and then use "differencing" techniques to compare the deobfuscated versions to the originals

(e.g., [3, 4, 5, 15, 16, 26]; Chapter 8 discusses this related work). This approach, however, has weaknesses. First, increasingly sophisticated obfuscation types are being created, such as those that alter program structure, and these can hobble deobfuscators (Chapter 2 discusses this further). Second, even when deobfuscators are able to function, they do not necessarily retrieve code matching the original code; instead they focus on re-engineering code into a format that is more easily understood by engineers.

In this thesis we explore an alternative approach. We present SEMEO, an effective and efficient technique for identifying methods in an obfuscated application that are semantically equivalent to methods in an original application. The remaining methods are potentially non-equivalent and analysts can focus on those. Our approach differs from prior approaches in that, to our knowledge, it is the first to attempt to directly identify (without first deobfuscating) obfuscated methods in Android apps that are semantically equivalent to original non-obfuscated methods. SEMEO also handles a broad range of complex and widely used obfuscation techniques; techniques that are currently being employed by authors of malware on Android apps in particular.

We present the results of an empirical evaluation assessing the efficiency and effectiveness of SEMEO, in which we applied it to nine Android apps of varying complexity. Our evaluation reveals that the approach can achieve between 76% (when five obfuscation types are compositely applied) and 100% (when one or two types are applied) recall with respect to the numbers of obfuscated methods. Given a recall of 76%, analysts need to apply deobfuscation and further analysis techniques only to the remaining 24% of the methods. In addition, modified methods misidentified as semantically equivalent can be particularly damaging to security analysis as they may be overlooked by analysts. In our empirical study, however, our approach

achieved 100% precision (i.e., there was no misidentification of non-equivalent methods as equivalent) in all cases, even though each app considered was obfuscated in 325 different ways.

We also present the results of three additional empirical studies. In the first, we compare SEMEO to FSQUADRA, an existing tool for detecting repackaged apps, and we find that SEMEO is much more effective. In the second study, we apply a second layer of obfuscation to the apps studied in our initial evaluation using ProGuard; and our results show that even in this case, SEMEO remains highly effective, while continuing to outperform FSQUADRA. In the third study, we apply SEMEO to a repackaged, malicious version of the non-trivial real-world app, *Pokémon Go*, and again we find that SEMEO is effective.

The remainder of this thesis is organized as follows. Chapter 2 provides background information on Android Dalvik instructions and obfuscation techniques. Chapter 3 describes SEMEO in detail, including its overall analysis workflow and detailed algorithms for each of its steps. Chapter 4 describes our empirical study and answers three research questions about our approach. Chapter 6 provides additional discussion of our results. Chapter 7 presents results of our three additional empirical studies. Chapter 8 discusses related work, and Chapter 9 concludes.

Chapter 2

Background

We next provide background information on the Android Runtime System, the DEX instructions it relies on, and common obfuscation methods.

2.1 The Android Runtime System and DEX Instructions

The Android software system consists of four layers: a Linux kernel, Libraries, an Application Framework, and Applications. Android apps use either the Dalvik virtual machine (VM) or Android Runtime (ART) environment to execute code in DEX format.

DEX code is a register-based machine language. Each DEX method has its own predefined number of virtual registers; these registers correspond to variables that can store primitive types and object references. The execution engine stores the states of method registers in an internal execution state stack, with the most current method's register on top of the stack. Because all the operations and computations performed are register-based, all values must be loaded from and stored into class fields [2].

Figure 2.1 presents an example of Android DEX instructions from the Android app DRAGON, obtained by applying a dexdump tool. The first line contains the package name (`com.example.dragon`), class name (`MainActivity`), and method name (`checkHiTone`). This strictly follows JVM specifications[37]. Subsequent lines display the offsets of each DEX instruction, then detailed instructions and register information. The code shown first obtains a field from `Lcom/example/dragon/Conditions` and stores it in register v0 via an `sget`. The code then compares the contents of register v0 to the contents of register v1. A detailed explanation of DEX instructions is provided in the Android specification [2].

```
|[049a5c] com.example.dragon.MainActivity.checkHitone:()V
|0000: sget v0, Lcom/example/dragon/Conditions;.x:I // field@052c
|0002: const/4 v1, #int 2 // #2
|0003: if-le v0, v1, 001a // +0017
|0005: sget v0, Lcom/example/dragon/Conditions;.y:I // field@052d
|0007: const/16 v1, #int 19 // #13
|0009: if-ge v0, v1, 001a // +0011
|000b: sget v0, Lcom/example/dragon/Conditions;.z:I // field@052e
|000d: const/4 v1, #int 1 // #1
|000e: if-ge v0, v1, 001a // +000c
|0010: const-string v0, "Congrats,target FOUR is reached!!" // string@021c
|0012: const/4 v1, #int 0 // #0
|0013: invoke-static {v2, v0, v1},
ent/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast; // method@13b4
|0016: move-result-object v0
|0017: invoke-virtual {v0}, Landroid/widget/Toast;.show:()V // method@13b5
|001a: return-void
```

Figure 2.1: Examples of DEX instructions

2.2   Obfuscation Methods

Rastogi et al. [34] classify common obfuscation techniques into three categories: (1) trivial obfuscations, which can be easily detected by most antivirus tools; (2) DSA obfuscations, which can be detected by static analysis techniques; and (3) NSA ob-

fuscations, which cannot be detected by static analysis. In this thesis we focus on DSA obfuscations. According to Rastogi et al. [34], most commercial antivirus tools cannot cope with DSA obfuscations, as this class of obfuscations typically changes the control flow and data flow of programs. We focus further on five specific classes of DSA obfuscations: junk code insertion, code reordering, method indirection, function inlining and function outlining [8, 22, 23, 32, 38]. These five classes of DSA obfuscations are those most commonly found in practice [34].

Junk code insertion involves the insertion of unnecessary code into an app. The additional code may execute but does not affect the behavior of the program.

The three most common types of junk code insertion are nop insertion, branch insertion and garbage code insertion. Nop insertion simply adds sequences of nop instructions to the code; this obfuscation type is easy to detect and remove [23]. Branch insertion introduces branch instructions based on simple templates; for example, branch predicates can be added that are always false so that the branches are never actually taken. This obfuscation type may create additional dependencies in control flow analysis [38]. Garbage insertion is also called dead code insertion; this involves inserting instructions (other than nop instructions) that has no effect on the semantics of the code [32].

Code reordering involves changing the execution order of statements or blocks of code. This obfuscation type can be difficult to detect and remove. Because changing the execution order of statements or blocks of code can affect the information flow in a program it can also render it difficult to determine whether code thus obfuscated is semantically equivalent to the original code. Figure 2.2 illustrates an application of code reordering to the code of the Dragon app originally shown in Figure 2.1. The original code tests whether the value in v0 is less-than-or-equal-to 2 in Lines 2 and 3,

```
|[049a5c] com.example.dragon.MainActivity.checkHitone:()V
|0000: sget v0, Lcom/example/dragon/Conditions;.y:I // field@052d
|0002: const/16 v1, #int 19 // #13
|0004: if-ge v0, v1, 001a // +0016
|0006: sget v0, Lcom/example/dragon/Conditions;.x:I // field@052c
|0008: const/4 v1, #int 2 // #2
|0009: if-le v0, v1, 001a // +0011
|000b: sget v0, Lcom/example/dragon/Conditions;.z:I // field@052e
|000d: const/4 v1, #int 1 // #1
|000e: if-ge v0, v1, 001a // +000c
|0010: const-string v0, "Congrats,target FOUR is reached!!" // string@021c
|0012: const/4 v1, #int 0 // #0
|0013: invoke-static {v2, v0, v1},
ent/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast; // method@13b4
|0016: move-result-object v0
|0017: invoke-virtual {v0}, Landroid/widget/Toast;.show:()V // method@13b5
|001a: return-void
```

Figure 2.2: An illustration of code reordering

then tests whether the value of v1 is greater-than-or-equal-to 19 in Lines 7 and 9. The reordered code reverses the order of these tests.

Method indirection inserts additional calls into an app, and is an obfuscation type designed to manipulate call graphs. With this approach, a given method call (e.g., $m_0 \rightarrow m_1$) can be converted to a call to a previously non-existing method (e.g., m_2) that then calls the originally called method; (e.g., yielding $m_0 \rightarrow m_2 \rightarrow m_1$). The technique is applicable to calls to framework libraries as well as calls to methods within an app [22].

Function inlining replaces method calls with the actual body of called methods. Normally used by compilers for optimization, this obfuscation type breaks abstraction boundaries created by the programmer [8].

Function outlining is the inverse of function inlining; it involves decomposing a function into multiple smaller functions. This process has been used (non-maliciously) to remove duplicate code in large programs [22]; in the context of obfuscation, its strength lies in requiring interprocedural analyses to perform deobfuscation.

Chapter 3

Semantic Equivalence Analysis of Obfuscated Code

We now present our approach for Semantic Equivalence Analysis of Obfuscated Code (SEMEO). The key objective of SEMEO is *to provide an efficient technique for determining whether a method that has been obfuscated is semantically equivalent to the original version of the method.* Because a repackaged app typically includes only a small set of methods that have been semantically altered to enact malicious behavior, the majority of that app's obfuscated methods will be equivalent to the original unmodified methods. If our approach is effective, it should be able to identify a larger percentage of these semantically equivalent methods, allowing security analysts to focus on methods that cannot be conclusively identified as semantically equivalent.[1]

Figure 3.1 provides an architectural overview of SEMEO's workflow. An analyst provides two apps to SEMEO: an app P and a version P' of P that is suspected to have been repackaged. SEMEO compares methods in P' (m'_j) to methods in P (m_i). Note, however, that the mapping of methods in P' to methods in P may not

---

[1]In general, the problem of determining the semantic equivalence of two programs is undecidable [19], so our approach is necessarily an heuristic.
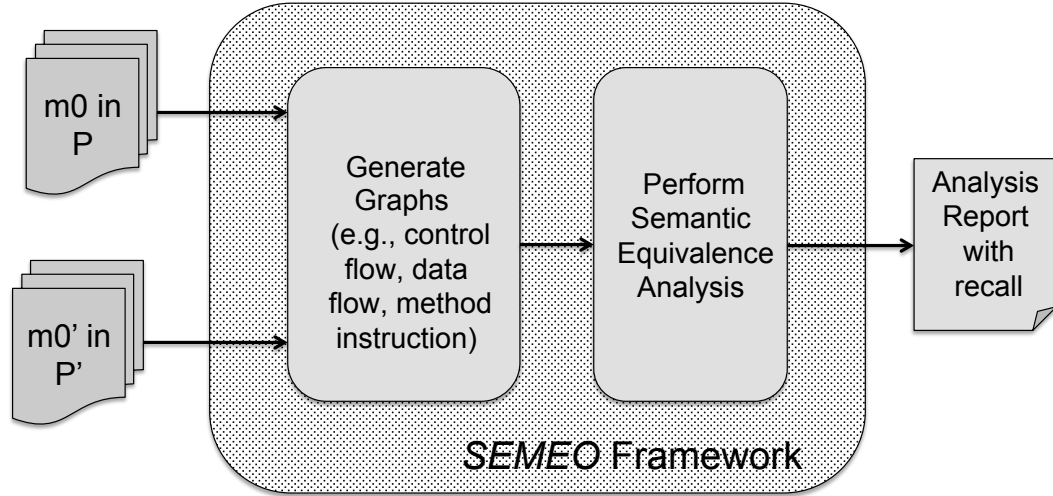
Figure 3.1: Architectural overview of SEMEO

be one-to-one due to the use of obfuscation techniques that merge methods, extract new ones, or make it difficult to determine which obfuscated methods correspond to which original methods. Thus, the approach must account for this.

For example, suppose that P contains two methods, (m_0 and m_1), and suppose that P', a repackaged version of P, contains three methods (m'_0, m'_1, and m'_2). Suppose that the additional method has been created by the use of a method outlining obfuscation technique that splits m_1 into m'_1 and m'_2. SEMEO begins by comparing m_0 to m'_0. If they are not found to be semantically equivalent, it then compares m_0 to m'_1, and so on. If, on the other hand, they *are* found to be semantically equivalent they are marked as such. SEMEO does not now need to visit m_0 again; instead it compares m_1 with m'_1. In this case in which the two modules are not semantically equivalent; m'_1 now calls m'_2 so the analysis needs to consider both methods ($m'_1 + m'_2$) and then evaluates whether the combined result is semantically equivalent

to m_1. Similarly, if inlining is used to obfuscate two methods, there may be a situation in which two or more methods in P are, together, semantically equivalent to a method in P'.

When SEMEO completes its analysis, it outputs a list of methods that have been determined to be semantically equivalent and not equivalent, and a percentage indicating what proportion of the methods were determined to be semantically equivalent. In the rest of this section we describe each step of the process in turn.

3.1   Preliminary Information

SEMEO operates on DEX instructions, and its goal is to identify DEX instructions that can potentially change the semantic meaning of an app that uses them. Thus, before presenting our approach we present information on these instructions. Table 3.1 shows categories and examples of DEX instructions, derived from the Android Specification [2]. Some of these classes of instructions cannot alter the semantics of the app; these are shown in plain font at the end of the table.

As an example of an instruction that can change the semantics of an app, if a cybercriminal modifies a method by injecting a few `read` instructions such as `iget`, these instructions would perform field accesses and store the retrieved values into value registers. These operations, in effect, overwrite these value registers with new and potentially incorrect values, thus infecting the app. These values, however, remain dormant until they are propagated via operations that use them as operands or store them in other registers (e.g., `move`) or object fields (e.g., `iput`) that can be used by others. We refer to instructions that can cause infection and propagation to be "suspicious DEX instructions", and these are the targets of our analysis. Other instructions, shown in italics in Table 3.1, can also change the semantics of the app:

Table 3.1: DEX Instruction Categories

Instruction Category	Examples
Invoke	invoke static, invoke virtual
Read	iget, aget, sget
New	new array, new instance
Array	fill new array, fill array data
Write	iput, aput, sput
Move	move
Arithmetic op	binary, unary operation
Branch	if, go to, switch
Return	return
Comparison	if, ifz, cmp
Constant	const wide, const, const string
Exception	throw
No op	nop
Casting	check_cast, instance of
Synchronization	monitor enter, monitor exit

the `write` instruction, the `new` operation, and arithmetic operators are examples.

Notice that we classify `throw` instructions as one of those that cannot change the semantics of a method. Exception handling occurs at runtime and modifying code by adding `throw` instructions does not always change control flow – a change can occur only if the exception occurs and is caught. Due to the dynamic nature of the `throw` instruction, we consider it to be an instruction that does not change the semantics of the code as part of our static analysis.

SEMEO compares a pair of apps at the method level. To support the necessary analysis we created a tool to construct data flow and control flow graphs for the instructions in each method, and method graphs to help track method calls. Data flow are generated by using reaching definition algorithm, and the flow information are based on def-use pair. We also created tools to construct class graphs and field graphs (discussed later in this chapter). While it is possible to use existing program analysis tools such as SOOT to perform the required analyses, we chose to create

tools that work directly with DEX to minimize the amount of translation that must be performed and can potentially add another layer of complication to the obfuscated code (e.g., DEX to Jimple for Soot) [25]. We provide a simple illustration of our analysis graphs in Figure 3.2, which shows an instruction graph representing a method with data flow information.
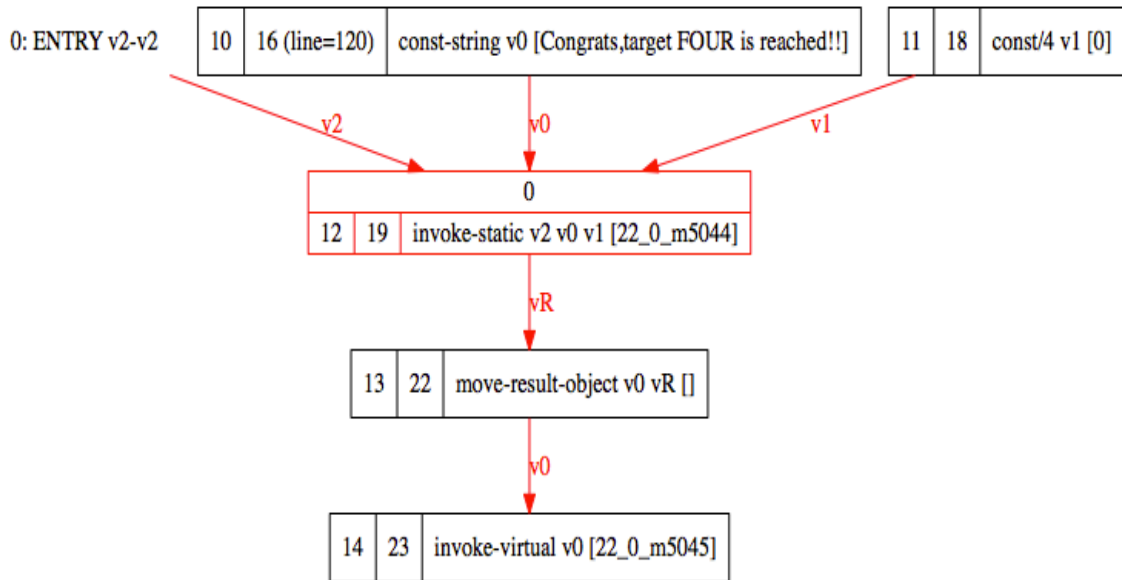
Figure 3.2: Example of an instruction graph with data flow information

3.2 Algorithms

We now describe the algorithms that comprise SEMEO. Algorithm 1 shows the procedure that is applied to a pair of methods Mi and Mj. In Line 2, the algorithm determines whether a method semantically equivalent to M_i has been previously found. It does this using a map bit to represent the status for each method. If the map bit for M_i is set, a semantically equivalent method in P' has already been found.

Algorithm 1 Equivalence Analysis

1: **procedure** CHECKEQUIVALENCE:(Mi, Mj')
2:     **if** (CheckMapBit(Mi) == false) **then**
3:         $Gi \leftarrow$ ExtractDataFlow(Mi)
4:         $Sum1 \leftarrow$ ExtractInstructionSummary(Gi)
5:         **if** (CheckMapBit(Mj') == false) **then**
6:             $Gj \leftarrow$ ExtractDataFlow(Mj')
7:             $Sum2 \leftarrow$ ExtractInstructionSummary(Gj)
8:             **if** SummariesMatch($Sum1, Sum2$) **then**
9:                 SetMappingFlag(Mi, Mj')
10:            **end if**
11:        **end if**
12:    **end if**
13: **end procedure**

If it is not set, the analysis continues (Line 3). The algorithm analyzes the data flow graph for Mi (Line 3) and produces its instruction summary (Line 4).

Procedure ExtractInstructionSummary (called in Lines 4 and 7) traverses the data flow graphs for each of the methods being considered using the DFS visitor in the *Boost graph library*, a graph processing library available on most computing platforms [1]. By using DFS visitor, SEMEO transverse each node in data flow graph from entry point and check the suspicious instruction. Except *ENTRY*, each node has an incoming edge with registers based on the data flow information. If that incoming register is not in the path summary, then this incoming register will be added into path summary. Path summary is a map to record the register and its dex operation pair for each path. After that, SEMEO will check each node for suspicious instruction, and update the path summary. When it visits the last node of each path, it will combine current path summary into instruction summary, then clear the current path summary and visit another new path. By running DFS visitor, SEMEO traverse all paths on the data flow, extracting suspicious DEX instructions and constructing an instruction summary for each method. As such, each instruction summary contains only suspicious DEX instructions. Each instruction consists of

three components: an opcode, registers and a constant value.

The information contained in an instruction summary represents the behavior of a method but not its full structure. Each suspicious DEX instruction is a node in the summary. Because DEX is a register-based instruction set, data flows via registers and each flow of data can be represented as an edge between two nodes. Because method arguments enter a method via entry registers set up by the caller, there are no assignment edges coming into entry registers.

Turning again to the instruction graph in Figure 3.2, an argument is passed in to the method through register $v2$, as indicated by the keyword *ENTRY*. There are three instructions in this graph, `invoke-static`, `move-result-object`, and `invoke-virtual`. All three instructions are classified as "suspicious DEX instructions" and thus they are included in the summary.

In Line 5, the algorithm checks whether Mj' has already been determined to be semantically equivalent to an original method. If not, the algorithm extracts its data flow graph and associated instruction summary (Lines 7-8). Now, both summary sets are compared (Line 8) and if they are semantically equivalent, a map bit is set for (Mi, Mj).

The analysis of an instruction summary proceeds as follows. First, the analysis looks for `invoke` instructions in the summary. In the DEX instruction set, `invoke` is used to perform method calls. Obfuscation techniques such as method indirection, function inlining and function outlining use `invoke` instructions to perform obfuscation by making additional method calls. To handle this situation, our technique, as shown in Algorithm 2, performs an interprocedural analysis of the caller and callee methods to perform *summary inlining*. For each previously computed summary, whenever there is an `invoke` instruction (Line 3), the algorithm visits the callee method (Line 4) and checks whether that method has ever been inlined with

---

Algorithm 2 Summary Inlining

 1: **procedure** INLINING($Sum, Inlined$)
 2:     **for** each instruction Ins in Sum **do**
 3:         **if** Ins is "invoke" **then**
 4:             $M \leftarrow$ `GetCalleeMethod(Ins)`
 5:             **if** NotInList(M,$Inlined$) **then**
 6:                 $Inlined \leftarrow Inlined \cup M$
 7:                 $G \leftarrow$ `ExtractDataFlow(`M`)`
 8:                 $Sum_i \leftarrow$ `ExtractInstructionSummary(`G`)`
 9:                 $Sum \leftarrow Sum \cup Sum_i$
10:                 $Sum \leftarrow Sum - Ins$
11:             **end if**
12:         **end if**
13:     **end for**
14: **end procedure**

---

the caller's summary (Line 5). If it has, the analysis has been done previously and the algorithm terminates. If it has not, the algorithm uses the DFS visitor method to compute an instruction summary for that callee method (Lines 5 to 8). Finally, it merges the result with the summary of the caller method (Line 9). This process also applies when a callee method makes calls to other methods. After merging summary information, the algorithm removes the invoke instruction from the caller's summary (Line 10). This process is repeated until there are no more invoke instructions in either the caller's or callee's summaries.

In the scenario presented earlier in our example, method outlining has been used to obfuscate P to create P'. In this case, m_1 has been obfuscated to create m_1' and m_2' through outlining. Algorithm 2 first analyzes the summary of m_1' to identify `invoke` instructions. In this case, there is one `invoke` call to m_2'. Since this is the first time this summary has been analyzed, the algorithm creates a summary of m_2' and merges it with that of m_1'. The combined summary is then compared to that of m_1.

After performing summary inlining, SEMEO compares the summaries to determine whether they are semantically equivalent. The comparison process applies

several heuristics. For example, by using the instruction summary, the algorithm can remove all junk code that has been inserted without changing the semantics of the program. This is because our "suspicious DEX instructions" already cover a wide range of semantics-changing instructions. However, considering just these suspicious instructions is not sufficient. Obfuscation approaches such as those that change loop structures would cause the instructions and data-flow patterns to change. To handle this, we also refer to common loop patterns and look for cases where a pattern has been changed to another equivalent pattern. We then perform template matching to see if the obfuscation is simply changing the loop structure. For example, for loops and while loops may have the same semantic meanings in different flow structures. To handle this, by traversing the data flow in both situations, our DFS visitor will check all possible conditions and include suspicious DEX instructions into the instruction summary until it reaches a fixed point. As our instruction summary only involves in register information and DEX instructions, the structural difference will not affect our result.

Sum and Sum_i are lists of maps, where each map stores the register and suspicious DEX instructions from the previous DFS visitor execution. The next step is to compare both instruction summaries. To handle code reordering, we sort Sum and Sum_i alphabetically in terms of their DEX instructions before beginning to compare them. Instruction reordering is difficult to analyze if we retain the order of the instructions as found in the summary. Sorting takes care of this concern as order is no longer preserved based on when instructions appear in the method. Instead, we preserve relationships through data flow information. After comparing instructions and data flow information, our approach analyzes constant values, which may provide additional insights as some constant values including strings may not change as they provide specific information for the methods (e.g., URL strings, constant integers).

After visiting all of the methods in both apps, SEMEO examines the analysis result obtained for the obfuscated app and then calculates the percentage of methods in the obfuscated app found to be semantically equivalent to those in the original app. If this number is less than 100%, SEMEO outputs the names of all methods in the obfuscated app that are not found to be semantically equivalent to methods in the original app.

The complexity of the comparison process as represented in Algorithm 1 is $O(n^2)$ in the worst case, and $O(n)$ in the best case, where n is the larger of the number of methods in P and P'.

Chapter 4

Empirical Study

To evaluate SEMEO we conducted an empirical study, considering the following research questions.

RQ1. How effective is SEMEO at detecting whether an app and a semantically equivalent obfuscated version of that app are in fact semantically equivalent?

RQ2. How effective is SEMEO at identifying repackaged methods in obfuscated apps?

RQ3. How efficient is SEMEO?

4.1   Objects of Analysis

To answer these research questions, we wished to obtain several Android apps of varying complexities, for which Java source code and build procedures were available. Ultimately, we selected nine apps; Table 4.1 provides details. Column 1 provides an app number that is used later, Column 2 provides the app name, Column 3 lists the number of methods in each app, and Column 4 lists the number of lines of code.[1] (We

---

[1]Lines of code were counted manually by adding up the total lines of source code for each file, as reported in the Eclipse IDE.

Table 4.1: Applications

App	Name	Methods	LOC	Modified Methods
1	PicViewer	21	139	2
2	CalcC	63	462	6
3	DeviceAdmin2	161	1675	20
4	Orienteering	697	10246	20
5	SysMon	752	3490	18
6	Pondl	1573	20664	99
7	YARR	2027	1224	57
8	NewsCollator	2935	3535	19
9	TextSecure	7218	37486	243

discuss the rightmost column later.) These apps were created by DARPA to support their Automated Program Analysis for Cybersecurity (APAC) program [13]. As the table shows, the apps ranged in size from 21 to 7218 methods, and from 139 to 37,486 lines of code as reported by Eclipse IDE.

We considered the seven DSA obfuscation types discussed in Chapter 3; these are listed in Table 4.2, where we provide a "Type ID" for use in subsequent references, and the name of the obfuscation. Most of these obfuscation types are known to be difficult for deobfuscation tools to handle. To apply the first five obfuscation types (T1-T5) we used ALAN [27], an Android malware obfuscation engine capable of applying one or more of these types to a given app in any order. ALAN can be configured to obfuscate only a portion of an app or the entire app. We chose to obfuscate entire apps to create a scenario similar to the one created by malware authors when they repackage apps.

We were unable to find any tool support for the function inlining and outlining obfuscation types (T6 and T7), so for these we enlisted the help of an undergraduate student who at that time had no knowledge of our approach for determining semantic equivalence. Where inlining is concerned, we instructed the student to inline string

Table 4.2: Obfuscation Types

Type ID	Obfuscation Name
T1	insert `nop` operation
T2	insert branch
T3	insert garbage
T4	reorder code
T5	method indirect
T6	function inlining
T7	function outlining

operations and the contents of called methods. Where outlining is concerned, we instructed the student to group branch condition bodies into other methods and to move some parts of functions into other small functions. The student applied these modifications to randomly selected methods from each of the apps. We then considered instances in which just inlining, just outlining, or both were applied. Table 4.3 provides information on the numbers of methods inlined and outlined. The table provides the app number and name in Columns 1 and 2, together with the number of methods on which inlining, outlining, and both ("hybrid") were performed for each app (Columns 3 through 5, respectively).

Table 4.3: Numbers of Inlined and Outlined Methods

App	Name	Inlined Methods	Outlined Methods	Inlined+Outlined Methods
1	PicViewer	2	2	2
2	CalcC	6	6	6
3	DeviceAdmin2	10	16	10
4	Orienteering	20	20	20
5	SysMon	10	10	10
6	Pondl	10	10	10
7	YARR	2	2	2
8	NewsCollator	2	2	2
9	TextSecure	10	10	10

ALAN is able to apply obfuscation types T1 - T5 individually or in any combinations, so we chose five different methods for grouping obfuscation types (Table 4.4). Grouping $G1$ considers single obfuscation types; since there are five obfuscation types this yields cases in which just obfuscation type T1 is applied, cases in which just obfuscation type T2 is applied, and so forth. Grouping $G2$ considers all pairs of obfuscation types; "T12" refers to the case in which obfuscation type T1 is applied followed by obfuscation type T2. The order in which obfuscations are applied also matters, so we considered all sequences of pairs (e.g., we also considered "T21"). In the case of Grouping $G2$, then, a total of 20 different sequences of obfuscations are applied. Similar reasoning applies to Groupings $G3$, $G4$, and $G5$, which involve all possible sequences of applications of all possible combinations of three, four, and five obfuscation types, respectively. For function inlining and outlining, we applied each singly and applied both together; thus, there are three different sequences of applications of these obfuscation types, which we refer to as G6 through G8.

Table 4.4: Obfuscation Type Groupings

Grouping	Example Grouping and Sequence	Number
G1	T1, T2, T3, T4, T5	5
G2	T12, T23, T34, T45, T21...	20
G3	T123, T345, T251, T231...	60
G4	T1234, T1245, T4213...	120
G5	T12345, T12453, T45213...	120
G6	T6	1
G7	T7	1
G8	T6+T7	1

To address RQ2 we require repackaged apps. Ultimately, when considering malware, we are interested in methods into which malicious code has been injected, and to which obfuscations have then been applied. Finding suitable malware samples that meet our requirements for objects of study, however, is challenging. For ex-

ample, many malware samples that have been used in academic research are quite old, and they often are not available with source code, which is needed to perform obfuscation. In addition, we also require the source code of the original as well as the repackaged apps for our studies. Finally, finding numbers of modified methods containing malware that are adequate to support any quantitative conclusions about the effectiveness of our approach would likely not be possible.

For these reasons, for the purpose of this initial study, we chose to modify our objects of analysis ourselves. (In Chapter 7 we present a case study in which we apply our approach to an app that does contain actual malware). This gave us the ability to use versions of methods that have been semantically modified in diverse manners, in numbers sufficient to support quantitative conclusions. We asked an undergraduate student to perform this task. To do this, for each app, prior to applying any obfuscation types to the apps, he randomly selected a number of methods and manually modified their code. Modifications involved relatively simple but provably semantics-affecting changes such as negating branch conditions, changing input parameters, removing method contents, and changing return type. While these modifications do not involve insertions of malicious code, we argue that such code would most likely be more complicated than these modifications; thus, if SEMEO is able to correctly deduce that our modified methods are indeed semantically different from the original methods, it is likely to be able to do so for methods involving actual malicious code, where the changes are more extensive. The numbers of modified methods created and used in our study are shown in the rightmost column of Table 4.1.

4.2 Variables and Measures

4.2.1 Independent Variables

Because SEMEO may perform differently on different obfuscation type groupings and we wish to assess such differences in performance, we treat obfuscation type groupings as an independent variable. As noted earlier, our groupings consider each of the obfuscation types separately, while also considering all possible sequences of the obfuscation types that are supported by ALAN.

4.2.2 Dependent Variables

As dependent variables, we chose metrics appropriate to our research questions, as follows.

Recall. For RQ1 and RQ2 we measure *recall*, which represents SEMEO's ability to identify semantically equivalent methods. Recall is calculated as $\frac{m_{\text{seq}}}{m_{\text{total}} - m_{\text{mod}}}$, where m_{total} represents the total number of methods in an app; m_{mod} represents the number of modified methods in the app; and m_{seq} represents the number of methods in the app identified as semantically equivalent. In the case of RQ1, we use apps that have been obfuscated but not modified, so in this case m_{mod} equals 0. For RQ2 recall is calculated using the same equation but in this case the apps have been modified, so m_{mod} is non-zero.

Precision. Precision represents SEMEO's ability not to mis-identify methods as semantically equivalent that are not in fact semantically equivalent. Modified methods mis-identified as semantically equivalent can be particularly damaging to security analysis as they may be overlooked. For RQ1, where we do not have modified methods, the notion of precision does not apply. For RQ2, however, we do have

modified methods, so in that case we also calculate precision. We calculate precision as $\frac{m_\text{seq} - m_\text{neq}}{m_\text{seq}}$, where m_seq represents the number of methods in the app identified as semantically equivalent, and m_neq represents the number of modified methods that have been mistakenly identified as semantically equivalent to methods in the original app.

Efficiency. We calculate efficiency by measuring the time required by SEMEO to perform its analysis. We use seconds to report our results. The measurement begins at the time we load the two apps and ends when the analysis result is reported.

4.2.3   Study Operation

To address RQ1, we use SEMEO to compare the obfuscated apps with the original apps, and note how many methods in the obfuscated apps are flagged as semantically equivalent to the original ones. To address RQ2 we apply the same process, but in this case we also note how the results relate to methods that were actually modified. To address RQ3 we follow the same process as for RQ1 and measure the amount of time needed to perform the analysis.

To perform this study we used a MacBook Pro running OS X El Capitan version 10.11.2, with an 8GB memory and a 2.5GHz Intel Core i5. The performance times we gather are from within this environment.

4.3   Threats to Validity

External validity concerns the extent to which results may generalize. Where external validity is concerned, we have studied only nine apps, but they do represent an important sub-class of the apps that malware authors target, and they do vary in size and complexity. Further, two of our obfuscation types, inlining and outlining,

were applied by hand. Additional studies are needed to address these threats. We also do not consider actual malware, instead using semantic modifications made by a programmer. One of the further studies we present in Chapter 7, however, helps address this threat by considering an application that does contain actual malware.

Internal validity concerns whether the observed results can in fact be attributed to differences among the choices of independent variables. Where internal validity is concerned, errors in the tools we rely on could affect our results, but we have attempted to rigorously test them.

Construct validity concerns the extent to which the measures utilized capture the true costs and values associated with an approach. Where construct validity is concerned, we measure precision, recall, and analysis time, but we do not collect any measures related to actual engineer effort.

Chapter 5

Results

We now report the results of our empirical evaluation, discussing each research question in turn.

5.1   RQ1

RQ1 concerns the effectiveness of SEMEO at detecting whether an app and a semantically equivalent obfuscated version of that app are in fact semantically equivalent. Figure 5.1 presents boxplots showing the distribution of recall values achieved by SEMEO for obfuscation type groupings $G1$ through $G5$ on all nine apps. In the figure, the x-axis organizes the data per app. For each app, five boxes display the data for obfuscation type groupings $G1$ through $G5$, respectively. Thus, the leftmost box in each set of five represents App 1 with obfuscation type grouping $G1$, the next box to the right represents App 1 with obfuscation type grouping $G2$, and so forth. Each individual box represents the data for a given obfuscation type grouping across all sequences used for that grouping. That is, T12, T21, T31, and others are all part of $G2$, and their values on App 1 are all included in the data from which the second box
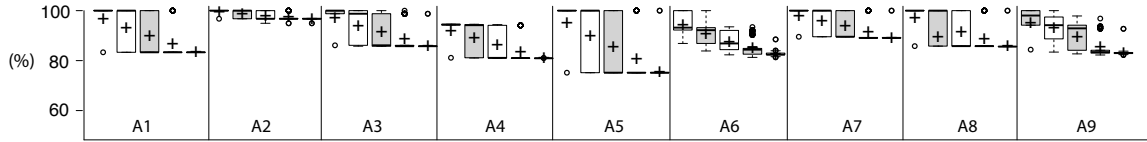
Figure 5.1: Recall for obfuscation type groupings G1-G5 on RQ1 Apps

from the left was generated). The y-axis reports recall percentages, computed using the equation provided in Section 4. The mean value within each group is denoted by a "+".

As the boxplots illustrate, for each app, as obfuscation complexity increases, mean recall also decreases. For obfuscation type grouping $G1$, in seven of nine cases there is little variance in results, and the mean recall values are between 95% and 100%, indicating that with single obfuscations applied, SEMEO is highly effective at identifying semantically equivalent methods. Even when the obfuscations applied are the most complex (obfuscation type grouping $G5$) the mean recall exceeds 80% on eight out of nine apps. The lowest mean recall (76%) occurs on App 5 for obfuscation type grouping $G5$. Larger variance in results occurs most often on obfuscation type groupings $G2$ and $G3$; in five of nine cases these are the only obfuscation type groupings that display large degrees of variance. This suggests that when only two or three types of obfuscations are applied they can interact in a wider variety of ways that impact SEMEO's performance more than when larger numbers are applied.

For function inlining and outlining (G6-G8), the recall data is presented in Table 5.1. (Boxplots are not appropriate in this case, because we do not use multiple permutations of obfuscation technique orderings in this case, and thus do not have a distribution of data points). For these obfuscation type groupings, recall ranges from 80.56% to 100%.

Table 5.1: Recall for Obfuscation Type Groupings T6, T7 and T6+T7 on RQ1 Apps

Grouping	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9	Average
G6	90.48	95.45	94.41	95.25	100	89.92	100	99.52	83.33	94.26
G7	91.3	95.52	90.96	94.41	99.61	95.77	99.86	99.9	83.72	94.56
G8	82.61	80.56	91.72	94.24	99.61	91.14	99.95	99.8	83.22	91.43

SEMEO achieved average recall values ranging from 91.43% to 94.56%. The somewhat lower recall values in this case attest to the difficulty of determining semantic equivalence in the presence of substantial changes in program structure.

5.2 RQ2

RQ2 concerns the effectiveness of SEMEO at identifying repackaged methods in obfuscated apps. We apply all seven obfuscation types listed in Table 4.2 in RQ2 as well, under the same sets of obfuscation types groupings.

Figure 5.2 reports recall values for obfuscation type groupings G1 through G5. SEMEO achieved average levels of recall ranging from 69% to 100%. For App 2, App 3, App 5, and App 6, the recall values for RQ2 are lower than those in RQ1. The reductions range from 20% for App 2 to about 10% for the other apps.

Table 5.2 displays the recall values for SEMEO for obfuscation type groupings G6-G8. Again, changes in program structure make determining semantic equivalence more challenging. As shown, the average recall value for each of the three obfuscation type groupings is reduced by 5% compared to that of the corresponding obfuscation type groupings in RQ1. The greatest decrease in recall occurs on App 2, which is a small app.

Turning to precision, SEMEO achieved 100% precision on all apps repackaged with all obfuscation types and obfuscation type groupings. This means that none of the modified methods were mistakenly identified as semantically equivalent.
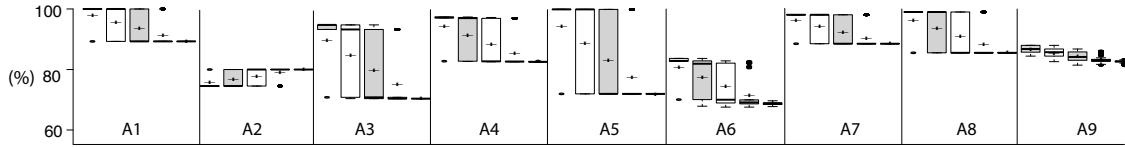
Figure 5.2: Recall for obfuscation type groupings G1-G5 on RQ2 Apps

Table 5.2: Recall for Obfuscation Type Groupings G6-G8 on RQ2 Apps

Grouping	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9	Average
G6	84.21	48.61	88.98	98.23	99.18	82.45	81.64	87.5	90.69	84.61
G7	72.73	70.49	77.24	95.7	97.08	81.46	80.84	86.98	90.69	83.69
G8	72.73	48.53	80.56	98.08	97.61	80.59	80.84	86.94	90.69	81.84

5.3   RQ3

RQ3 concerns the efficiency of SEMEO. Figure 5.3 and Figure 5.4 show mean performance values for SEMEO, gathered on runs with the apps used for RQ1 and RQ2, for the five obfuscation type groupings G1-G5. In both figures, the results for Apps 1 through 7 are similar. The results for Apps 8 and 9 show increasing analysis times.

Figure 5.3: Performance of SEMEO on obfuscation type groupings G1-G5 on RQ1 Apps (seconds)

Table 5.3 and Table 5.4 show the performance values for SEMEO for the inlining, outlining, and hybrid obfuscation type groupings.
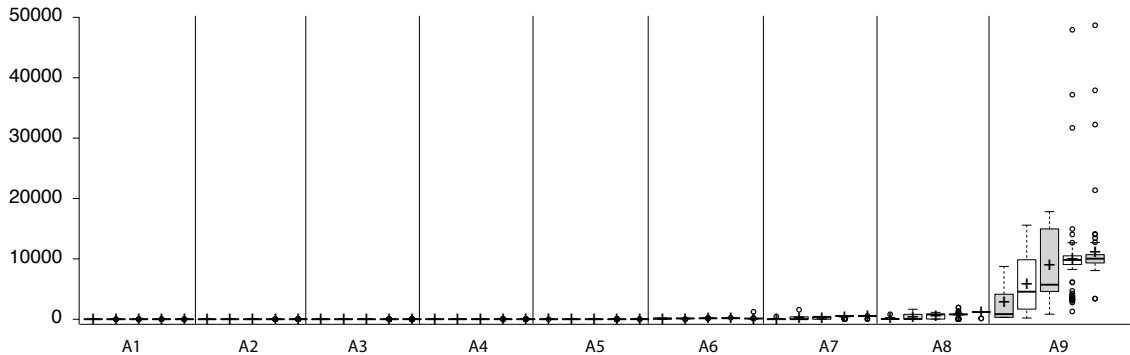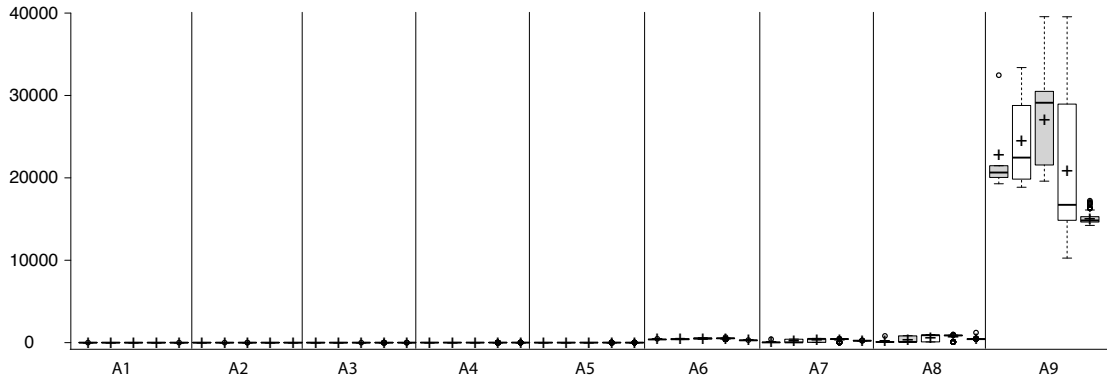
Figure 5.4: Performance of SEMEO on obfuscation type groupings G1-G5 on RQ2 Apps (seconds)

Table 5.3: Performance of SEMEO on Obfuscation Type Groupings G6-G8 on RQ1 Apps (Seconds)

Grouping	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9
G6	0.27	0.31	0.47	2.14	1.02	125	7.36	26.95	6808.98
G7	0.29	0.29	0.46	2.33	1.07	55.88	9.24	27.24	7113.36
G8	0.57	0.4	0.58	2.47	1.13	117.99	8.09	21.21	6056

Table 5.4: Performance of SEMEO on Obfuscation Type Groupings G6-G8 on RQ2 Apps (Seconds)

Grouping	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9
G6	0.31	0.57	1.05	3.41	2.71	236.91	403.91	813.11	6570.87
G7	0.38	0.45	0.9	12.15	3.62	246.06	419.63	855.74	6738.51
G8	0.34	0.62	1.29	2.76	3.84	259.06	414.03	818.56	6506.24

As the data shows, SEMEO can be quite efficient when the number of methods it needs to analyze is of small to moderate size. Only as the number of methods in an app neared 3000 (App 8) did execution time from 34 seconds to 1267 seconds (21 minutes). In the case of App 9, which has over 7000 methods, the analysis time ranged from 203.46 seconds to 12693.6 seconds (2.7 hours). Note that SEMEO is much faster when the pair of original and obfuscated apps to be analyzed are semantically the same. As shown in Table 5.1, App 7 and App 8 have nearly 100% recall for RQ1.

In this case, the analysis times are also quite fast (less than 30 seconds as shown in Table 5.3) in spite of the presence of over 2000 methods. However, for RQ2 as shown in Table 5.2, the recall values for App 7 and App 8 decrease to 80% and 86%, respectively, due to the presence of modified methods. In this case, the analysis time also increases to 400 seconds and 850 seconds, respectively (see Table 5.4). The performance overhead in these cases arises primarily due to the complexity of Algorithm 1 ($O(n^2)$).

For RQ2, we report performance results for SEMEO on obfuscation type groupings G1-G5 in Figure 5.4. Table 5.4 shows mean performance values for SEMEO for the inlining and outlining obfuscation type groupings of RQ2. Similar to the case with RQ1, larger apps experience higher runtime overheads. However, the amount of time required to perform equivalence analysis on a repackaged app is generally the same as the time required to analyze the same apps for RQ1.

Chapter 6

Discussion

Based on our results, some methods in obfuscated apps are incorrectly deemed by SEMEO to be non-equivalent to those in the original apps. There are various reasons behind such misidentifications. First, *SEMEO* analyzes obfuscation grouping types $G1$ and $G2$ very accurately because these are not as complex as $G3$, $G4$, and $G5$. The majority of lower recall values occur on groupings $G4$ and $G5$, where four and five layers of obfuscation have been applied. Such high degrees of composite obfuscation makes analyzing apps for semantic equivalence more challenging.

Our analysis involves register comparisons when comparing the DEX instructions of two methods. However, when the method inlining and outlining techniques are used, we require interprocedural analysis. Additional methods can cause the register assignments performed by the compiler to change. In this case, SEMEO would report these methods as not semantically equivalent, which means that the apps need to be further analyzed to verify whether they are indeed semantically different. As such, our design errs toward being more conservative; however, the recall numbers still show that the number of methods that must be analyzed represents only a small fraction of the number of total methods for each app. For example, our largest app

has 11,174 methods after obfuscation. Our analysis leaves only 15% of these methods for analysis.

While SEMEO does mistakenly identify some semantically equivalent methods as non-equivalent, it did not, in the cases we considered, mistakenly identify modified methods as semantically equivalent. This means that in these cases, the approach would not allow modified methods to escape analysis. We also discovered that when we introduce modified methods that change application semantics, our recall degrades slightly. This is because these modified methods can affect some superclass and subclass relationships. Some modifications to global variables in the modified methods can also affect other methods that share these global variables. These scenarios can cause SEMEO to identify some equivalent methods as non-equivalent. Still, the approach filters outs a large portion of equivalent methods, leaving only a small percentage of methods to be analyzed. The amount of additional overhead spent on dealing with these complexities is also very small as the reported time for RQ2 is about the same as the reported time for RQ1, for the same app.

To scale SEMEO to handle larger apps, significant reengineering efforts will be required. For example, additional heuristics might be able to reduce the complexity of Algorithm 1. One option is to first sort all the methods based on size to allow searches to be more localized based on size. Other heuristics that may facilitate faster searching would be to partition methods based on method signatures, input/output flows, and the types of calls they make to libraries.

Obfuscation techniques that try to alter library calls can be complex and would likely affect the semantics of apps. As such, they are not likely to be used so relying on them as a way to help partition similar methods to allow searching to be applied to smaller data sets or support searching in parallel can reduce the cost of our approach. Because SEMEO is based on graphs that can be processed by the *Boost*

Graph Library, we also plan to explore existing graph algorithms such as isomorphism testing [36] that may help reduce the analysis overhead.

Chapter 7

Additional Case Studies

In this chapter, we report the results of three additional studies designed to evaluate the accuracy and performance of SEMEO under realistic settings. In the first study, we compare the accuracy of SEMEO with that of an alternative approach for detecting repackaged apps. In the second study, we apply SEMEO to detect semantically equivalent methods when PROGUARD, a commonly used commercial obfuscation tool, is used in addition to our adopted obfuscation methods. In the third study, we use SEMEO to detect modified methods in a complex, real-world repackaged malware sample.

7.1 Comparing SEMEO to an Existing Alternative Technique

We now consider other techniques that can detect repackaged Android applications in the presence of code obfuscation. We examined VIEWDROID [39] and FSQUADRA [40]. VIEWDROID uses UI-based birthmarks to help detect differences in UI connections between an original app and a repackaged app. Unfortunately, VIEWDROID is not pub-

licly available. FSQUADRA uses resource-usage-based birthmarks to detect repack-
aged apps. Specifically, it looks for identical resources that are present in the orig-
inal app and the potentially repackaged app. According to reported results [40],
FSQUADRA is efficient and accurate in detecting repackaged apps. FSQUADRA is
also publicly available. Thus, we compare the performance of SEMEO to that of
FSQUADRA.

Tables 7.1 and 7.2 provide data on the use of FSQUADRA to identify semantically
equivalent methods in the apps used in our initial study for RQ1 and RQ2, respec-
tively. Each app was individually obfuscated with obfuscation grouping types G1, G6,
and G7 (hence, we applied all seven obfuscation types, T1 to T7). Our goal was to
observe whether the underlying analysis approaches used by these two approaches are
sensitive to these obfuscation types. We compared the recall performance of SEMEO
with that of FSQUADRA for these obfuscation groupings.

Recall that for RQ1 our apps were obfuscated but not modified. As such, the
number of semantically equivalent methods should be 100%. As Table 7.1 shows,
however, the average recall performances of FSQUADRA on these apps ranged from
19.05% to 43.48% when obfuscation type grouping G1 was applied. SEMEO, in
contrast was able to achieve recall performances across all nine apps ranging from
95% to 100% in this case (as shown by the boxes representing G1 in Figure 5.1).
When inlining (grouping G6) and outlining (grouping G7) were applied, FSQUADRA
achieved higher recall values than on obfuscation type grouping G1, ranging from
38.12% to 95.74% with an average of 76.09%. SEMEO, on the other hand, was able
to achieve recall performances ranging from 80.56% to 100% with an average above
91.43% in this case(see Table 5.1).

For the set of apps that were modified and then obfuscated (used to answer RQ2),
FSQUADRA was able to achieve recall performances ranging from 19.04% to 42.86%

Table 7.1: Recall of FSQuADRA Using RQ1 Apps

Grouping	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9	Average
G1	37.5	20	42.86	43.48	19.05	37.78	25	23.08	43.04	32.42
G6	69.23	63.6	95.74	77.78	92.31	93.75	76.47	77.78	38.12	76.09
G7	69.23	63.6	95.74	77.78	92.31	93.75	76.47	77.78	38.12	76.09

Table 7.2: Recall of FSQuADRA Using RQ2 Apps

Grouping	App 1	App 2	App 3	App 4	App 5	App 6	App 7	App 8	App 9	Average
G1	37.5	20	42.86	42.85	20	19.04	25	23.08	42.32	30.29
G6	69.23	63.63	95.74	95.74	92.31	93.75	76.47	77.78	21.8	76.27
G7	69.23	63.63	95.74	95.74	92.31	93.75	76.47	77.78	21.8	76.27

when obfuscation type grouping G1 was applied (see Table 7.2). SEMEO, in contrast, achieved recall performances ranging ranging from 75% to 100% (boxes representing G1 in Figure 5.2). For obfuscation type groupings G6 and G7, FSQuADRA's recall ranged from 21.8% to 95.74% with average of 76.27%. SEMEO, on the other hand, achieved recall performances ranging from 48.53% to 99.18% with an average above 81.84%(see Table 5.2).

The results we present for FSQuADRA do not include those for composite obfuscation (obfuscation type groupings G2-G5). This is because when we applied these groupings, we found that the recall performance of FSQuADRA was not sensitive to the differences in groupings. We believe that the approaches used to create composite obfuscation groupings in our study do not change resource usage, which is the main analysis criterion utilized by FSQuADRA. In contrast, SEMEO's recall performance *is* sensitive to these different approaches: under all approaches, SEMEO is significantly more accurate than FSQuADRA at detecting semantically equivalent methods.

7.2 Applying SEMEO on Apps Obfuscated by ProGuard

We next sought to assess whether SEMEO can identify semantically equivalent methods in apps that have been obfuscated by PROGUARD, an obfuscation tool for Android. PROGUARD is commonly used to protect intellectual property. Obfuscation through PROGUARD is applicable within the Android Studio IDE.

To perform our evaluation, we chose apps that we can successfully apply PRO-GUARD to; these include six of the apps used in our initial study. In this case, we utilized apps that contain modifications (the apps used to answer RQ2). We first applied PROGUARD to the source code. Then, we applied obfuscation grouping G1 (individual obfuscation types) and obfuscation grouping G5 (composite obfuscation types). Table 7.3 lists the numbers of modified methods used, and recall values involved, when SEMEO and FSQUADRA were used in these cases.

Note that once we applied PROGUARD to an app, the structure of the app can change. For example, PROGUARD removes dead code and performs some optimization including inlining. As such, we had to reapply ALAN after an app that has been obfuscated by PROGUARD. This caused the number of modified methods for each app to be different than those listed in Table 4.1 due to changes to the program structure.

As Table 7.3 shows, SEMEO was more effective than FSQUADRA at detecting semantically equivalent methods when two or more layers of obfuscations (PROGUARD followed by our own additional obfuscation types) were applied. SEMEO achieved recall values ranging from 52.02% to 94.87% when two layers of obfuscations were applied (e.g., PROGUARD and an obfuscation type in $G1$), with the average recall value of 79.04%. When six layers of obfuscations were applied (i.e., PROGUARD and

Table 7.3: Recall Results (%) Achieved by SEMEO and FSquaDRA When Applying ProGuard on Obfuscation Type Grouping G1 and G5

Apps	Modified Methods	G1		G5	
		Semeo	FSquaDRA	Semeo	FSquaDRA
App 1	3	73.34	37.50	55.56	37.50
App 2	8	52.02	20.00	51.40	20.00
App 3	10	68.47	42.86	64.74	42.86
App 6	10	94.87	37.78	91.31	37.78
App 7	11	94.36	25.00	84.11	25.00
App 8	10	91.18	23.08	91.05	23.08
Average		79.04	31.04	73.03	31.04

$G5$), SEMEO achieved recall values ranging from 51.40% to 91.31% with an average recall value of 73.03%. FSquaDRA, on the other hand, achieved recall values ranging from 20% to 42.86% with an average of 31.04%. Further, as noted in Section 7.1, FSquaDRA was not sensitive to differences in obfuscation type groupings.

Table 7.4 reports the analysis times observed in this study. FSquaDRA was consistently much faster than SEMEO. In the case of App 8, when G1 was applied, FSquaDRA was more than 200 times faster than SEMEO. This is because FSquaDRA's resource usage analysis is much faster than our analysis, which must analyze code-level birthmarks that have been obfuscated. Despite being much slower, however, SEMEO produced much more precise analysis results in an amount of time that is not unreasonable: the longest analysis time we observed was just over 30 seconds.

Table 7.4: Analysis Times (Seconds) Required by SEMEO and FSquaDRA When Applying ProGuard on Obfuscation Type Grouping G1 and G5

Apps	G1		G5	
	Semeo	FSquaDRA	Semeo	FSquaDRA
App 1	1.92	0.10	0.70	0.11
App 2	6.56	0.10	2.74	0.15
App 3	8.54	0.15	4.07	0.19
App 6	19.98	0.13	21.94	0.11
App 7	5.80	0.13	3.91	0.11
App 8	33.00	0.16	18.86	0.11

7.3   Identifying Semantically Equivalent Methods in Real-World Repackaged Malware

Within a week of official release of *Pokémon Go* in the US, several repackaged malicious versions of the app were distributed through the third party stores. In countries that have no access to Google Play, third party stores are the main distribution channels by which Android users obtain apps. Furthermore, popular games such as *Pokémon Go* had different release dates in different countries, so many users who could not wait for the official release date in their countries downloaded the app from third party stores. Many of these stores, however, are less rigorous than Google Play when it comes to vetting submitted apps for security vulnerabilities. Moreover, three instances of repackaged *Pokémon Go* malware have been made available for download from Google Play [9, 12].

Typically, popular apps attract the attention of cyber-criminals because they are highly downloaded, and thus, repackaged versions unknowingly downloaded by users can infect a large number of devices quickly. For example, one version of *Pokémon Go* was repackaged with a *Remote Access Tool* or *RAT*. For this particular app, the malware author downloaded the legitimate version that had been obfuscated using PROGUARD. The malware author modified the app to contain a RAT tool called DROIDJACK, which allows cyber criminals to remotely take control of infected devices [20, 31]. The presence of this malicious version of the app was first detected three days after its official release.

In this case study, we investigated the effectiveness of SEMEO at identifying the repackaged components in this repackaged release of *Pokémon Go*. The legitimate version contains 37,024 methods while the repackaged version contains 38,878 methods. We used information from a security analysis result [20] to identify the methods

that had been modified or added. In total, there were 1,854 such methods. We then used SEMEO to analyze both versions of the app. SEMEO found 95.23% of the methods in the two versions of the app to be equivalent. The remaining 4.77% of methods are in fact all the modified methods that have been previously reported [20]. The analysis time was approximately 2,300 seconds. We also used FSquaDRA to analyze these two versions of *Pokémon Go*. FSquaDRA found 89.47% of the methods to be similar. This lower accuracy translates to 2,240 additional methods that a security analyst needs to analyze because they are reported as not equivalent.

Chapter 8

Related work

We have already discussed VIEWDROID [39] and FSQUADRA [40], and presented results comparing SEMEO to the latter of these. These two techniques are the most directly comparable to SEMEO.

There are other de-obfuscation tools that can be used to indirectly help with the task SEMEO performs: a de-obfuscator can be applied, and then de-obfuscated modules can be differenced against original unobfuscated modules. We did attempt to apply this approach using several such tools. One tool, dex-oracle [4], looks for specific patterns and cannot deobfuscate our programs. We also considered ANDROSIM, which is a commonly used Android reverse engineering tool in the Androguard toolset [15]. ANDROSIM identifies similarities between two applications. However, when we applied ANDROSIM to original and obfuscated applications it misclassified many obfuscated methods as not equivalent to their original methods. This occurred because Androsim can handle only simple obfuscation types. We also attempted to use *Simplify* [5] but it also failed to deobfuscate most methods.

In addition to the tools just discussed, there is some other work on detecting obfuscation types. Myles et al. [28] analyze binary code to look for similarity based on K-

gram. However, their approach cannot handle two types of obfuscation, namely, junk code insertion and code reordering. SAFE, which is a malware detection algorithm, can handle simple obfuscations, like inserting NOP instructions [7]. Kruegel [24] uses static analysis on binaries to detect kernel-level rootkits. Apposcopy [16] is a semantics-based analysis tool to detect Android malware based on signatures. Dexteroid [21] is a tool that detects behavior-based malware according to the Android life cycle model. None of these tools can analyze obfuscated code to look for semantic equivalence.

There are also existing Android clone detection tools such as AnDarwin [11] and DNAdroid [10]. The purpose of these clone detection tools is to detect clone apps. As such, they are capable of working with some forms of obsfucation techniques. However, they are not publicly available so we could not use them in our studies. Symbolic semantic analysis tools can also be used to determine the semantic equivalence of two applications. However, they do not scale well for applications in large applications [29, 30, 33].

Chapter 9

Conclusions and Future work

We have presented a technique for directly identifying (without first deobfuscating) obfuscated methods in Android apps that are semantically equivalent to original non-obfuscated methods. Our empirical results show that our approach, SEMEO, can achieve a high level of recall at no loss of precision in identifying such methods. SEMEO operates on types of obfuscation many of which are difficult to automatically deobfuscate. Our approach is also reasonably efficient on apps consisting of no more than 2200 methods; as such the current approach is sufficiently efficient to apply to a large percentage of existing Android apps.

We also compared the accuracy of SEMEO with that of FSQUADRA, an alternative approach for obfuscation resilient repackaged app detection. The results indicate that SEMEO is more accurate at identifying methods that have been modified. We also evaluated the capability of SEMEO to deal with PROGUARD and find that it can effectively handle obfuscation types that PROGUARD utilizes. Lastly, we used SEMEO and FSQUADRA to identify repackaged components of a version of *Pokémon Go* malware. The results indicate that SEMEO can detect all modified malicious methods while FSQUADRA mistakenly identifies over 2000 benign methods as malicious.

The potential benefit of our approach involves its ability to reduce the number of methods that analysts or analysis tools must consider when searching for malicious repackaged code, allowing them to apply their efforts more cost-effectively than would otherwise be possible.

In future work we intend to explore methods for improving the scalability of our approach, several approaches for which were discussed in Chapter 6. We will also consider methods for improving the approach's recall, particularly in cases where more complex composite obfuscations are used. Finally, we intend to conduct additional studies of the approach, including studies applying it to more repackaged malicious apps.

Bibliography

[1] *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Dalvik bytecode. On-line article, 2015. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[3] Andrey Ponomarenko. A tool for checking backward API/ABI compatibility of a Java library. On-line article, 2013. https://github.com/lvc/japi-compliance-checker.

[4] CalebFenton. A pattern based Dalvik deobfuscator which uses limited execution to improve semantic analysis. On-line article, 2015. https://github.com/CalebFenton/dex-oracle.

[5] CalebFenton. Generic Android Deobfuscator. On-line article, 2015. https://github.com/CalebFenton/simplify.

[6] Joshua Cannell. Obfuscation: Malware's best friend. On-line article, March 2013. https://blog.malwarebytes.org/threat-analysis/2013/03/obfuscation-malwares-best-friend/.

[7] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Secu-*

rity Symposium - Volume 12, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[8] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[9] contagio mini dump. Pokemon GO with Droidjack - Android sample. On-line article, 2016. http://contagiominidump.blogspot.com.

[10] Jonathan Crussell, Clint Gibler, and Hao Chen. *European Symposium on Research in Computer Security*, chapter Attack of the Clones: Detecting Cloned Applications on Android Markets, pages 37–54. Springer, Berlin, Heidelberg, 2012.

[11] Jonathan Crussell, Clint Gibler, and Hao Chen. *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, chapter AnDarwin: Scalable Detection of Semantically Similar Android Applications, pages 182–199. Springer, Berlin, Heidelberg, 2013.

[12] Dan Goodin. Fake Pokémon Go app on Google Play infects phones with screen-locker. On-line article, 2016. http://arstechnica.com/security/2016/07/fake-pokemon-go-app-on-google-play-infects-phones-with-screenlocker/.

[13] DARPA. Automated Program Analysis for Cybersecurity (APAC). On-line article, 2012. http://www.darpa.mil/program/automated-program-analysis-for-cybersecurity.

[14] G Data. At a glance. In *G Data Mobile Malware Report*, 2015.

[15] Anthony Desnos. AndroGuard. On-line article, May 2013. http://androguard.blogspot.com/ .

[16] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[17] Guard Square. ProGuard. On-line article, 2016. http://proguard.sourceforge.net/.

[18] Inc. IDC Research. Smartphone OS Market Share, 2015 Q2, 2015. https://mobiforge.com/news-comment/mobile-os-market-share-q2-2015/.

[19] Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge, MA, USA, 1997.

[20] Joseph Sullivan. Pokmon Go bundles with Malicious Remote Administration Tool DroidJack. On-line article, 2016. http://blog.trustlook.com/2016/09/02/pokemon-go-bundles-with-malicious-remote-administration-tool-droidjack/.

[21] Mohsin Junaid, Donggang Liu, and David Chenho Kung. Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *CoRR*, abs/1506.05217, 2015.

[22] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 155–169, 2000.

[23] Evgenios Konstantinou. Metamorphic Virus: Analysis and Detection. Technical Report RHUL-MA-2008-02, Royal Holloway, University of London, Janary 2008.

[24] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[26] Linux Foundation. JavaAPI Compliance Checker, 2015. http://ispras.linuxbase.org/index.php/ Java_API_Compliance_Checker.

[27] Mr.Trojans. ALAN – Android Malware Evaluating Tools Released. On-line article, 2015. http://seclist.us/alan-android-malware-evaluating-tools-released.html.

[28] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 314–318, Santa Fe, New Mexico, 2005.

[29] Nimrod Partush and Eran Yahav. Abstract semantic differencing via speculative correlation. *SIGPLAN Not.*, 49(10):811–828, October 2014.

[30] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[31] Proofpoint Staff. DroidJack Uses Side-LoadIt's Super Effective! Back-doored Pokemon GO Android App Found. On-line article, 2016. https://www.proofpoint.com/us/threat-insight/post/droidjack-uses-side-load-backdoored-pokemon-go-android-app.

[32] Babak Bashari Rad and Maslin Masrom. Metamorphic virus variants classification using opcode frequency histogram. In *Proceedings of the International Conference on Computers*, pages 147–155, 2010.

[33] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[34] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 329–334, 2013.

[35] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 274–283, Atlanta, Georgia, USA, 2007.

[36] Jeremy G. Siek. An implementation of graph isomorphism testing, 2001. http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/isomorphism-impl.pdf.

[37] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. JVM specifications. On-line article, 2013. https://docs.oracle.com/javase/specs/jvms/se7/html/.

[38] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[39] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. View-droid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, pages 25–36, Oxford, United Kingdom, 2014.

[40] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: Fast detection of repackaged applications. In *IFIP 28th Annual Data and Applications Security and Privacy*, pages 130–145, Vienna, Austria, July 2014.

[41] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, San Antonio, Texas, USA, 2012.

[42] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 95–109, San Francisco, CA, USA, May 2012.