SEMEO: A Semantic Equivalence Analysis Framework for Obfuscated Android Applications

Zhen Hu[1], Bruno Vieira Resende E. Silva[1], Hamid Bagheri[1], Witawas Srisa-an[1(✉)], Gregg Rothermel[2], and Jackson Dinh[1]

[1] School of Computing, University of Nebraska-Lincoln, Lincoln, NE 68588, USA
{bagheri,witawas}@unl.edu
[2] Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
gerother@ncsu.edu

Abstract. Software repackaging is a common approach for creating malware. Malware authors often use software repackaging to obfuscate code containing malicious payloads. This forces analysts to spend a large amount of time 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 analysts must consider could be reduced, allowing them to be more productive. In this paper, we present SEMEO, an obfuscation-resilient approach for semantic equivalence analysis of Android apps. SEMEO automatically and with high accuracy determines whether a repackaged and obfuscated version of a method is semantically equivalent to an original version thereof. SEMEO further handles widely-used and complicated types of obfuscations, as well as the scenarios where multiple obfuscation types are applied in tandem. Our empirical evaluation corroborates that SEMEO significantly outperforms the state-of-the-art, achieving 100% precision in identifying semantically equivalent methods across almost all apps under analysis. SEMEO consistently provides over 80% recall when one or two types of obfuscation are used and 73% recall when five different types of obfuscation are compositely applied.

Keywords: Malware · Android · Security

1 Introduction

Software repackaging is the leading approach employed to create Android malware [17], where cybercriminals modify a legitimate app by adding code that performs malicious behavior. To render malicious components more difficult to spot, malware developers typically obfuscate the entire codebase of a repackaged app [20]. From the malware author's perspective, they use obfuscation to

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2022
Published by Springer Nature Switzerland AG 2022. All Rights Reserved
T. Hara and H. Yamaguchi (Eds.): MobiQuitous 2021, LNICST 419, pp. 322–346, 2022.
https://doi.org/10.1007/978-3-030-94822-1_18

permanently change the structure of a malicious app to avoid detection. However, these changes *must not be too drastic* such that they change the behaviors of the app. As such, malware authors utilize obfuscation techniques that maintain the semantic equivalence of the two programs. Because these two versions are semantically the same, the obfuscated version contains statically detectable semantic traces. Rastogi et al. refer to this type of obfuscation as *Detectable by Static Analysis* or DSA, and identify five classes of DSA obfuscation that can change the structure while preserving the original behaviors [37].

To further hide malevolent behavior, malware authors often apply these DSA obfuscation techniques in tandem. A study shows that only two layers of obfuscation are needed to defeat sophisticated, commercial malware scanners [37]. A recent study by Li et al. indicates that despite significant progress in the state-of-the-art in dealing with obfuscation, they still lose their effectiveness by as much as 75% when facing malware obfuscated with five layers [25].

From the security analyst's perspective, it is essential to analyze each detected malware to determine whether it is a true- or false-positive. If it is indeed malware, the analysts need to perform forensic to characterize attack patterns, attack surfaces, and threat models. To do so, they need to identify which components have been repackaged and further analyze the malicious code to develop knowledge-base, countermeasures, and detectors. Identifying which modified components to analyze is a challenging task, especially when malware authors obfuscate every component in the app [33].

Unfortunately, existing techniques to identify tampered components are not effective. These approaches often apply deobfuscation mechanisms to the obfuscated segments, followed by applying "differencing" techniques to compare the deobfuscated versions with the originals (e.g., [1,3,4,13,15,26]). Thus, increasingly sophisticated obfuscation types can hobble deobfuscators (Sect. 2 discusses this further). Even when deobfuscators can be applied, they do not necessarily retrieve code matching the original code; instead, they focus on re-engineering code into a format digestible by engineers. Thus, the differencing process may not produce accurate results.

This paper contributes a novel approach, called SEMEO, for automatically analyzing **sem**antic **e**quivalence of **o**bfuscated Android applications. Unlike all prior techniques that first require to apply deobfuscation mechanisms to the obfuscated segments, SEMEO has the potential to significantly expand the scope of semantic equivalence analysis by directly analyzing obfuscated methods in Android applications, without additional deobfuscation. This, in turn, enables reasoning about a broad range of sophisticated and widely-used obfuscation techniques, including those currently employed in practice by authors of Android malware in particular [20,29,32].

SEMEO first extracts an instruction summary of each method in the two apps and identify semantic traces, which are vital program traits that obfuscation cannot entirely mask. For example, because malware authors cannot apply obfuscation to Android API calls, one semantic trace is dataflow information into these calls. Using dataflow analysis, we can reconstruct such flows and compare them

to similar flows in the unobfuscated app. Furthermore, SEMEO can identify data sources in the original app and look for similar sources in the unobfuscated app. The other vital semantic trace is information flows from these sources to data sinks. While the flows between the two apps may change, the source-sink information should generally remain the same. Once SEMEO automatically derives this information from each app, it conducts semantic equivalence analysis. For the forensic purpose, the analysis would identify and eliminate code portions in the obfuscated app that are semantically equivalent to those in the unmolested app. Scanty non-equivalent methods that remain can then be further analyzed to pinpoint possible malicious components, relieving the analysts of the error-prone and time-consuming task of scrutinizing the entire codebase.

We present the results of an empirical study assessing the efficiency and effectiveness of SEMEO, in which we applied it to 14 real-world Android apps of varying complexity. We also compare SEMEO's performance to that of FSQUADRA [42], a state-of-the-art approach for identifying repackaged apps. Our evaluation shows that SEMEO can achieve between 73% (when five obfuscation types are compositely applied) and 100% (when one or two types are applied) recall in identifying obfuscated methods. Given the worst-case recall of 73%, analysts need only consider the remaining 27% of the methods. SEMEO also achieved 100% precision (i.e., there was no misidentification of non-equivalent methods as equivalent) in 13 out of 14 cases. FSQUADRA, on the other hand, achieves between 20% and 99% recall, but it does not provide any information regarding its precision.

We also present the results of two additional case studies. In the first case study, we used ProGuard to apply a second layer of obfuscation to a subset of apps studied in our initial evaluation. Our results show that even under these conditions, SEMEO remains highly effective, while continuing to outperform FSQUADRA. In the second case study, we applied SEMEO to a repackaged, malicious version of *Pokémon Go*, and found SEMEO to be effective.

2 Background on Obfuscation Methods

Rastogi et al. [37] classify three categories of common obfuscation techniques: (1) trivial obfuscations, which can be easily detected by most antivirus tools; (2) DSA obfuscations, which theoretically can be detected by static analysis techniques; and (3) NSA obfuscations, which often apply encryption to the code. NSA obfuscations are, however, not permanent, as it requires the encrypted app to be decrypted prior to execution on a typical Android VM. A study has shown that the decrypted version can be copied from a device's memory just prior to execution [14].

This paper focuses on DSA obfuscations due to their ability to evade detection and preserve semantics. We focus further on five commonly found classes of DSA obfuscations [6,16,29,32]: junk code insertion, code reordering, method indirection, function inlining and function outlining [7,22,23,35,37,41].

Junk code insertion involves inserting unnecessary code such as a NOP operation (referred to hereafter as obfuscation type **T1**), a branch with predicates to ensure the branch can never be taken **(T2)**, and dead code instructions **(T3)** into an app. The additional code does not affect program behavior [23,35,41].

Code reordering (T4) involves changing the execution order of statements or blocks of code. This obfuscation type can be difficult to detect and remove, and can render it difficult to determine whether obfuscated code is semantically equivalent to the original code. A reordered version, for example, can reverse the order of conditional tests.

Method indirection (T5) inserts additional calls into an app to manipulate call graphs. With this approach, a given method call (e.g., $m_0 \rightarrow m_1$) is 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 (T6) replaces method calls with the actual bodies of called methods. Normally used by compilers for optimization, this obfuscation type breaks abstraction boundaries created by the programmer [7].

Function outlining (T7) 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.

3 SEMEO

We now present our approach for Semantic Equivalence Analysis of Obfuscated Code (SEMEO). Its key objective 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 should be semantically equivalent to the original unmodified methods. Identifying a large percentage[1] of these semantically equivalent methods, enables security analysts to focus on the other methods that cannot be conclusively identified as semantically equivalent.

Figure 1 illustrates the workflow of SEMEO. It takes as input a pair of apps in the APK form: an app P and an obfuscated version P' of P that is suspected to have been repackaged. In *Step I*, SEMEO analyzes both P and P' to generate the necessary information including method call and instruction graphs. Once this information has been generated, SEMEO, in *Step II*, analyzes these graphs to produce interprocedural and intraprocedural control-flow and data-flow information. It then uses the information to perform semantic equivalence analysis.

[1] In general, the problem of determining the semantic equivalence of two programs is undecidable [18], so our approach is necessarily a heuristic.

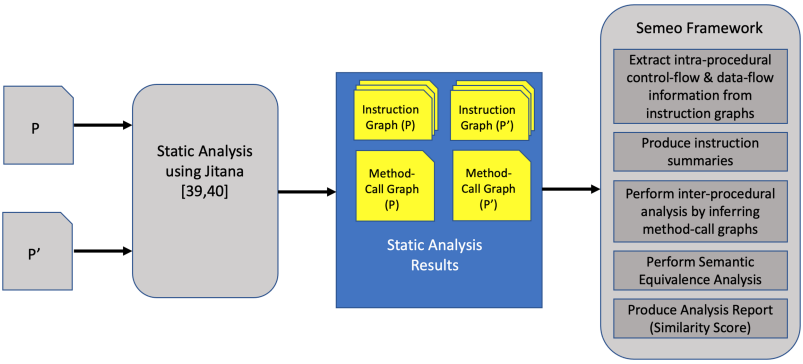

Fig. 1. Steps to perform semantic equivalence analysis in SEMEO

It then reports similarity score which reflects the percentage of methods determined to be semantically equivalent between P and P'.

Note that state-of-the-art approaches including REPDETECTOR (not publicly available) and FSQUADRA (used as our baseline system) also use pair-wise comparison [16,42]. Also note that app P can be previously obfuscated, and P' has one or more layers of obfuscation on top of P. SEMEO compares methods in P' (m'_j) to methods in P (m_i). The mapping of methods in P' to methods in P, however, may not be one-to-one as obfuscation techniques can merge methods, extract new ones, or obscure the mapping between obfuscated and original methods. SEMEO accounts for this.

For example, suppose that P contains three methods, (m_0, m_1, and m_2) and that P', an obfuscated, repackaged version of P, contains two methods, (m'_0 and m'_1), where m'_1 is the result of inlining 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. SEMEO uses the method-call graph of each app to help identify potential candidates based on the calling patterns. In case the methods m_0 and m'_0 *are* found to be semantically equivalent, they are marked as such. SEMEO does not need to visit m_0 again. It then compares m_1 with m'_1. In this case, the two modules are not semantically equivalent; m_1 calls m_2, so SEMEO considers both methods ($m_1 + m_2$) and evaluates whether the combined result is semantically equivalent to m'_1. Similarly, if outlining is used to split m_1 in P into m'_1 and m'_2 in P', they may together be semantically equivalent to m_1 in P. When SEMEO completes its analysis, it outputs a list of methods that have been determined to be semantically equivalent and not equivalent.

In the rest of this section, we describe each step of the process in turn.

3.1 Step I: Generating Basic Information

SEMEO compares a pair of apps at the method level. To support the necessary analysis, it first constructs the necessary analysis infrastructures such as instruction and method-call graphs. The static analysis step uses JITANA, a static

Table 1. Mutation 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

analysis framework for Android apps [39,40]. This section describes the analysis process, with an emphasis on the important improvements on prior work.

SEMEO operates on Dex instructions by analyzing the instruction graph of each method. An instruction graph is a list of Dex instructions in a method. Thus, there is the same number of instruction graphs as that of the methods. In each instruction graph, JITANA embeds intra-procedural control-flow and data-flow information. Figure 2 illustrates an instruction graph of a method generated by JITANA. The intraprocedural control-flow graph appears as blue edges; and the intraprocedural data-flow graph appears as red edges. By traversing the instruction graphs, we can identify Dex instructions that can potentially change the structure of an app that uses them but preserve semantics. Table 1 shows categories and examples of such Dex instructions derived from the Android Specification [2].

As an example, applying junk code insertion may involve injecting a few `read` instructions such as `iget` and write instructions such as `iput` to perform field accesses and store the retrieved values into temporary registers (`move` can also be used in place of read/write). These operations, in effect, add more edges to the data-flow without changing the behavior. Branching type and return instructions can also be used for outlining. We refer to such instructions that can change the number of operations without changing the semantic of the program as "mutation instructions", forming the targets of our analysis. Other instruction categories, shown in bold in Table 1, can also change the structures of the app while preserving its semantic.

3.2 Step II: Performing Semantic Equivalence Analysis

In the first step, SEMEO selects the instruction graph (shown in Fig. 2) of each candidate method in the two apps. It then extracts the intra-procedural control-flow

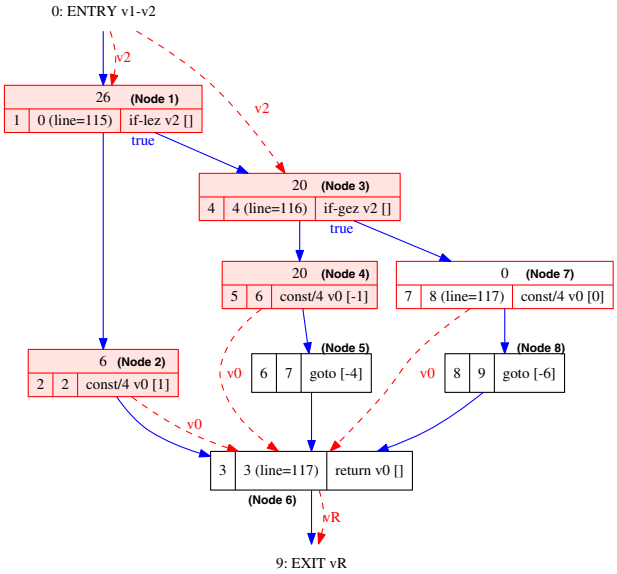

Fig. 2. Illustration of an instruction graph. (Color figure online)

(blue edges) and data-flow (red dotted edges) graphs from the instruction graph. It also infers interprocedural control-flow information from the method-call graph of each app. Algorithm 1 shows the procedure that is applied to a pair of methods m_0 and m_0'. In Line 2, the algorithm determines whether a method semantically equivalent to m_0 has been previously found. It does this using a bit map to represent the status for each method. If the mapping for m_0 is set, a semantically equivalent method in P' has already been found. The analysis continues if it is not set yet.

The algorithm uses the data-flow graph from the instruction graph for m_0 (Line 3). Each instruction consists of an opcode, registers and a constant value. Each node in the data-flow graph that is not the entry point (start of the program) has an incoming edge with registers based on the data-flow information. For example, *Nodes 1 and 3* have a shared incoming value from register *V2*.

Next, procedure `ExtractInstructionSummary` (called in Lines 4) traverses the extracted data-flow graph of each method using the DFS visitor in the *Boost graph library*, a graph processing library available on most computing platforms [38]. SEMEO traverses the data-flow graph one path at a time to check for mutation instructions. Each path summary includes instructions along these data-flow edges. The analysis constructs path summaries as an intermediate data structure so that it can inspect each path and filter out mutation instructions. When SEMEO visits the last node on each path, it combines the current path summary with the instruction summary, which is a collection of path summaries for a method. It then visits a new path and constructs a new path summary. For Fig. 2, the instruction summary of the method would have the following path summaries (PS): PS0 [Node 1], PS1 [Nodes 2 and 6], PS2 [Node 3], PS3 [Nodes 4 and 6], and PS4 [Nodes 7 and 6].

Algorithm 1. Equivalence Analysis

```
1: procedure CheckEquivalence:(Mi, Mj')
2:     if (CheckMapBit(Mi) == false) then
3:         Gi ← DataFlow(Mi)
4:         Sum1 ← ExtractInstructionSummary(Gi)
5:         if (CheckMapBit(Mj') == false) then
6:             Gj ← DataFlow(Mj')
7:             Sum2 ← ExtractInstructionSummary(Gj)
8:             if SummariesMatch(Sum1, Sum2) then
9:                 SetMappingFlag(Mi, Mj')
10:            end if
11:        end if
12:    end if
13: end procedure
```

In Line 5, the algorithm checks whether m'_0 has already been determined to be semantically equivalent to an original method. Otherwise, the algorithm extracts its data-flow graph and produces 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 (m_0, m'_0).

In our example, method inlining is used to combine two original methods (m_1, m_2) in app P into one obfuscated method (m'_1) in app P'. SEMEO first uses the method call graphs of P and P' to identify potential methods in P' that may be similar to m_1. In this example, it identifies m'_1 as a candidate. It creates the instruction summaries for m_1 and m'_1. The analysis of an instruction summary of m_1 proceeds as described in Algorithm 2.

For each previously computed summary, whenever there is an `invoke` instruction (Line 3), the algorithm visits the callee methods (m_2) and retrieves its instruction graph. Next, it checks whether that method has been inlined with the caller's summary (Line 5). If it has, the analysis has already been done and the algorithm terminates. If it has not, the algorithm uses the DFS visitor method to compute an instruction summary for m_2 (Lines 5 to 8). Finally, it merges the result with the summary of m_1 (Line 9). Next, the algorithm removes the invoke instruction from the caller's summary (Line 10). This process also applies when a callee method calls other methods. It is repeated until there are no more invoke instructions in the caller's or callee's summaries. It then performs Equivalence Analysis on the two summaries.

Note that SEMEO can handle method outlining obfuscation by simply applying the same inlining analysis to the obfuscated method being compared instead of the original method. Inlining analysis can also be used to handle obfuscation through method indirection.

The comparison process applies several heuristics. Among others, by using the instruction summary, the algorithm can remove all inserted or removed code that does not change or only slightly change the data-flow information of the obfuscated program (e.g., junk code insertion). However, obfuscation approaches

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 ← GetCalleeMethod(Ins)
5:             if NotInList(M,Inlined) then
6:                 Inlined ← Inlined ∪ M
7:                 G ← ExtractDataFlow(M)
8:                 Sum_i ← ExtractInstructionSummary(G)
9:                 Sum ← Sum ∪ Sum_i
10:                Sum ← Sum − Ins
11:            end if
12:        end if
13:    end for
14: end procedure
```

such as those that change loop structures would also change data-flow patterns. To handle this, we also refer to common loop patterns (e.g., `for` loops and `while` loops) 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 code reordering obfuscation techniques, the purpose is to alter the execution flow of the program without changing its semantic meaning. SEMEO handles code reordering by deriving a canonical form of the summaries (Sum and Sum_i) by sorting them alphabetically in terms of their Dex instructions before conducting the comparison. Because each summary contains the relationship between a register and associated Dex instructions, after sorting, the order of appearance of instructions is no longer preserved. This helps remove effects caused by code reordering. In this case, the comparison between the two summaries would focus only on the register with its associated instructions.

After comparing instructions and data flow information, our approach analyzes constant values, which may provide additional insights because some constant values including strings may not change, since 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 *similarity score* based on 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 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'.

4 Empirical Study

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

- **RQ1.** What is the overall accuracy of Semeo in detecting whether an app and a semantically equivalent obfuscated version of that app are in fact semantically equivalent compared to the state-of-the-art technique?
- **RQ2.** How effective is Semeo in identifying repackaged methods in obfuscated apps compared to the state-of-the-art technique?
- **RQ3.** How efficient is Semeo? What is the performance of Semeo's semantic equivalence analysis?

4.1 Objects of Analysis

While Semeo can operate directly on Dex code, for this study we needed to have source code so that we could apply the targeted obfuscation techniques (inlining and outlining) for which no automated tools could be found (an issue that is discussed further below). Finding malware samples that meet our requirements for objects of study in numbers that support quantitative analysis, however, is challenging. For example, commonly used malware samples for academic research are quite old and rarely have publicly available source code (i.e., they are distributed only as Dex code). Therefore, to answer RQ2, we also needed to create alternative versions of each original app by modifying some of their methods.

Ultimately, we selected 14 apps, nine apps that were created by DARPA to support their Automated Program Analysis for Cybersecurity (APAC) program [12] and five pairs of apps with known histories of revisions. To address RQ1, we applied various obfuscation techniques to create obfuscated versions. For the five pairs of revised apps, we used the original version of each pair of apps.

To address RQ2, we required repackaged apps. When considering malware, we are interested in methods into which malicious code has been injected, and to which obfuscations have then been applied. For the nine apps from DARPA, we asked an undergraduate student not familiar with this work to randomly select a number of methods for each app, prior to applying any obfuscation types, and manually modify their code. Target modifications involved relatively simple but provably semantics-affecting changes such as negating branch conditions, changing input parameters, removing method contents, and changing return types. 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. For the remaining five pairs of revised apps, we simply obfuscated the revised version of each app and then compared the obfuscated versions with the original (i.e., unmodified and unobfuscated) app. This gives us the ability to evaluate apps that have been modified by real-world practitioners. Because these apps come with revision information including revised methods, we used these revised locations to help compute recall and precision.

Table 2. Objects of analysis

	App	Name	Methods	LoDC	Modified methods (RQ2)			
					G1–G5	G6	G7	G8
Apps from DARPA APAC program	App1	PicViewer	21	3888	2	2	2	2
	App2	CalcC	63	4593	6	6	6	6
	App3	DeviceAdmin2	161	9262	20	10	16	10
	App4	Orienteering	697	35263	20	20	20	20
	App5	SysMon	752	32549	18	10	10	10
	App6	Pondl	1573	58667	99	10	10	10
	App7	YARR	2027	48050	57	2	2	2
	App8	NewsCollator	2935	51806	19	2	2	2
	App9	TextSecure	7218	112804	243	10	10	10
Apps with known Rev.	App10	arxiv mobile	323	4684	8	5	5	10
	App11	KISS Launcher	811	11631	513	5	5	10
	App12	SolitaireCG	455	7260	12	5	5	10
	App13	Vanilla Music	1613	34103	23	5	5	10
	App14	WorkoutLog	748	1274	1	5	5	10

While the modifications and revisions we studied do not involve insertions of malicious code, we argue that malicious code would most likely be more complicated than these subtle modifications; thus, if Semeo is able to correctly detect non-equivalent methods from among these, it is likely to be able to do so for methods involving actual malicious code. To verify this, we also conducted a case study to detected repackaged components using a malicious version of *Pokémon Go* (see Sect. 6.2).

Table 2 describes our objects. Column 1 provides identifiers that are used later to refer to the apps, Column 2 provides the app's actual names, Columns 3 and 4 list the numbers of methods and lines of Dex code (LoDC) in the apps. The numbers of modified methods created and used in our study are shown in Columns 5–8. As the table shows, the apps ranged in size from 21 to 7,218 methods, and from 1,274 to 112,804 lines of Dex code.

Next, we considered the seven DSA obfuscation types discussed in Sect. 2, where we assigned a "Type ID" (T1–T7) to each obfuscation type. For types T1–T5, we used Alan [27], an Android malware obfuscation engine capable of applying one or more of these types to a given APK in any order. We chose ALAN because it is a state-of-the-art obfuscation tool; also, since Semeo is not specifically designed to target ALAN obfuscations, this lets us avoid a potential threat to validity for the study.

Alan can be configured to obfuscate an entire app or just a portion of an app. We chose to obfuscate entire apps to create a scenario similar to the one created by malware authors when they obfuscate repackaged apps. Alan is able to apply obfuscation types T1–T5 individually or in any combinations, so we chose five methods for grouping obfuscation types (Table 3). Grouping $G1$ considers single obfuscation types; since there are five obfuscation types this yields cases in which just obfuscation type T1 is applied, just obfuscation type T2 is applied, and so

Table 3. 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

forth. Grouping $G2$ considers all pairs of obfuscation types; for example, "T12" refers to the case in which obfuscation type T1 is applied followed by 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.

Unfortunately, as noted above, 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 two students who at that time had no knowledge of our approach for determining semantic equivalence. For inlining, we instructed the students to inline string operations and the contents of called methods. For outlining, we instructed the students to group branch condition bodies into other methods and to move some parts of functions into other small functions. The students applied these modifications to methods randomly selected from each of the apps. We applied function inlining and outlining singly and together; thus, there are three sequences of applications of these obfuscation types, which we refer to as G6, G7, and G8. We also sign these obfuscated apps with *SignAPK*[2] so that FSquaDRA (discussed next) can analyze them.

4.2 Variables and Measures

Independent Variables. Typically, when studying a new program analysis technique, we choose the technique itself as our primary independent variable, locating one or more other baseline techniques to compare against. For this study, we chose FSquaDRA as a baseline technique. FSquaDRA's ultimate goal is to detect repackaged apps while being insensitive to code changes. To do this, FSquaDRA analyzes the contents of the resource files instead of just analyzing code and reports how similar the two files are based on the Jaccard similarity

[2] Available from https://github.com/techexpertize/SignApk.

coefficient. Fundamentally, FSquaDRA's analysis attempts to identify similar resource files among all of the resource files used by the apps. To speed up the process, it uses hash of each file generated as part of app signing [42].

The main rationale behind FSquaDRA resides in an observation that a repackaged app attempts to change the app in a manner that maintains the app's look and feel. Therefore, the code may change but the resource files should remain the same [42]. As such, the results provided by FSquaDRA should be comparable to those provided by Semeo in terms of their abilities to overcome the layer of analysis complexity induced by code obfuscation. While FSquaDRA would not be able to precisely locate methods that have been modified, we still expect that for RQ1 and RQ2, the similarity score produced by comparing the original app to an obfuscated version of the same app would be very high.

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

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. We calculate recall using the following formula: $\frac{m_{seq}}{m_{total}-m_{mod}}$. Here, m_{total} represents the total number of methods, and m_{mod} represents the total number of modified methods. As such, the difference between them is the total number of semantically equivalent methods. m_{seq} *represents the number of semantically equivalent methods that have been correctly identified*. Note that for RQ1, the total number of modified methods for each app is 0 (i.e., $m_{mod} = 0$). For RQ2, recall is calculated using the same equation, but in this case each app has been modified so m_{mod} is non-zero.

Similarity. FSquaDRA reports similarity scores, each of which represents the ratio between the number resource files identified as similar (r_{sim}) and the total number of resource files (r_{total}) (i.e., $similarity = \frac{r_{sim}}{r_{total}}$). As such, for RQ1, our recall value is the same as FSquaDRA's similarity score; they both report the degree of similarity between the two apps. For RQ2, our recall value is different than FSquaDRA's similarity score. This is because when we calculate recall, the denominator of our formula represents only the number of unmodified methods. However, when FSquaDRA computes similarity, the denominator of FSquaDRA's Jaccard similarity coefficient represents the total number of resource files and not just the unmodified ones. Even though the two metrics differ, they both report the degree of similarity between the two apps. As such, we simply compare FSquaDRA's similarity scores with our recall values.

Precision. Precision represents Semeo's ability to avoid incorrectly identifying methods that are not semantically equivalent as semantically equivalent.

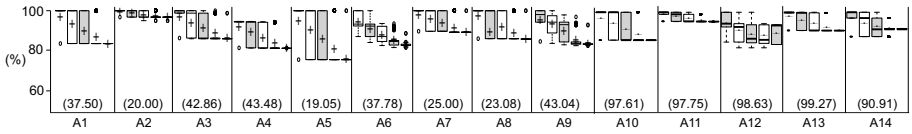

Fig. 3. Recall and similarity for obfuscation type groupings G1–G5 on RQ1 apps.

Modified methods mis-identified as semantically equivalent can be 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, we calculate precision as $\frac{m_{seq}}{m_{neq}+m_{seq}}$, where m_{seq} again represents the number of methods in the app correctly identified as semantically equivalent (true positives), and m_{neq} represents the number of modified methods that have been mistakenly identified as semantically equivalent to original methods (false positives).

For example, suppose an app A has 100 methods (m_{total}), and five of these methods have been modified (m_{mod}). Suppose that SEMEO identifies 90 methods as semantically equivalent, with four methods incorrectly identified (m_{neq}) and 86 methods correctly identified (m_{seq}). In this case, recall is $\frac{86}{100-5}$ or 91% and precision is $\frac{86}{4+86}$ or 96%.

Efficiency. We calculate efficiency by measuring the time required by SEMEO to perform its analysis. We use seconds to report our results. The measurement is from the time SEMEO loads the two apps to when the analysis result is reported.

4.3 Study Operation

To address RQ1, we used SEMEO and FSQUADRA to compare the obfuscated apps with the original apps, and noted how many methods in the obfuscated apps were flagged as semantically equivalent to the original ones. To address RQ2 we applied the same process, but in this case we also noted how the results related to methods that were actually modified. To address RQ3 we followed the same process as for RQ1 and RQ2, and measured the amount of time needed to perform the analysis by each technique. To perform this study we used a MacBook Pro running OS X El Capitan version 10.11.2, with an 8 GB memory and a 2.5 GHz Intel Core i5. The performance times we gather are all recorded within this environment.

4.4 Threats to Validity

Where external validity is concerned, due to the time and effort required to apply Alan to the enormous numbers of combinations of obfuscations in grouping G1–G5, and the cost of manually applying obfuscation groupings G6–G8, we have studied only 14 apps, but they do represent an important sub-class of the apps that malware authors target, and they do vary in size and complexity. Furthermore, five of these apps contain modifications to the original apps so they represent real

modifications made by developers. In addition, two of our obfuscation types, inlining and outlining, were applied by hand. We also do not consider actual malware, instead using semantic modifications made by software developers. One of the further studies we present in Sect. 6.2, however, helps address this threat by considering an application that does contain actual malware.

Where internal validity is concerned, errors in the tools we rely on could affect our results, but we have attempted to rigorously test them. Where construct validity is concerned, we measure precision, recall, and analysis time, but we do not collect any measures related to actual engineer effort.

4.5 Results for RQ1

RQ1 concerns the effectiveness of Semeo and FSquaDRA at detecting whether an app and a semantically equivalent obfuscated version of that app are in fact semantically equivalent. Figure 3 presents boxplots showing the distribution of recall values achieved by Semeo for obfuscation type groupings $G1$ through $G5$ (see Table 3 for grouping definitions) on all 14 apps. In the figure, the x-axis organizes the data per app. For each app, five boxes display the data for obfuscation type groupings $G1$ (leftmost box) through $G5$ (rightmost box), respectively. The y-axis reports recall percentages, computed using the equation provided in Sect. 4. The mean value within each grouping is denoted by a "+". We also include the reported similarity value for FSquaDRA in Fig. 3 just above the x-axis for each app (e.g., 37.50 for App1). We report only one value per app because FSquaDRA reports the same recall value for each app on obfuscation types $G1$–$G5$. Again, this is not surprising because FSquaDRA mainly analyzes resource files.

For each app, as obfuscation complexity increased, mean recall decreased. For obfuscation type grouping $G1$, in 12 of 14 cases there is little variance in results, and the mean recall values are between 93% (App12) and 100% (App2), indicating that with single obfuscations applied, Semeo was highly effective at identifying semantically equivalent methods. Even in cases in which the most complex obfuscations were applied (obfuscation type grouping $G5$) the mean recall exceeds 80% on 13 of 14 apps. The lowest mean recall (76%) occurs on App5 for the most complex obfuscation type grouping $G5$. In nine of 14 cases $G2$ and $G3$ are the only obfuscation type groupings that display large degrees of variance. 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.

When we compare the performance of Semeo to that of FSquaDRA, we see that Semeo maintained a recall average of over 80% on 13 of 14 apps for $G1$–$G5$, whereas the similarity scores of FSquaDRA did not exceed 44% on the first nine apps (an average of 32.42%). FSquaDRA did perform substantially better, however, on the last five apps, containing real developer modifications. We discuss the main cause of this difference in performance in Sect. 5.

For function inlining and outlining (G6–G8), we compare the recall performance of Semeo with the similarity score of FSquaDRA in Table 4. (Boxplots

Table 4. Recall and similarity for obfuscation type groupings G6–G8 on RQ1 apps

	G6		G7		G8	
	Sem.	FSq.	Sem.	FSq.	Sem.	FSq.
App1	90.48	69.23	91.3	69.23	82.61	69.23
App2	95.45	63.6	95.52	63.6	80.56	63.6
App3	94.41	95.74	90.96	95.74	91.72	95.74
App4	95.25	77.78	94.41	77.78	94.24	77.78
App5	100.00	92.31	99.61	92.31	99.61	92.31
App6	89.92	93.75	95.77	93.75	91.14	93.75
App7	100.00	76.47	99.86	76.47	99.95	76.47
App8	99.52	77.78	99.9	77.78	99.8	77.78
App9	83.33	38.12	83.72	38.12	83.22	38.12
App10	95.36	97.61	100.00	97.61	95.43	97.61
App11	95.85	97.75	100.00	97.75	95.89	97.75
App12	98.90	98.63	100.00	98.63	98.91	98.63
App13	98.76	99.27	99.32	99.27	98.15	99.27
App14	99.06	90.91	98.93	90.91	98.15	90.91
Average	95.44	83.50	96.38	83.50	93.53	83.50

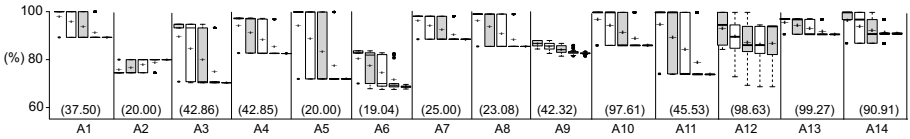

Fig. 4. Recall and similarity for obfuscation type groupings G1–G5 on RQ2 apps

are not appropriate in this instance, because we do not use multiple permutations of obfuscation technique orderings in this case, and thus do not have a distribution of data points). We find that SEMEO is more accurate than FSQUADRA on nine of the 14 apps. On average, SEMEO achieved average recalls of 95.44, 96.38, and 93.53 for $G6$, $G7$, and $G8$, respectively, whereas the average of the similarity scores for FSQUADRA is 83.50%. We discuss this further in Sect. 5.

4.6 Results for RQ2

RQ2 concerns the effectiveness of SEMEO at identifying repackaged methods in obfuscated apps. With respect to G1 and G5, Fig. 4 shows that SEMEO can achieve recall values ranging from 69% (App6, G5) to 100% (several apps). Also note that FSQUADRA's similarity scores did not exceed 45% on 10 of the 14 apps; in fact, its average similarity score across all 14 apps was only 45.30%.

We also applied SEMEO to groupings G6, G7, and G8; Table 5 reports the average recall values and similarity scores. As shown, the greatest decrease in recall occurred on App2, which is a small app (recall levels decrease from 95.45%

Table 5. Recall and similarity for obfuscation type groupings G6–G8 on RQ2 apps

	G6		G7		G8	
	Sem.	FSq.	Sem.	FSq.	Sem.	FSq.
App1	84.21	69.23	72.73	69.23	72.73	69.23
App2	48.61	63.6	70.49	63.6	48.53	63.6
App3	88.98	95.74	77.24	95.74	80.56	95.74
App4	98.23	77.78	95.70	77.78	98.08	77.78
App5	99.18	92.31	97.08	92.31	97.61	92.31
App6	82.45	93.75	81.46	93.75	80.59	93.75
App7	81.64	76.47	80.84	76.47	80.84	76.47
App8	87.50	77.78	86.98	77.78	86.94	77.78
App9	90.69	21.8	90.69	21.8	90.69	21.8
App10	95.63	95.29	99.69	95.29	96.88	95.29
App11	57.39	45.53	57.83	45.53	57.76	45.53
App12	99.11	89.54	100.00	89.54	94.85	89.54
App13	95.83	99.27	95.78	99.27	95.65	99.27
App14	99.46	40.00	99.47	40.00	98.19	40.00
Average	86.35	74.15	86.14	74.15	84.28	74.15

for G6 and 80.56% for G8 in Table 4 to 48.61% and 48.53%, respectively, in Table 5). FSquaDRA, on the other hand, maintained nearly the same recall performance as achieved for the first nine apps in relation to RQ1, where it performed worse than Semeo. On average, Semeo still achieved an average recall of 86.35% while the average similarity score of FSquaDRA was 74.15%.

Turning to precision, Semeo achieved 100% precision on 13 of 14 repackaged apps, for all obfuscation types and obfuscation type groupings. The only exception was App11, which has the largest number of modified method (513). For this app, the average precision achieved by Semeo for G1 and G5 were 73% and 98%, respectively. (FSquaDRA, as noted earlier, does not report details at the method level; it reports only whether an app has been repackaged with respect to an original app through a comparison of resource usage – it analyzes certificate and manifest files. Thus, we cannot compare its precision to that of Semeo.)

4.7 Results for RQ3

RQ3 concerns the efficiency of Semeo. Our evaluation shows that the analysis overheads of apps measured while exploring RQ1 generally fell into four categories: (1) negligible overhead (a fraction of a second, App1–App3), (2) low overhead (less than 10 s, Apps4, App5, App10, App11, App12, App13, and App14), (3) moderate overhead (several hundred to 1000 seconds, App6 and App8), and (4) high overhead (several thousand seconds, App9). We also found that the runtime overhead of Semeo increased as the obfuscation type grouping became more

Table 6. Performance of SEMEO and FSQUADRA on obfuscation type groupings G6–G8 on RQ1 apps (seconds)

	G6		G7		G8	
	Sem.	FSq.	Sem.	FSq.	Sem.	FSq.
App1	0.27	0.10	0.29	0.07	0.57	0.07
App2	0.31	0.09	0.29	0.10	0.40	0.09
App3	0.47	0.08	0.46	0.11	0.58	0.10
App4	2.14	0.10	2.33	0.09	2.47	0.12
App5	1.02	0.09	1.07	0.10	1.13	0.10
App6	125	0.11	55.88	0.14	117.99	0.11
App7	7.36	0.10	9.24	0.07	8.09	0.07
App8	26.95	0.12	27.24	0.10	21.21	0.10
App9	6808.98	0.15	7113.36	0.10	6056.00	0.09
App10	0.71	0.10	0.69	0.10	0.71	0.13
App11	1.74	0.14	1.78	0.14	1.75	0.13
App12	1.03	0.13	1.00	0.11	0.96	0.12
App13	2.61	0.13	2.46	0.12	2.45	0.12
App14	0.85	0.08	0.82	0.09	0.76	0.10

complex. The analysis overheads for FSQUADRA also increase across groupings, ranging from 10.54 ms to 30.42 ms. As previously reported, however, the recall performances for FSQUADRA are generally much lower than those for SEMEO.

Table 6 displays the performance values for SEMEO for the inlining, outlining, and hybrid obfuscation type groupings ($G6$–$G8$). As the data shows, SEMEO was quite efficient when the number of methods it analyzed was of small to moderate size. Only as the number of methods in an app exceeded 3000 (Apps 8 and 9) did SEMEO's worst case analysis time increase substantially, in those cases from 1267 s (21 min) to 12693.6 s (2.7 h). SEMEO was much faster when the original and obfuscated apps being analyzed were semantically identical. As previously reported in Table 4, the algorithm had nearly 100% recall for RQ1 on Apps7 and App8. In this case, the analysis times were also quite fast (less than 30 s as shown in Table 6) in spite of the presence of over 2000 methods. The analysis times for FSQUADRA, on the other hand, displayed a linear increase across all apps for grouping $G6$–$G8$ (70 ms to 170 ms). While FSQUADRA was more efficient than SEMEO, it was not as effective; its similarity scores were very low in many cases.

With respect to apps considered while exploring RQ2, the analysis overheads also fall into the same four categories, with the same set of apps for each category (e.g., App9 falls into the high overhead category). The analysis time for FSQUADRA continued to increase linearly across all groupings (from 10.13 ms to 46.36 ms) while yielding lower similarity scores for these groupings as previously reported in Fig. 4.

Table 7. Performance of SEMEO and FSQUADRA on groupings G6–G8 on RQ2 apps (seconds)

	G6		G7		G8	
	Sem.	FSq.	Sem.	FSq.	Sem.	FSq.
App1	0.31	0.07	0.38	0.07	0.34	0.07
App2	0.57	0.08	0.45	0.08	0.62	0.08
App3	1.05	0.08	0.90	0.08	1.29	0.08
App4	3.41	0.07	12.15	0.07	2.76	0.07
App5	2.71	0.07	3.62	0.08	3.84	0.07
App6	236.91	0.07	246.06	0.08	259.06	0.08
App7	403.91	0.07	419.63	0.07	414.03	0.07
App8	813.11	0.07	855.74	0.07	818.56	0.08
App9	6570.87	0.12	6738.51	0.12	6506.24	0.12
App10	0.68	0.08	0.73	0.07	0.72	0.07
App11	1.80	0.08	1.72	0.08	1.75	0.07
App12	1.01	0.09	0.97	0.08	0.96	0.07
App13	2.67	0.08	2.71	0.08	2.64	0.08
App14	0.78	0.06	0.81	0.07	0.69	0.07

Table 7 displays the performance values for SEMEO and FSQUADRA for the inlining, outlining, and hybrid obfuscation type groupings. The analysis times for RQ2 were generally consistent with those for RQ1, with the exception of the analysis times for App7 and App8. For these two apps, the analysis times increased from nine seconds to 404 s and from 27 s to 856 s, respectively. The performance overhead in these cases arises primarily due to the complexity of Algorithm 1 ($O(n^2)$). FSQUADRA, however, performs consistently across the three groupings; its analysis times ranged from 70 ms to 120 ms. Its similarity scores, however, were also quite low for eight of 14 apps as previously reported in Table 5.

5 Discussion

The main underlying hypothesis of FSQUADRA is that malware author would less likely to change the resource files as part of repackaging. However, our investigation shows that in apps that heavily use resources, certain types of obfuscation can have affect on these resources and thus, affecting FSQUADRA's ability to identify repackaged counterparts of an original app. We have seen similarity scores as low as 19.05% when it compares two apps; one has no modifications except for the various layers of obfuscation applied to it. This indicates that obfuscation can render FSQUADRA ineffective. SEMEO, on the other hand, due to its high recall performance can also be used to accurately identify repackaged counterparts of an app. Furthermore, by analyzing code instead of resource

usage, it can also precisely report methods that have been modified, allowing analysts to directly analyzed those methods for malicious behaviors.

Our results show that some methods in obfuscated apps are incorrectly deemed by SEMEO to be non-equivalent to those in the original apps, for several reasons. First, SEMEO analyzes obfuscation grouping types $G1$ and $G2$ very accurately but this is less true for grouping types $G3$, $G4$, and $G5$, where three, four, and five layers of obfuscation have been applied. Such high degrees of composite obfuscation render analyzing apps for semantic equivalence more challenging.

As a second reason, we discovered that when we introduced modified methods that change application semantics, our recall degrades slightly because these modified methods can affect superclass and subclass relationships. Some modifications to global variables in modified methods can affect other methods that share these 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 additional overhead is also very small, because the reported time for RQ2 is about the same as that for RQ1 on the same app.

For the apps that FSQUADRA returns high similarity scores, we noticed that obfuscation has very little effect on the size of the main class file (`classes.dex`). As an example, for App13, the size of `classes.dex` changes by 8% after obfuscation and the similarity score as reported in the study to answer RQ1 is 99.27%. However, for App14, the size of the class file increases by 54% after obfuscation and the similarity score decreases to only 90%. This shows indirect sensitivity to obfuscation. Furthermore, FSQUADRA does not report the locations of the code changes. Thus, analysts still need to identify such locations themselves. This can be quite challenging in heavily obfuscated apps.

In terms of analysis time, the analysis performed by FSQUADRA appears to take nearly constant time for both RQ1 and RQ2 in spite of the significant changes made to the code via various obfuscation sequences. This is because it focuses only on resource usage and not the code so obfuscation complexity applied to the code has no affect on its analysis time.

6 Additional Studies

We now report the results of two additional studies designed to evaluate the accuracy and performance of SEMEO in realistic settings. In the first study, we apply SEMEO to detect semantically equivalent methods when PROGUARD, a commonly used commercial obfuscation tool, is used in addition to our own obfuscation methods. In the second study, we use SEMEO to detect modified methods in a complex, real-world repackaged malware sample.

6.1 Applying SEMEO on Apps Obfuscated by PROGUARD

We assessed whether SEMEO can identify semantically equivalent methods in apps that have been obfuscated by PROGUARD. To perform our evaluation, we applied PROGUARD to the source code of apps that contain modifications and

Table 8. Recall and similarity (%) achieved by SEMEO and FSQUADRA when applying PROGUARD (groupings G1 and G5).

Apps	Modified methods	G1		G5	
		Sem.	FSq.	Sem.	FSq.
App1	3	73.34	37.50	55.56	37.50
App3	10	68.47	42.86	64.74	42.86
App7	11	94.36	25.00	84.11	25.00
App6	10	94.87	37.78	91.31	37.78
Average		82.76	35.79	73.93	35.79

were used to answer RQ2. We observed that once we applied PROGUARD to an app, the structure of the app can change due to optimizations such as dead code removal and code inlining. This can lead to different numbers of modified methods than those listed in Table 2. We then applied obfuscation type grouping G1 (individual obfuscation types) and obfuscation type grouping G5 (the largest composite obfuscation type grouping) to the apps. Due to space limitation, Table 8 only reports the results of apps from smallest (App1 and App3) to the largest (App7 and App6) based on LoDC, as previously reported in Table 2. The table lists the modified methods used and the recall and similarity values found, for SEMEO and FSQUADRA. Note that we could not apply PROGUARD to our largest app (App9).

As Table 8 shows, SEMEO is more effective than FSQUADRA at detecting semantically equivalent methods when we apply two or more layers of obfuscations (PROGUARD followed by our own additional obfuscation types). SEMEO achieved recall values ranging from 68.47% to 94.87% when two layers of obfuscation were applied, with an average recall value of 82.76%. When six layers of obfuscations were applied (i.e., PROGUARD and *G*5), SEMEO achieved recall values ranging from 55.56% to 91.31% with an average recall value of 73.93%. FSQUADRA, on the other hand, achieved similarity values ranging from 25% to 42.86% with an average of 35.79%. Further, as noted in Sect. 4.5, FSQUADRA was not sensitive to differences in obfuscation type groupings.

6.2 Applying SEMEO on Real-World Repackaged Malware

Popular apps attract the attention of cyber-criminals because they are highly downloaded, and thus, if repackaged, can infect a large number of devices quickly. For example, one version of *Pokémon Go* was repackaged with *DroidJack*, a *Remote Access Tool* or *RAT*. In this case, the malware author downloaded a legitimate version of *Pokémon Go* 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 [8,11,19,34]. 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 the aforementioned repackaged release of *Pokémon Go*. The legitimate version contains 37,024 methods, whereas the repackaged version contains 38,878 methods. We used information from a security analysis result [19] 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 in fact include *all* of the modified methods that have previously been reported [19]. The analysis time required was approximately 2,300 s. We also used FSQUADRA to analyze these two versions of *Pokémon Go* and it produced the similarity score of 89.47%.

7 Related Work

REPDETECTOR [16], like FSQUADRA, performs semantic equivalence analysis by monitoring input and output states and employing an SMT solver to compute similarity scores between pair of apps. REPDETECTOR is efficient because it only focuses on core functions. This optimization may be reasonable for app-level comparisons but may be too coarse grained for method-level comparisons, which is the focus of our technique. Last, REPDETECTOR has not been publicly released.

In addition, there are de-obfuscation tools that can be used to indirectly help with the task SEMEO performs. In particular, a de-obfuscator can be applied to obfuscated modules, and then the de-obfuscated modules can be differenced against original unobfuscated modules. We considered using these in our empirical work but they all had drawbacks. ANDROSIM, which is a commonly used Android reverse engineering tool in the Androguard toolset [13], identifies similarities between two applications; however, it can handle only simple obfuscation types and it misclassifies many obfuscated methods in our apps as non-equivalent with respect to the original methods. Dex-oracle [3] looks only for specific patterns, and it cannot deobfuscate our programs. *Simplify* [4] is a deobfuscation-by-decryption tool so it does not function with the obfuscation techniques used in this work.

There has been work on detecting obfuscation types. Myles et al. [28] analyze binary code to look for similarities based on K-gram-based software birthmarks. However, their approach cannot handle junk code insertion or code reordering. SAFE, a malware detector, can handle only simple obfuscations (e.g., NOP insertions) [5]. Kruegel [24] uses static analysis on binaries to detect kernel-level rootkits. Apposcopy [15] is a semantics-based analysis tool that detects Android malware signatures. Dexteroid [21] is a tool that detects behavior-based malware based on the Android life cycle model. These tools cannot analyze obfuscated apps for semantic equivalence.

There are also tools such as ANDARWIN [10] and DNADROID [9] to detect cloned apps. As such, they are capable of working with some forms of obfuscation techniques. However, they are not publicly available. 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 [30,31,36].

8 Conclusion and Future Work

We have presented Semeo, a technique for directly identifying (without first deobfuscating) obfuscated methods in Android apps that are semantically equivalent to original non-obfuscated methods. Our comparison of Semeo and FSquaDRA indicates that Semeo is more accurate at identifying methods that have been modified. We also evaluated the capability of Semeo to deal with ProGuard and found that it can effectively handle obfuscation types that ProGuard utilizes. Last, we used Semeo and FSquaDRA to identify repackaged components of a version of *Pokémon Go* malware. Semeo could detect all modified malicious methods while FSquaDRA mistakenly identified over 2000 benign methods as malicious.

In future work we intend to improve the scalability of our approach as discussed in Sect. 5. 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. We will publicly release Semeo after the publication of this work.

References

1. Ponomarenko, A.: A tool for checking backward API/ABI compatibility of a Java library (2013). https://github.com/lvc/japi-compliance-checker
2. Android. Dalvik bytecode (2015). https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html
3. Fenton, C.: A pattern based Dalvik deobfuscator which uses limited execution to improve semantic analysis (2015). https://github.com/CalebFenton/dex-oracle
4. Fenton, C.: Generic Android Deobfuscator (2015). https://github.com/CalebFenton/simplify
5. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12, SSYM'03, Washington, DC, pp. 12 (2003)
6. Collberg, C., Myles, G., Huntwork, A.: Sandmark-a tool for software protection research. IEEE Secur. Priv. **1**(4), 40–49 (2003)
7. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, 1st edn. Addison-Wesley Professional, Boston (2009)
8. Contagio Mini Dump: Pokemon GO with Droidjack - Android sample (2016). http://contagiominidump.blogspot.com
9. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: detecting cloned applications on android markets. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 37–54. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33167-1_3

10. Crussell, J., Gibler, C., Chen, H.: AnDarwin: scalable detection of semantically similar android applications. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013. LNCS, vol. 8134, pp. 182–199. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40203-6_11
11. Goodin, D.: Fake Pokémon Go app on Google Play infects phones with screenlocker (2016). http://arstechnica.com/security/2016/07/fake-pokemon-go-app-on-google-play-infects-phones-with-screenlocker/
12. DARPA: Automated Program Analysis for Cybersecurity (APAC) (2012). http://www.darpa.mil/program/automated-program-analysis-for-cybersecurity
13. Desnos, A.: AndroGuard, May 2013. http://androguard.blogspot.com/
14. Duan, Y., et al.: Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In: 25th Annual Network and Distributed System Security Symposium, NDSS, San Diego, CA, pp. 18–21 (2018)
15. Feng, Y., Anand, S., Dillig, I., Aiken, A.: 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, pp. 576–587. ACM, New York (2014)
16. Guan, Q., Huang, H., Luo, W., Zhu, S.: Semantics-based repackaging detection for mobile apps. In: Caballero, J., Bodden, E., Athanasopoulos, E. (eds.) ESSoS 2016. LNCS, vol. 9639, pp. 89–105. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30806-7_6
17. I. IDC Research: Smartphone OS Market Share, 2015 Q2. IDC Research Report (2015)
18. Jones, N.D.: Computability and Complexity: From a Programming Perspective. MIT Press, Cambridge (1997)
19. Sullivan, J.: Pokémon Go bundles with malicious remote administration tool DroidJack (2016). http://blog.trustlook.com/2016/09/02/pokemon-go-bundles-with-malicious-remote-administration-tool-droidjack/
20. Cannell, J.: Obfuscation: malware's best friend (2013). https://blog.malwarebytes.com/threat-analysis/2013/03/obfuscation-malwares-best-friend/
21. Junaid, M., Liu, D., Kung, D.C.: Dexteroid: detecting malicious behaviors in android apps using reverse-engineered life cycle models. CoRR arXiv:1506.05217 (2015)
22. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 155–169 (2000)
23. Konstantinou, E.: Metamorphic virus: analysis and detection. Technical Report RHUL-MA-2008-02, Royal Holloway, University of London, January 2008
24. Kruegel, C., Robertson, W., Vigna, G.: Detecting kernel-level rootkits through binary analysis. In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC'04, Tucson, AZ, USA, pp. 91–100 (2004)
25. Li, Z., Sun, J., Yan, Q., Srisa-an, W., Tsutano, Y.: Obfusifier: obfuscation-resistant android malware detection system. In: Chen, S., Choo, K.-K.R., Fu, X., Lou, W., Mohaisen, A. (eds.) SecureComm 2019. LNICST, vol. 304, pp. 214–234. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37228-6_11
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 (2015). http://seclist.us/alan-android-malware-evaluating-tools-released.html

28. Myles, G., Collberg, C.: K-gram based software birthmarks. In: Proceedings of the 2005 ACM Symposium on Applied Computing, SAC'05, Santa Fe, New Mexico, pp. 314–318 (2005)
29. National Cyber Security Center (UK): Code Obfuscation (2014). https://www.ncsc.gov.uk/content/files/protected_files/guidance_files/Code-obfuscation.pdf
30. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: SIGPLAN Not., vol. 49, no. 10, pp. 811–828 (2014)
31. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT'08/FSE-16, pp. 226–237. ACM, New York (2008)
32. Pomilia, M.: A study on obfuscation techniques for android malware. Technical Report, Sapienza University of Rome, March 2016. http://midlab.diag.uniroma1.it/articoli/matteo_pomilia_master_thesis.pdf
33. Preda, M.D., Maggi, F.: Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. J. Comput. Virol. Hacking Tech. **13**(3), 209–232 (2016). https://doi.org/10.1007/s11416-016-0282-2
34. Proofpoint Staff: DroidJack Uses Side-Load...It's Super Effective! Backdoored Pokemon GO Android App Found (2016). https://www.proofpoint.com/us/threat-insight/post/droidjack-uses-side-load-backdoored-pokemon-go-android-app
35. Rad, B.B., Masrom, M.: Metamorphic virus variants classification using opcode frequency histogram. In: Proceedings of the International Conference on Computers, pp. 147–155 (2010)
36. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 669–685. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_55
37. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the ACM Symposium on Information, Computer and Communications Security, pp. 329–334 (2013)
38. Siek, J.G., Lee, L.-Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
39. Tsutano, Y., Bachala, S., Srisa-an, W., Rothermel, G., Dinh, J.: An efficient, robust, and scalable approach for analyzing interacting android apps. In: Proceedings of the International Conference on Software Engineering, Buenos Aires, Argentina, May 2017
40. Tsutano, Y., Bachala, S., Srisa-an, W., Rothermel, G., Dinh, J.: JITANA: a modern hybrid program analysis framework for android platforms. J. Comput. Lang. **52**, 55–71 (2019)
41. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
42. Zhauniarovich, Y., Gadyatskaya, O., Crispo, B., La Spina, F., Moser, E.: FSquaDRA: fast detection of repackaged applications. In: Atluri, V., Pernul, G. (eds.) DBSec 2014. LNCS, vol. 8566, pp. 130–145. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43936-4_9