Evaluating Model Checking for Cyber Threats Code Obfuscation Identification

Fabio Martinelli\textsuperscript{a}, Francesco Mercaldo\textsuperscript{a}, Vittoria Nardone\textsuperscript{b}, Antonella Santone\textsuperscript{c}, Arun Kumar Sangaiah\textsuperscript{d}, Aniello Cimitile\textsuperscript{b}

\textsuperscript{a}Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Pisa, Italy
\textsuperscript{b}Department of Engineering, University of Sannio, Benevento, Italy
\textsuperscript{c}Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy
\textsuperscript{d}School of Computing Science and Engineering, VIT University, Vellore 632014, India

Abstract

Code obfuscation is a set of transformations that make code programs harder to understand. The goal of code obfuscation is to make reverse engineering of programs infeasible, while maintaining the logic on the program. Originally, it has been used to protect intellectual property. However, recently code obfuscation has been also used by malware writers in order to make cyber threats easily able to evade antimalware scanners. As a matter of fact, metamorphic and polymorphic viruses exhibit the ability to obfuscate their code as they propagate. In this paper we propose a model checking-based approach which is able to identify the most widespread obfuscating techniques, without making any assumptions about the nature of the obfuscations used. We evaluate the proposed method on a real-world dataset obtaining an accuracy equal to 0.9 in the identification of obfuscation techniques.

Keywords: obfuscation, Android, model checking, formal methods, malware

Email addresses: fabio.martinelli@iit.cnr.it (Fabio Martinelli), francesco.mercaldo@iit.cnr.it (Francesco Mercaldo), vnardone@unisannio.it (Vittoria Nardone), antonella.santone@unisannio.it (Antonella Santone), arunkumarsangaiah@gmail.com (Arun Kumar Sangaiah), cimitile@unisannio.it (Aniello Cimitile)
1. Introduction

Code obfuscation is a practice that makes programs hard to understand. It consists of applying transformation techniques to convert programs in order to hide their sensitive information while preserving their functionality. Code obfuscation can be used for several purposes. Originally, it has been used to protect intellectual property, i.e., to protect commercial software against reverse engineering and user sensitive and private information \[1\] \[2\] \[3\]. As cited in \[4\], obfuscation has been found to be the cheapest and easiest solution for protecting software and data \[5\] even if other methods exist such as encryption, server-side execution and native code. However, recently code obfuscation has been also used by malware writers to evade antimalware scanners \[6\]. In fact, obfuscations used by malware writers are generally utilized for the aim of changing the signature of the code \[7\] \[8\], i.e., the sequence of instructions so that a virus scanner will be unable to use search strings to detect the presence of a malware \[9\] \[10\] \[11\]. Metamorphic and polymorphic viruses are a consequence of the application of code obfuscation in order to alter the signature of the malware for each different propagation \[12\] \[13\]. Metamorphic malware is rewritten with each iteration so that each succeeding transmitted versions of the code are different from the preceding one \[14\]. The code changes makes it difficult for signature-based antimalware software in order to recognize that different propagations are related to the same malicious program \[15\]. Polymorphic malware also makes changes to code to avoid detection: it contains two parts, but one part remains the same with each iteration, which makes the malware less difficult to detect \[16\] \[17\].

In the end, both uses have the same basic intent: to obscure understanding of the cyber threats \[18\] \[19\]. Of course each task involves different limiting properties \[20\] \[21\] \[22\]. Malware need to deal with the fact that their obfuscations must continually change, or else scanners will have a known signature to match against \[23\]. This brings up time and space restrictions. Conversely, commercial software has performance requirements, which limit the types and
Two classes of obfuscating transformations can be distinguished. The first is called *surface obfuscation* and it focuses on obfuscating the syntax of a program, as for example changing variable names. The second is called *deep obfuscation* and it attempts to obfuscate the actual structure of a program, i.e., by changing its control flow. The first one makes it harder for a human to understand the source code, but it has no effect on algorithms used for reverse engineering. On the other hand, deep obfuscation affects the efficacy of tools for program analyses and reverse engineering, since it changes the actual structure of a program.

Model checking is a useful method to automatically verify the correctness of a system with respect to a desired behavior, by checking if a mathematical model of the system satisfies a formal specification of this behavior, expressed using a temporal logic. So far, it has been successfully used in many application domains such as biological systems, business processes, web service orchestration among others. Nevertheless, as far as we know, this is the first attempt to exploit a model checking technique for mining code obfuscation. We explore the possibility of identifying obfuscated code in platforms like Java and Android using model checking and Java Bytecode. We focus on detecting deep obfuscation, which requires reasoning about semantic aspects of the program and therefore it is more difficult than detecting surface obfuscation, which is syntactically based. Performing code obfuscation detection on Java Bytecode and not directly on the Java code has several advantages: (a) independence of the source programming language; (b) code obfuscation detection without decompilation even when source code is lacking; (c) ease of parsing a lower-level code; (d) more effectiveness than approaches at source code level.

We propose a novel formal based approach to mine code obfuscation. In this paper, we present a model checking based approach which is structured in three main steps. The first step aims at deriving formal models starting from the Java Bytecode. The second step aims to define the characteristic of the more pervasive obfuscating techniques by means of the construction of the temporal
logic properties. Finally, a formal verification environment, including a model checker, is invoked to mine obfuscated code. Our approach has applications to both software engineering and software security. In software engineering, we show how formal methods can be used to enhance reverse engineering, even for code that has been designed to be difficult to reverse engineer. For software security, our approach can help to identify malware codes even if they are obfuscated.

In order to evaluate our methodology we use a dataset of real-world Android applications, since software theft such as unauthorized duplication and plagiarism is also increasing fast. As demonstrated in [33], with increased incentives and low barriers to entry, plagiarists and clones have followed. To combat cloning, mobile markets need robust techniques to identify these clones, in order to defend the intellectual property of developers, that using obfuscation to try to make less understandable their code [12, 34]. There is a need to protect users from malicious applications [35, 36] and developers from plagiarists who wish to benefit from a legitimate developer’s hard work. But the point is that on Android, due to its open-source nature, reverse engineering is very easy and thus obfuscating techniques can be used to protect intellectual property of software. For this reason it is very important to develop a general approach to mine obfuscating transformations also expendable in Android environment. The main distinctive features of our approach are:

- we use model checking technique;
- we use Java Bytecode;
- we propose a static analysis, i.e., it is not required to run the code but only to analyze the code;
- we are able to localize the code fragment in which the obfuscating technique is applied;
- we validated our results using a dataset of real-world Android applications;
we identify the obfuscating technique with an average accuracy equal to 0.983.

The rest of the paper is organized as follows: Section 2 describes the related work, Section 3 provides background on virtual machines, obfuscating techniques and formal methods; Section 4 presents our methodology; Section 5 discusses the evaluation results and, finally, conclusion and future works are given in in Section 6.

2. Related Work

In last years several researchers focused on the identification of obfuscating techniques as we discuss below.

Authors in [32] propose an approach with the aim to measure software similarity between original and obfuscated programs in order to determine if the obfuscated version is an illegal copy of the original version. Basically they empirically analyze the code obfuscation effects on Android application similarity analysis. The measurements were done using five different Android applications with the DashO obfuscator. In the evaluation they show that similarity measures at Bytecode level are more effective than those at source code level in order to analyze software similarity from the point of view of the similarity metrics they computed. The obfuscating techniques applied are relative to layout, data, control and string encryption.

Xu et al. [37] performs a measurement study related to obfuscating techniques in Javascript code. They find that JavaScript obfuscation is a common practice among malicious JavaScript code in order to evade detection, realizing that JavaScript obfuscating techniques are dangerous threats to Web security because of the simplicity of applying such techniques on existing malicious codes and the lack of efficient and effective detection approaches.

Likarish et al. [38] consider a method for automatic detection of obfuscated JavaScript using a machine-learning approach. Using a dataset of regular and obfuscated samples from a content delivery network and the Alexa top 500 web-
sites, they demonstrate that it is possible to distinguish between obfuscated and not obfuscated scripts with precision and recall around 99%.

Lu et al. [39] describe an approach to deobfuscation of JavaScript code based on dynamic analysis and slicing. Experiments using a prototype implementation indicate that the technique is effective even against highly obfuscated programs. Guillot at al. [40] propose an approach based on a local semantic analysis, with the aim to rewrite the binary code in a simpler way. Their objective is no longer to seek and destroy known patterns, but to proceed to a complete, on-the-fly, optimized code rewriting.

Sharif et colleagues [41] present a PC malware obfuscating technique able to automatically conceal specific trigger-based behavior from these malware analyzers, i.e., the conditional code obfuscation. They automatically transform a program by encrypting code that is conditionally dependent on an input value with a key derived from the input and then removing the key from the program. In the evaluation, they demonstrate that on existing malware samples their method can hide a significant portion of trigger-based code.

Researchers in [42] starting from the consideration that when new malware is encountered, such obfuscations have to be penetrated or removed in order to understand the internal logic of the code and devise countermeasures, discuss a method for deobfuscation of obfuscated executable code. The proposed approach uses semantics-preserving program transformations to away obfuscation code using a Control Flow Graph (CFG) similarity algorithm. Authors in the evaluation consider samples obfuscated with garbage code techniques and call indirection. Our work differently consider also the widespread cyphering obfuscation techniques. In the obfuscation resilience evaluation they consider the garbage insertion transformations: our method, working at Bytecode level, exploits the optimization features of the compiler, able to eliminate the trivial garbage code injection: this is the reason why we do not consider garbage code injection in the techniques identifiable by our work.

BE-PUM (Binary Emulator for PUsdown Model generation), which generates a precise CFG, under presence of typical obfuscating techniques of malware,
e.g., indirect jump, self-modification, overlapping instructions, and structured exception handler (SEH), which cover packers, is proposed in [43]. This model is evaluated against 2000 real-world malware examples and the results are compared with those of popular commercial disassembler: the conclusion is that BE-PUM correctly traces CFGs whereas the commercial tools fail. Considering manual inspection on 300 malware examples, authors observe that the starts of these failures exactly locate the entries of obfuscation code.

Researchers in [44] provide a formal framework for code obfuscation based on abstract interpretation and program semantics. They consider code obfuscation through opaque predicate insertion and they show how the degree of abstraction needed to disclose different opaque predicates. The technique is evaluated using the constant propagation obfuscating technique i.e., a transformation that, knowing the values that are constant at a given program point on all possible executions of a program, propagates these constant values as far forward through the program as possible. Authors do not evaluate the technique on a real-world dataset.

Several works in the literature evaluated the effectiveness of existing mobile malware detection mechanisms through code obfuscation: following papers demonstrate that with obfuscating techniques the malware writer is able to fool signature-based detection i.e., malware correctly recognized as malicious after the injection of code obfuscation is not more identified by antimalware. In [45] authors present ADAM, an automated system for evaluating the detection of Android malware. Using ADAM, researchers apply a set of trivial obfuscating techniques to a dataset containing 222 malware samples. For each antimalware product, results show how each of the obfuscations can significantly reduce the detection rate. ADAM implements only a few transformations, such as: (i) renaming methods, (ii) introducing defunct methods, (iii) code reordering, and (iv) string encoding, in addition to repacking and assembling/disassembling.

Fraunhofer security experts test 11 antimalware solutions using 10 malware
samples and 10 altered ones. In the evaluation they show that 7 out of 10 antimalware were able to recognize all the sample belonging to malware dataset; while using the altered samples the malware identification decreases dramatically.

Researchers in [46] evaluate 10 antimalware tools using 6 original and transformed malware samples belonging to six different families. They conclude that all the antimalware products are susceptible to common evasion techniques. In [47] the authors demonstrated that using simple code transformations to existing malware that is well recognized by malware detectors turns it into a version that is no longer recognized by the most malware detectors. Several transformations are considered: Identifier Renaming, Data Encoding, Call Indirections, Code Reordering, Junk Code Insertion, Encrypting Payloads and Native Exploits.

As emerges from the discussion and at the best of authors knowledge, this is the first work with the aim to identify obfuscating techniques in Java and Android environment using model checking.

3. Background

In this section we provide, in order to better understand our methodology, background notions about Java and Android Virtual Machines, the code obfuscations considered in the work and model checking technique.

3.1. A peek under Virtual Machines

As we stated in the introduction our method is able to identify a set of widespread obfuscating techniques on Java Bytecode. To better understand how our method is able to identify obfuscation in Java and Android environment, we provide details about the Virtual Machine, i.e., the execution environment of Java and Android and about the Bytecode, i.e., the code executed by Virtual
Machines 48.
Furthermore we provide details about the two main implementations of Virtual Machines (VM): stack- and register-based.

Basically a VM is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer 49. Their implementations may involve specialized hardware, software, or a combination thereof.

The need to ensure a rapid boot and response time from the applications that run on hardware with limited resources has prompted Google to design a software that can make the most of limited resources available from a mobile device: the answer was the Dalvik Virtual Machine (DVM). The DVM is a virtual machine designed by Dan Bornstein, a Google employee, and takes its name from the fishing village whose Bornstein’s family is from.

The DVM is designed and optimized to exploit the small memory available on mobile devices, allowing multiple instances of the virtual machine to simultaneously run concealing thread management and memory to the operating system 50: in this way it is simplified to reuse the code and to eliminate or at least to prevent incompatibility issues.

Usually software engineers when designing a virtual machine, preferred a stack-based architecture (as the Java Virtual Machine, JVM), piling up all instructions within the stack.

In this way it is very simple to implement both the Virtual Machine and the relative back-end compiler 51.

Literature demonstrated that a stack-based approach for Virtual Machine is not efficient: as a matter of fact, comparing stack-based approach with register one it turns out that the second requires 32.3% less time and 47% less instructions than the first one 52.

In order to better understand this difference of performances, consider the following expression:
\[ b = c + d \]
A stack based machine will translate this expression in the following code (com-
posed by four different instructions) for the JVM:

\texttt{ILOAD d, ILOAD c, IADD, ISTORE b}

While using a register-based VM the same expression will be translated in one instruction:

\texttt{IADD a,b,c}

As explained, the code used from register-based architecture is significantly reduced if compared to that used by a stack-based Virtual Machine although, on average, the code involves a real machine for more than 1.07% of work for each instruction because of a greater length; the number of instructions to be processed, then, explains why the first turns out to be much more faster than the second. For this reason Google decided to base the Dalvik Virtual Machine on an register-based architecture: processing power and memory limits, in fact, are very respected constraints by the DVM implementation\textsuperscript{[53]}. In this paper we propose a method that is able to work with code targeting both the platforms, since, as explained above, the Java Virtual Machine is able to interpret Java Bytecode, while using well-know free available open source tools we are able to convert the Dalvik executable file into Java Bytecode.

\subsection{Obfuscation}

Code obfuscation was invented in order to hide the implementation from a non-desired analysis or reverse engineering. At the beginning, obfuscation was used for protecting intellectual property, secrets contained in the code, or for placing hidden functions in the code. By the late 1980s, obfuscation was adopted by malware writers for hindering analysis and detection \textsuperscript{[54]}. Polymorphic engines are a clear example of this usage, since they generate differently looking versions of the same malware to evade signature-based detection. In this paper we identify the following obfuscating techniques:

- \textbf{Code reordering}: the aim of this transformation is to modify the order of instructions in the methods. A random reordering of instructions has been accomplished by \texttt{goto} instructions inserting. Obviously, the trans-
formations preserve the original runtime execution trace, i.e., the business logic of the applications is not modified by transformations.

- **Call Indirections**: several detection signatures can consider the use of the call graph application. In order to evade such signatures this transformation is able to alter the original call graph, by modifying every method invocation in the smali code through a call to a new method inserted by the transformation with the only responsibility to invoke the original method without adding other instructions. This technique is able to alter signatures based on the call graph of the applications making the identification of altered applications no longer possible.

- **Data Encryption**: The Dalvik Executable (dex) file of Android applications contains the strings and arrays used in the program. Strings can be considered to create detection signatures with the aim to identify malware by several antimalware. In order to elude such signatures, this transformation encodes strings with a Caesar cipher. The original string will be restored at application run-time, with a call to a method that knows the Caesar key, i.e., this transformation will encode each strings into the application and will embed the code to decrypt them at run-time.

In Appendix 6 we show some examples of the above obfuscating techniques adopted by two tools, DroidChameleon [55] and Carnival, an obfuscation engine proposed in [47].

In order to obfuscate the code we use two obfuscating tool: DroidChameleon [55] and Carnival, an obfuscation engine proposed in [47]. They adopt different implementations of the obfuscating techniques discussed above. Both tools are able to statically apply the set of transformations to the smali representation in an automated way. To apply the transformation, the tools extract the dex file of the application which is totally unreadable; for this reason both the tool convert this dex files to a more understandable form represented by the smali

[http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html](http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html)
language. The smali syntax is loosely based on Jasmin’s/dedexer’s syntax\footnote{http://jasmin.sourceforge.net/} and supports the full functionality of the dex format (annotations, debug info, line info) and Dalvik opcodes.

### 3.3. Model checking

The basic concepts of the model checking are introduced. To apply model checking methods, we need:

1. A Formal Specification Language;
2. A Temporal Logic;
3. A Formal Verification Environment.

#### A Formal Specification Language

A detailed notation for specify systems is necessary. For this purpose, the Milner’s Calculus of Communicating Systems (CCS) \cite{56} is used. CCS is one of the best-known process algebras. Process algebras have proposed as formal descriptions of complex software systems, with particular emphasis to parallel and distributed systems. The algebraic structure of the concurrent processes is one of the most important issue in the definition of a Process Algebra. This uses a labelled transitions approach, where states (resp. transitions) correspond to processes (resp. actions). In the following, a brief overview of the main features of CCS is reported. Readers unfamiliar with CCS are referred to \cite{57, 56} for further details. The syntax of processes is the following:

\[
p ::= \text{nil} \quad \text{nil} \\
| \quad \alpha.p \quad \text{prefix} \\
| \quad p + p \quad \text{summation} \\
| \quad p|p \quad \text{composition} \\
| \quad p\backslash L \quad \text{restriction} \\
| \quad p[f] \quad \text{relabeling} \\
| \quad x \quad \text{constant}
\]
where

- \( \alpha \) ranges over a finite set of actions \( \mathcal{A} = \{ \tau, a, \bar{a}, b, \bar{b}, \ldots \} \). Input actions are labelled with “non-barred” names, i.e., \( a \), while output actions are “barred”, i.e., \( \bar{a} \). The action \( \tau \in \mathcal{A} \) is called internal action. \( \tau \) actions represent some level of abstraction in the characterization of the processes, since they can hide an arbitrarily complex sequence of actions whose details are kept private. The set \( \mathcal{V} = \mathcal{A} - \{ \tau \} \) is the set of visible actions. Each action \( l \in \mathcal{V} \) (resp. \( \overline{l} \in \mathcal{V} \)) has a complementary action \( \overline{l} \) (resp. \( l \)). Processes may communicate when a process is ready to execute some action \( l \) and the other one is ready to execute the complementary action \( \overline{l} \).

- In processes like \( p \setminus L \), the set \( L \) is a set of actions such that \( L \subseteq \mathcal{V} \).

- In processes like \( p[f] \), the relabelling function \( f \) is a total function, \( f : \mathcal{A} \rightarrow \mathcal{A} \), where the constraint \( f(\tau) = \tau \) is satisfied.

- Every constant is defined by a constant definition. For example, \( x \overset{\text{def}}{=} p \), where \( x \) ranges over a set of constant names and \( p \) is called the body of \( x \).

The standard operational semantics \cite{milner1989} is given by a relation \( \rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P} \), where \( \mathcal{P} \) represents the set of processes. We give the semantics for CCS by induction over the structure of processes.

- \textit{Nil}. \( \text{nil} \) represents a process that can do nothing. \( \text{nil} \) cannot evolve, thus no rule exists for \( \text{nil} \).

- \textit{Prefix}. The process \( \alpha.p \) can perform the action \( \alpha \) and then become the process \( p \). This is encoded by the rule \textbf{Act}:

\[
\begin{array}{c}
\textbf{Act} \\
\alpha.p \overset{\alpha}{\rightarrow} p
\end{array}
\]

- \textit{Summation}. The process \( p + q \) is a process that non-deterministically behaves either as \( p \) or as \( q \). This is expressed by the rules \textbf{Sum} \( \mathbf{1} \) and
Sum$_1$: \[ p \xrightarrow{\alpha} p' \]
\[
\begin{array}{c}
p + q \xrightarrow{\alpha} p' \\
p + q \xrightarrow{\alpha} q' \\
\end{array}
\]
Sum$_2$: \[ q \xrightarrow{\alpha} q' \]
\[
\begin{array}{c}
p + q \xrightarrow{\alpha} p' \\
p + q \xrightarrow{\alpha} q' \\
\end{array}
\]

- **Composition.** The parallel composition is encoded by the operator $|$. $p$ and $q$ may run independently: if the process $p$ can execute the action $\alpha$ and become $p'$, then $p|q$ can execute the action $\alpha$ and become $p'|q$. The same holds for $q$. This is stated by the rules Par$_1$ and Par$_2$:

\[
\begin{array}{c}
\text{Par}_1 \quad p \xrightarrow{\alpha} p' \\
\hline
p|q \xrightarrow{\alpha} p'|q \\
\end{array}
\]
\[
\begin{array}{c}
\text{Par}_2 \quad q \xrightarrow{\alpha} q' \\
\hline
p|q \xrightarrow{\alpha} p|q' \\
\end{array}
\]

Furthermore, $p$ and $q$ may also be involved in a communication when a handshake occurs, i.e., both processes are simultaneously able to execute the complementary actions. Handshake corresponds to an internal communication by means of the action $\tau$; in this way the two processes exchange information between them. More precisely, if $p$ can execute an input visible action $l$ and become $p'$, and $q$ can execute the complementary output action $\overline{l}$ and become $q'$, then $p|q$ communicate performing $\tau$ and become $p'|q'$. This is encoded by the rule Com:

\[
\text{Com} \quad p \xrightarrow{l} p', \quad q \xrightarrow{\overline{l}} q' \\
\hline
p|q \xrightarrow{\tau} p'|q' \\
\]

- **Restriction.** Suppose $L \subseteq V$, $p|L$ is a process that behaves as $p$ except that it cannot execute externally any of the actions belonging to $L$ (as well as the corresponding complementary actions), even if each pair of these complementary actions can internally be executed for the communication. This is encoded by the rule Res:

\[
\text{Res} \quad p \xrightarrow{\alpha} p' \\
\hline
p|L \xrightarrow{\alpha} p'|L, \quad \alpha, \overline{\alpha} \not\in L
\]
• **Relabelling.** The operator \([f]\) represents the relabelling of actions. If \(p\) can execute \(\alpha\) and become \(p'\), then \(p[f]\) can execute \(f(\alpha)\) and become \(p'[f]\). This is encoded by the rule \textbf{Rel}:

\[
\text{Rel} \quad \frac{p \xrightarrow{\alpha} p'}{p'[f] \xrightarrow{f(\alpha)} p'[f]}
\]

• **Constant definition.** Suppose \(x \overset{\text{def}}{=} p\). The behavior of the constant \(x\) is the same of the behavior of the process \(p\) as encoded by the rule \textbf{Con}:

\[
\text{Con} \quad \frac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p', \overset{\text{def}}{=} p}
\]

A *(labeled) transition system* is a quadruple \(T = (S, A, \rightarrow, s)\), where \(S\) is a set of states, \(A\) is a set of transition labels (actions), \(s \in S\) is the initial state, and \(\rightarrow \subseteq S \times A \times S\) is the transition relation. If \((s, \alpha, s') \in \rightarrow\), we write \(s \xrightarrow{\alpha} s'\). If \(\delta \in A^*\) and \(\delta = \alpha_1 \ldots \alpha_n, n \geq 1\), \(s \xrightarrow{\delta} s'\) means \(s \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s'\). Furthermore, \(s \xrightarrow{\lambda} s\), where \(\lambda\) is the empty sequence. Suppose \(s \in S\), the set of the states reachable from \(s\) by \(\rightarrow\) is denoted by \(\mathcal{R}(p) = \{s' \mid s \xrightarrow{\delta} s'\}\). The *standard transition system* for a CCS process \(p\) is \(S(p) = (\mathcal{R}(p), A, \rightarrow, p)\). It is worth noting that, with abuse of notation, \(\rightarrow\) is used for denoting both the operational semantics and the transition relation among the states of the transition system.

**A Temporal Logic**

A detailed notation for defining properties is necessary. A temporal logic \[58\] solves this need. A temporal logic allows to state in a formal way that, for instance, all scenarios will satisfy some property at every step, or that some particular action will eventually happen, and so on. The most considerable examples are:

• **Safety properties.** A safety property states that nothing bad happens.
- **Liveness properties.** A liveness property states that something good eventually happens.

The *mu-calculus* logic [58] is used in this paper. The syntax of the *mu-calculus* is reported below, where we suppose that $Z$ ranges over a set of variables, $K$ and $R$ range over sets of actions.

$$
\phi ::= \mathsf{tt} \mid \mathsf{ff} \mid Z \mid \phi \lor \phi \mid \phi \land \phi \mid [K] \phi \mid \langle K \rangle \phi \mid \nu Z. \phi \mid \mu Z. \phi
$$

The satisfaction of a formula $\phi$ by a state $s$ of a transition system, denoted by $s \models \phi$, is so defined:

- each state satisfies $\mathsf{tt}$ and no state satisfies $\mathsf{ff}$;
- a state satisfies $\phi_1 \lor \phi_2$ ($\phi_1 \land \phi_2$) if it satisfies $\phi_1$ or (and) $\phi_2$.
- $[K] \phi$ and $\langle K \rangle \phi$ are the modal operators:
  
  $[K] \phi$ is satisfied by a state which, for every performance of an action in $K$, evolves in a state obeying $\phi$.
  
  $\langle K \rangle \phi$ is satisfied by a state which can evolve to a state obeying $\phi$ by performing an action in $K$.

In Table 1 is reported the precise definition of the satisfaction of a closed formula $\varphi$ by a state $s$ (denoted $s \models \varphi$). $\mu Z. \phi$ and $\nu Z. \phi$ are the fixed point formulae, where $\mu Z$ ($\nu Z$) binds free occurrences of $Z$ in $\phi$. An occurrence of $Z$ is free if it is not within the scope of a binder $\mu Z$ ($\nu Z$). A formula is *closed* if it contains no free variables. $\mu Z. \phi$ is the least fix-point of the recursive equation $Z = \phi$, while $\nu Z. \phi$ is the greatest one. A transition system $T$ satisfies a formula $\phi$, denoted $T \models \phi$, if and only if $q \models \phi$, where $q$ is the initial state of $T$. A CCS process $p$ satisfies $\phi$ if $S(p) \models \phi$.

**A Formal Verification Environment**

Finally, an algorithm to verify if a complex system satisfies a property is necessary. Formal verification is a mathematical based process able to verify if
\begin{align*}
p \not\models & \quad \text{ff} \\
p \models & \quad \text{tt} \\
p \models \varphi \land \psi & \iff p \models \varphi \text{ and } p \models \psi \\
p \models \varphi \lor \psi & \iff p \models \varphi \text{ or } p \models \psi \\
p \models [K] \varphi & \iff \forall \alpha' \forall \alpha \in K. p \xrightarrow{\alpha} p' \text{ implies } p' \models \varphi \\
p \models (K) \varphi & \iff \exists \alpha' \exists \alpha \in K. p \xrightarrow{\alpha} p' \text{ and } p' \models \varphi \\
p \models \nu Z. \varphi & \iff p \models \nu Z^n. \varphi \text{ for all } n \\
p \models \mu Z. \varphi & \iff p \models \mu Z^n. \varphi \text{ for some } n
\end{align*}

where:

- for each \( n \), \( \nu Z^n. \varphi \) and \( \mu Z^n. \varphi \) are defined as:

\begin{align*}
\nu Z^0. \varphi & = \text{tt} \\
\mu Z^0. \varphi & = \text{ff} \\
\nu Z^{n+1}. \varphi & = \varphi [\nu Z^n. \varphi / Z] \\
\mu Z^{n+1}. \varphi & = \varphi [\mu Z^n. \varphi / Z]
\end{align*}

where the notation \( \varphi [\psi / Z] \) indicates the substitution of \( \psi \) for every free occurrence of the variable \( Z \) in \( \varphi \).

\begin{table}[h]
\centering
\caption{Satisfaction of a closed formula by a state}
\end{table}
Figure 1: Model Checker

a design satisfies some requirements. Many techniques for verification are provided, among them model checking. In the model checking the approach \cite{59} the requirements are encoded using a temporal logic formula. Each formula is validated on the system modeled as transition systems. The inputs of a model checker tool are both a transition system and a temporal formula. The output is equal to true if the system satisfies the formula and false otherwise (see Figure 1). An exhaustive state space search is performed by the node checker tool. The termination is guaranteed since the model is finite. The research challenge in model checking is in developing algorithms allowing the management of big search spaces.

One of the most popular environments for verifying concurrent systems is the Concurrency Workbench, Aalborg Edition (CAAL) \cite{60}, which supports several different specification languages, CCS among them. In the CAAL, the verification of temporal logic formulae is model checking based \cite{59}. We have chosen the CAAL workbench since it provides a unique combination of features that no other tool presently offers, such that: (i) CAAL supports not only model-checking but
also equivalence checking, which plays a crucial role for component-based sys-

tems and compositional verification; (ii) many methodologies \cite{61, 62, 63} have

been developed by one of the author of this paper, to reduce the state explo-
sion problem (typical of the model checking technique) using the CCS language

supported by CAAL; (iii) CAAL supports concurrency.

4. Methodology

Our model-checking-based methodology to identify obfuscating transforma-
tions is presented in this section. It is based on two main processes:

1. Creation of a formal model of Java Bytecode.

2. Translation of obfuscating transformations into temporal logic formulae.

The first process uses Milner’s Calculus of Communicating Systems (CCS) \cite{56}

group specification to the construction of the model. Starting from .class

files written in Java Bytecode we generate a CCS specification. Thus, if the
code has \( n \) .class files, we obtain \( n \) CCS processes. The transformation from
Java Bytecode to CCS is performed defining a translation function \( T \). It is de-
defined for each Java Bytecode instruction. This translation has been completely
automated and all the methods are analyzed separately. We use a model similar

\( T(i) = x_i \stackrel{\text{def}}{=} \text{gotoj} x_j \)

\( i \in \{0, \ldots, c\} \), and \( c[i] \) is the instruction
at address \( i \), where \( c \) denotes the length of \( c \).

\textbf{Instruction:} \( c[i] = \text{goto} j \)
The instruction $c[i] = \text{goto} j$ is encoded as CCS process $x_i$ that executes the action $\text{goto} j$ and then jumps to the instruction $j$, corresponding to the CCS process $x_j$.

**Instruction:** $c[i] = \text{tstore } z$

$$T(i) = x_i \overset{\text{def}}{=} \text{store}.x_{i+1}$$

Each $\text{tstore} z$ instruction is encoded as $\text{store}$ followed by the constant process $x_{i+1}$ representing the CCS translation of the successive instruction. This translation is obtained without including the type $t$ and the name of the variable $z$.

**Instruction:** $c[i] = \text{if } \text{cond } j$

$$T(i) = x_i \overset{\text{def}}{=} \text{if } \text{cond_{tt}}.x_{i+1} + \text{if } \text{cond_{tt}}.x_j$$

Conditional jumps are instead specified as non-deterministic choices. The true (resp. false) condition is represented by the CCS action $\text{if } \text{cond_{tt}}$ (resp. $\text{if } \text{cond_{ff}}$), while $\text{cond}$ belongs in the following set:

$$\{ \text{ne, eq, icmeq, icmpne, } ... \}.$$  

Thus, if the condition is true, the CCS process $x_i$ performs the action $\text{if } \text{cond_{tt}}$ and the execution branches to the process $x_j$ corresponding to the process at the address $j$. If the condition is false, the execution continues at the successive instruction at address $i + 1$ (process $x_{i+1}$).

An example of the Java Bytecode to CCS transformation $T$ is reported in Figure 2. In the right side of the Figure 2 the corresponding automaton of the CCS fragment is depicted. The CCS process and the corresponding automaton are related to the Listing 2 of Appendix 6.

Another example refers to the code in Listing 4 of Appendix 6 which has been obfuscated with the code reordering technique. Figure 3 shows a fragment of an automaton corresponding to a CCS process obtained by applying our Java Bytecode to CCS transformation to the code snippet of Listing 4.
The second process aims to investigate if a Java Bytecode is obfuscated. The main aim of our methodology is to verify if a code is obfuscated with specific techniques. Thus, we use mu-calculus logic, \cite{58} as a branching temporal logic, to express code reordering, call indirection and data encryption characteristics, the most widespread obfuscating techniques.

The CCS model obtained by the first process is used to verify properties. In our approach, we invoke the Concurrency Workbench, Aalborg Edition (CAAL) \cite{60} as formal verification environment to establish when the code under analysis is obfuscated (also indicating the specific technique). Let code be the analyzed Java Bytecode and let $\varphi_t$ be the formula expressing the specific obfuscating technique $t$. The approach we have defined consists in counting the number of CCS processes obtained by analyzing code which satisfy the formula $\varphi_t$. Suppose that this number is $n$. Note that we generate a CCS process for each .class file of a code. In order to establish if code is obfuscated with $t$, we have to define a threshold such that, whenever $n$ is greater than this threshold, we can establish that code could be obfuscated with $t$. The introduction of the threshold stems
from the need to differentiate a normal programming code from an obfuscated one. As known from literature \cite{66}, programmers should avoid the creation of “spaghetti code”, i.e., unstructured programming with \texttt{goto} statement even if they could use it. Furthermore, a Java programmer could implement a method whose only task is to call another method, even if, as stated in \cite{67}, those methods are considered as code smells and they are denoted as \textit{lazy methods}. These are the reasons why we introduce a threshold in our methodology. A low value of $n$ is not representative of an obfuscated code, but it surely is representative of a bad programming practice, on the other hand a big value of $n$ could mean that the code has been obfuscated.
Some considerations

We highlight that our methodology is able to localize the code fragment in which the obfuscating technique is applied, since we know the CCS process (and therefore the .class) that satisfies the formula representing the obfuscating technique.

The logic formulae we stated are able to identify obfuscating techniques both on Java and on Android environments: this is possible because we analyze the Bytecode of the application under analysis.

Considering the Bytecode, a common element to Android and to Java programming languages, we are able to provide the model and the formulae working for both programming languages.

Figure 4 shows the steps of the compilation of an Android (and Java) application. The steps related to Java source code, Java compiler and Bytecode are the same involved in the compilation of a Java application. The difference is in the execution environment: in the case of Java, the Bytecode is directly executed in the JVM (a stack-based virtual machine), while in the case of Android, the Bytecode needs to be transformed into Dalvik Bytecode in order to be run on DVM (a register-based virtual machine).

This is done in order to ensure the compatibility (when possible) between Java
and Android, and considering the multiplicity of mobile applications developed in Java, it was necessary for Google to create a conversion system able to ensure portability for applications running different versions of Android operating system.

For this reason, by analyzing the Bytecode, we are able to identify obfuscating techniques in both programming languages.

As discussed in the background section, the JVM is stack-based, thus it is necessary to convert the Bytecode able to run on JVM in dex file (i.e., Dalvik Executable file able to run on DVM). To perform this operation, it is necessary to invoke the dex compiler, a tool able to transform the Java Bytecode in the Dalvik Bytecode able to execute on the register-based DVM with limited resources.

4.1. Logic formulae expressing obfuscating transformations

In this section we express code obfuscating transformation techniques in the mu-calculus logic.

**Code reordering:** code reordering modifies the order of the instructions in the methods of a program. This transformation is performed by reordering the instructions and inserting goto instructions. The behavior of the program is preserved.

The logic formula expressing code reordering is:

\[ \varphi_1 = \mu X. (\text{goto}) (\text{goto}) \top \lor (\neg \text{goto}) X \]

More formally, the meaning of \( \varphi_1 \) is: “it is possible to reach a state where 2 goto actions are consecutively performed, i.e., without performing other actions in between them”.

Formula \( \varphi_1 \) is true on the automaton shown in Figure 3. In fact, starting from state L12 in Figure 3 and following the path, the reader can find at least 2 goto instructions consecutively. The number near every goto indicates the number of consecutive goto instructions encountered along a path.
On the other hand the formula $\phi_1$ is false on the CCS process shown in Figure 2, since no two consecutive goto instructions exist. As explained above the automaton in Figure 2 (resp. Figure 3) is the automaton obtained from the original code $c$ (resp. from the code $c$ obfuscated with the code reordering technique).

**Call Indirections:** this transformation acts on the call graph with trivial modifications. More precisely, if the method $x$ calls another method, namely $y$, this invocation is substituted with the invocation of a non-existing method $z$ that only performs the original invocation of $y$.

The logic formula expressing call indirections is:

$$\phi_2 = \mu X. (load) (invoke) (return) \top \lor (\neg load) \top$$

More formally, the meaning of $\phi_2$ is: “it is possible to reach a state where the actions: load, invoke and return are consecutively performed, i.e., without performing other actions in between them”. This means that after an invoke only a return is performed meaning that the method invokes a previously non-existing method that then calls the method in the original call, without doing anything.

**Data Encryption:** the dex files contain all the strings and array data that have been used in the code. These strings and arrays may be used to develop signatures against malware. These can be kept in encrypted form.

The logic formula expressing data encryption is:

$$\phi_3 = \mu X. (invokeapplyCaesar) \top \lor (\neg invokeapplyCaesar) \top$$

More formally, the meaning of $\phi_3$ is: “it is possible to reach a state where invokeapplyCaesar is performed.

We consider in the evaluation the logic formula $\phi_3$ able to catch the Caesar cipher with a key equal to three: each letter in the plain text is replaced by a letter some fixed number of positions (i.e., the key) down the alphabet that in our evaluation is equal to three.
Table 2: Number of applications transformed for the item (I) of the dataset.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Code Reordering</th>
<th>Call Indirection</th>
<th>Data Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>DroidChameleon</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Carnival</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>

The tools we employ to obfuscate Android applications are able only to inject the Caesar cypher with a key equal to three, but our method is generic, for this reason it is able to catch all substitution cipher algorithms involved to perform code obfuscations.

5. Evaluation

In this section we present the result of our method on a real-world dataset of Android applications. We measure the performance of our method using the metrics of Precision, Recall, F-measure and Accuracy.

5.1. Dataset

In our evaluation we use two tools implementing the obfuscating techniques, as discussed in Section 3.2: DroidChameleon and Carnival. We evaluated obfuscated and not obfuscated applications in order to demonstrate the effectiveness of our methodology. We composed the dataset as following:

1. we retrieved 200 real-world Android applications and we applied to each application of the dataset a single obfuscating technique, as Table 2 shows. Since we use two tools and three obfuscating techniques, we obtain a total of 1400 applications, including the original ones;

2. we retrieved 300 real-world Android applications and we applied to each application of the dataset simultaneously all three obfuscating techniques provided by the two tools, for a total of 600 applications.
Table 3: Available categories on GooglePlay.

<table>
<thead>
<tr>
<th>Category</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Books &amp; Reference</td>
<td>Lifestyle</td>
</tr>
<tr>
<td>Business</td>
<td>Live Wallpaper</td>
</tr>
<tr>
<td>Comics</td>
<td>Media &amp; Video</td>
</tr>
<tr>
<td>Communication</td>
<td>Medical</td>
</tr>
<tr>
<td>Education</td>
<td>Music &amp; Audio</td>
</tr>
<tr>
<td>Finance</td>
<td>News &amp; Magazines</td>
</tr>
<tr>
<td>Games</td>
<td>Personalization</td>
</tr>
<tr>
<td>Health &amp; Fitness</td>
<td>Photography</td>
</tr>
<tr>
<td>Libraries &amp; Demo</td>
<td>Productivity</td>
</tr>
<tr>
<td>Shopping</td>
<td>Social</td>
</tr>
<tr>
<td>Sport</td>
<td>Tools</td>
</tr>
<tr>
<td>Travel &amp; Local</td>
<td>Transportation</td>
</tr>
<tr>
<td>Weather</td>
<td>Widgets</td>
</tr>
</tbody>
</table>

The total number of evaluated applications is 2000.

Note that the obfuscation is employed in both legitimate and malware applications. In fact, we examine, regarding the item (1), 100 malware and 100 trusted applications, while regarding the item (2) we consider 150 malware and 150 trusted applications.

The malware real-world samples used in the evaluation were gathered from the Drebin project’s dataset [68, 69]: a very well-known collection of malware used in many scientific works, which includes the most widespread Android families. The malware dataset is also partitioned according to the malware family: each family contains samples which have in common several characteristics, like payload installation, the kind of attack and events that trigger malicious payload [70].

In order to download trusted applications we crawled Google’s official app store using an open-source crawler. The obtained trusted dataset includes samples belonging to all the different categories available on the market. Table 3 shows the belonging categories of the legitimate samples we downloaded. The legi-

---

https://play.google.com/store
https://github.com/liato/android-market-api-py
mate applications were collected between January 2016 and April 2016. We analyzed the dataset with the VirusTotal service\(^6\), a service able to run 57 different antimalware software (e.g., Symantec, Avast, Kasperky, McAfee, Panda, and others): the analysis confirmed that the legitimate applications did not contain malicious payload while the malware ones were really malicious.

5.2. Threshold

As stated in Section 4, our methodology is based on the introduction of a threshold to distinguish a not obfuscated code from an obfuscated one. In order to define the threshold we have used 20 not obfuscated applications. These applications have been identified as not obfuscated through a manual inspection process. The process to determine the threshold and to establish if an application is to be considered obfuscated is has been carried out as follows:

1. Let us consider the above 20 applications. For each obfuscating technique \( t \) examined in the paper, we calculate the rate between the number of the CCS processes resulting obfuscated, i.e., that verify \( \varphi_t \), the formula representing \( t \), and the total number of the CCS processes of the application. We call this value index.

2. For each \( t \), we set the threshold as the maximum value of the index obtained in the previous step. Clearly, we have 3 different thresholds, one for every obfuscating technique.

3. We produce an obfuscated version of these applications.

4. We test the obfuscated applications and we compute again the new indexes for the obfuscated applications.

5. We compare the computed indexes with the thresholds defined at point 2. Whenever the computed index was greater than the threshold, then the analyzed application is considered obfuscated.

\(^6\)https://www.virustotal.com/
Table 4: Threshold for each Obfuscating Technique.

<table>
<thead>
<tr>
<th>Code Reordering</th>
<th>Call Indirection</th>
<th>Data Encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold</td>
<td>2%</td>
<td>13%</td>
</tr>
</tbody>
</table>

Table 5: Performance Evaluation

<table>
<thead>
<tr>
<th>Formula $\varphi$</th>
<th>Tool</th>
<th>Original</th>
<th>Samples Obfuscated with:</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>TN</th>
<th>PR</th>
<th>RC</th>
<th>Fm</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CR</td>
<td>CI</td>
<td>DE</td>
<td>All</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>Chameleon</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
<td>499</td>
<td>0</td>
<td>1</td>
<td>600</td>
<td>0.99</td>
</tr>
<tr>
<td>Carnival</td>
<td></td>
<td></td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
<td>488</td>
<td>4</td>
<td>12</td>
<td>596</td>
<td>0.99</td>
</tr>
<tr>
<td>CI</td>
<td>Chameleon</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
<td>495</td>
<td>8</td>
<td>5</td>
<td>592</td>
<td>0.98</td>
</tr>
<tr>
<td>Carnival</td>
<td></td>
<td></td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
<td>481</td>
<td>8</td>
<td>9</td>
<td>592</td>
<td>0.98</td>
</tr>
<tr>
<td>DE</td>
<td>Chameleon</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
<td>495</td>
<td>0</td>
<td>5</td>
<td>600</td>
<td>1.00</td>
</tr>
<tr>
<td>Carnival</td>
<td></td>
<td></td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>300</td>
<td>483</td>
<td>0</td>
<td>17</td>
<td>600</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Note: CR, CI and DE indicate Code Reordering, Call Indirection and Data Encryption obfuscating techniques, respectively.

Table 4 shows the thresholds resulting from the above process. They have been used in our methodology to verify if an application is obfuscated.

5.3. Results

Table 5 shows the results of the experiment applying our methodology using both DroidChameleon and Carnival. It is organized as follow:

- Column **Formula $\varphi$**: It indicates the formula $\varphi$ tested on the samples.
  There are three different formulae $\varphi$, one for each obfuscation technique.

- Column **Original**: It contains the number of not obfuscated samples used in our evaluation.

- Column **Sample Obfuscated with**: It contains the number of obfuscated samples which are obfuscated as follows:
  - Sub-Column **CR**: It contains the number of samples that are obfuscated with the Code Reordering technique.
  - Sub-Column **CI**: It contains the number of samples that are obfuscated with the Call Indirection technique.
– Sub-Column **DE**: It contains the number of samples that are obfuscated with the Data Encryption technique.

– Sub-Column **All**: It contains the number of samples that are obfuscated with all the three obfuscating techniques.

Clearly, all the samples are tested with the three formulae that compare in the row of the table.

- **Column TP**: It indicates the number of **True Positive**. A sample is considered as a TP if it is obfuscated with a particular technique and our methodology correctly identifies the applied obfuscating technique. We recall that for our methodology a sample is obfuscated with a particular obfuscating technique if the number of files that verify the formula is greater than a threshold as explained in Section 4.

- **Column FP**: It indicates the number of **False Positive**. A sample is considered as a FP if our methodology identifies the sample as obfuscated with a particular obfuscating technique but it is not obfuscated with that technique, i.e., a sample without any obfuscation or a sample obfuscated with other obfuscating techniques.

- **Column FN**: It indicates the number of **False Negative**. A sample is considered as a FN if our methodology not identifies the sample as obfuscated with a particular obfuscating technique but it is obfuscated with that technique, i.e., a sample obfuscated with the same technique indicated by the formula.

- **Column TN**: It indicates the number of **True Negative**. A sample is considered as a TN if our methodology not identifies the sample as obfuscated with a particular obfuscating technique and it is not obfuscated with that technique, i.e., a sample without any obfuscation or a sample obfuscated with other obfuscating techniques.
Table 6: apk size, dex size, jar size, uncompressed classed extracted by the jar file (identified by the PRJ column), States, Transitions and Execution Time for six applications with size ranging between 3.0 MB and 34.6 MB.

<table>
<thead>
<tr>
<th>App</th>
<th>Size</th>
<th>States</th>
<th>Transitions</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>apk</td>
<td>dex</td>
<td>jar</td>
<td>PRJ</td>
</tr>
<tr>
<td>APP-1</td>
<td>3.0 MB</td>
<td>4.5 MB</td>
<td>890.2 kB</td>
<td>92548</td>
</tr>
<tr>
<td>APP-2</td>
<td>5.1 MB</td>
<td>6.9 MB</td>
<td>35.8 kB</td>
<td>2335</td>
</tr>
<tr>
<td>APP-3</td>
<td>7.3 MB</td>
<td>2.6 MB</td>
<td>778.6 kB</td>
<td>71574</td>
</tr>
<tr>
<td>APP-4</td>
<td>9.9 MB</td>
<td>174.6 kB</td>
<td>261.3 kB</td>
<td>25747</td>
</tr>
<tr>
<td>APP-5</td>
<td>14.1 MB</td>
<td>6.2 MB</td>
<td>350.1 kB</td>
<td>31272</td>
</tr>
<tr>
<td>APP-6</td>
<td>34.6 MB</td>
<td>9.2 MB</td>
<td>37.4 kB</td>
<td>1235</td>
</tr>
</tbody>
</table>

• Columns PR, RC, Fm and Acc indicate the values of Precision, Recall, F-measure and Accuracy respectively, as defined in the following:

\[
PR = \frac{TP}{TP+FP}; \quad RC = \frac{TP}{TP+FN};
\]
\[
Fm = \frac{2PR \cdot RC}{PR + RC}; \quad Acc = \frac{TP+TN}{TP+FN+FP+TN}
\]

Results in Table 6 seem to be very promising: we obtain an Accuracy, Precision, Recall and F-Measure never less than 96%. Our logic rules correctly identify the applied obfuscating technique. The results show that we are able to identify the precise obfuscating technique also when the sample under exam is obfuscated with all the considered techniques simultaneously. In order to evaluate the time performances we conduct a deep analysis with the aim to understand whether some factor app-related can affect the execution time. We chose different size applications (from 3.0 MB to 34.6 MB) in order to highlight how the execution time varies when application of different size are considered.

Table 6 shows the size in MegaBytes (MB) related to the apk, the dex (expressed in MegaBytes), the jar the size of the uncompressed jar expressed in KiloByte (KB) (i.e., the application classes), the states, the transitions and the execution time (expressed in seconds) related to six different applications we analyzed in the evaluation. Analyzing the table we point out, as expected, that there is a correlation among the execution time, states and transitions in the PRJ analysis (i.e., the classes extracted from the jar file). The extraction of the jar
Figure 5: Model states and execution time (expressed in seconds) related to the model verification.

Figure 6: PRJ folder sizes and model states related to the model verification.
from the dex file has been obtained using the dex2jar tool (employed also in our methodology as previously explained) in order to obtain the classes in an uncompressed format. For this reason we report in the Table also the jar files size and the sizes of the folder extracted by the jar folder containing the classes (reported in the PRJ column). As a matter of fact, when the classes size is equal to 446.0 KB the execution time is equal to 0.822 s with 87758 states and 92348 transitions, while in case the jar file is equal to 37.4 KB our tool employs 0.010 s to perform the analysis with 1235 states and 1288 transitions. The same consideration can be deducted for the other apks considered in our evaluation.

Figure shows the relationship between the model states and the execution time in order to verify the model, while Figure shows the model states with the PRJ folders. As Figure exhibits, the time to verify the model is increasing when the model states are growing, while Figure shows the states trend when the classes file of the application under analysis in increasing: the formal models obtained applied our methodology grow linearly with the dimension of the extracted jar. From this analysis we conclude that the dimension of the classes of the application to evaluate (i.e., the classes of the application) influences the number of the states of the model, and the increasing number of states will affect the execution time related to model verification.

On the other hand, we do not rightly find any correlation analyzing the apk size and the execution time. In fact, the execution time is ranging between 0.010 and 0.822 seconds and it is not affected by the apk size (that is ranging between 3.0 MB and 34.6 MB), see for example APP-6 which is the bigger app but it exhibits the lower execution time. These results are expected, considering that the apk file contains not only the classes of the application but also the resources, as images and sounds useful to enjoy the mobile application from an user experience point of view. Moreover, also no correlation is find between the dex files (containing the classes of the application) and the execution time: the dex file is an optimized format targeting the Dalvik virtual machine.
6. Conclusion and Future Work

In this paper a model checking based approach able to identify the most widespread obfuscating techniques has been proposed. Usually, code obfuscation has been used to protect intellectual property, however recently, it has been also used to evade antivirus scanners. Nowadays Android malware detectors are able to identify a malware using its signature, typically extracted from manual inspection. The signature is based on code structure. It is very easy to evade the mechanism of signature recognition by trivial alteration of code structure, i.e., code obfuscation. Thus, we test our method on a real-world dataset of Android applications for three reasons: (1) Android application development is very close to Java; (2) to the best of our knowledge, no experimentation has been made on Android; (3) code obfuscation detection is very important to make the current anti-malware technologies able to recognize malware variants to limit the diffusion of zero-day attacks. The main distinctive features of our approach are: (i) we use model checking technique; (ii) we use Java Bytecode; (iii) we propose a static analysis, i.e., it is not required to run the code but only to analyze the code; (iv) we are able to localize the code fragment in which the obfuscating technique is applied; (v) we identify the obfuscating technique with an average accuracy equal to 0.983.

As future work we plan to extend the proposed method also to the identification of obfuscation techniques at native code and JavaScript code level. Moreover, we are able to identify whether the reflection API is invoked but, since our method is purely static, we are not able to identify the target of the refection. To manage this situation, as future work we plan to integrate our method with a dynamic one.

Acknowledgements

This work was partially supported by the H2020 EU funded project NeCS [GA #675320], by the H2020 EU funded project C3ISP [GA #700294].
References


39


Appendix

Examples of Code Obfuscations

In this section we show the resulting code after the application of some obfuscating techniques. For further information about the techniques we applied the reader can refer to Section 3.2. As we previously explained, the evaluation was conducted on real-world Android samples, and the code snippet we show in this section belongs to a real application. Listing 1 shows the Java source code related to the `createFromShortcut` method. In addition, we show the corresponding Java Bytecode, because our method, through the BCEL library, extracts Bytecode from the compiled application that is used to produce the formal CCS model. The Bytecode corresponding to the Java source code of Listing 1 is shown in Listing 2. Listing 4 reports the obfuscated Java Bytecode obtained by applying the Code Reordering technique. It is worth noting that in the obfuscated Java Bytecode there are many `goto` instructions. Although the normal flow of the code is preserved, there exist a lot of unconditional jumps that make the reconstruction of the normal flow of the code difficult. The newly inserted `goto` instructions produce “spaghetti code”, i.e., understandable code. Thus, modifying the control flow graph is important to evade the signature provided by antimalware. Listing 3 shows the Java source code related to Java Bytecode in Listing 2. This source code is obtained using a Java/Android reverse engineering environment. We highlight that the reconstruction of the source code, in the case of Code Reordering obfuscation, is difficult and in many cases is not a correct reconstruction.

Listing 3 and Listing 6 show the Java source code and the Java Bytecode respectively obtained by applying the Data Encryption obfuscation transformation. This obfuscating technique encrypts all strings present in the code. In the considered code there is a string as argument of the `println` method. The original

\footnote{http://commons.apache.org/bcel/}
public static void createShortcut(Context paramContext)
{
    System.out.println("createShortcut()");
    if (paramContext.getSharedPreferences(Constants.SETTINGS, 1).
        getBoolean(Constants.FIRST_START, true)) {
        createShortcut(paramContext, "Search", 17301583, "http://m
−001.net/i/");
    }
}

Listing 1: Example of Java Source Code.

string is “createShortcut()” (see Listing 1) while the obfuscated version is
“dsfbufTipsudvu)*” (see Listing 5). In order to preserve the original behavior
of the application the encrypted string is decrypted using a method performing
the decryption algorithm. As shown in Listing 5 the argument of the println
method is replaced by EncryptString.applyCaesar("dsfbufTipsudvu)*")
method call.

Finally, we show an example of Call Indirection application. Listing 7 and
Listing 8 report the codes obtained after obfuscating with the Call Indirection
technique. This type obfuscation changes the call graph of the code replacing
any method call with a non-existing method call. This new method contains the
original call. As shown in Listing 7 the old call to println method is replaced
with the call to method3 method. It is worth noting that in the method3 spec-
ification there is the old invocation of the println method. The new method
implements only the original invocation of the replaced method.
public static createShortcut(android.content.Context arg0) { //(
    Landroid/content/Context;)V
    getstatic java/lang/System.out:java.io.PrintStream
    ldc "createShortcut()" (java.lang.String)
    invokevirtual java/io/PrintStream println((Ljava/lang/String;)V);
    aload0
    getstatic com/alpha/Constants.SETTINGS:java.lang.String
    iconst_1
    invokevirtual android/content/Context getSharedPreferences((Ljava/lang/String;I)Landroid/content/SharedPreferences;);
    getstatic com/alpha/Constants.FIRST_START:java.lang.String
    iconst_1
    invokeinterface android/content/SharedPreferences getBoolean((Ljava/lang/String;Z)Z);
    ifeq L2
    aload0
    ldc "Search" (java.lang.String)
    ldc 17301583 (java.lang.Integer)
    ldc "http://m-001.net/i/" (java.lang.String)
    invokevirtual com/alpha/Alpha createShortcut((Landroid/content/Context;Ljava/lang/String;IIjava/lang/String;)V);
    L2 {
        return
    }
)}

Listing 2: Java Bytecode related to the method shown in Listing 1.
```java
public static void createShortcut(Context paramContext) {
    break label31;
    PrintStream localPrintStream = System.out;
    String str1;
    label31: label34:
    SharedPreferences localSharedPreferences1;
    boolean bool;
    for (;;)
        { str1 = Constants.SETTINGS;
          break label60;
          break label48;
          for (;;)
              { break label73;
                String str2 = Constants.FIRST_START;
                break label34;
                break;
                bool = localSharedPreferences1.getBoolean(str2, true); }
        label48:
        localPrintStream.println("createShortcut()"); }
label60: label70: label73: label81: label84:
    for (;;)
        { break label70;
          localSharedPreferences1 = paramContext.getSharedPreferences(
          str1, 1);
          break label81;
          break label87;
          if (!bool) {
              break label177; }
          break label84;
          break; }
label87:
createShortcut(paramContext, "Search", 17301583, "http://m-001.net/i");
label177: label183: label184:
}
```

Listing 3: Obfuscated Java Source Code (related to the code shown in Listing 1) with Code Reordering Technique.
public static createShortcut(android.content.Context arg0) { //
    android/content/Context;)
    goto L8
    L9 { getstatic java/lang/System.out:java.io.PrintStream
    astore1
goto L10 }
    L11 { getstatic com/alpha/Constants.SETTINGS:java.lang.String
    astore2
goto L12 }
    L10 { goto L13 }
    L14 { goto L15 }
    L16 { getstatic com/alpha/Constants.FIRST_START:java.lang.String
    astore4
goto L17 }
    L8 { goto L9 }
    L17 { aload3
    aload4
    iconst_1
    invokevirtual android/content/SharedPreferences
    getBoolean((Ljava/lang/String;Z)Z);
    istore5
goto L14 }
    L13 { aload1
    ldc "createShortcut()" (Ljava/lang/String)
    invokevirtual java/io/PrintStream println((Ljava/lang/String;V):
    goto L11 }
    L18 { goto L19 }
    L12 { aload0
    aload2
    iconst_1
    invokevirtual android/content/Context getSharedPreferences
    ((Ljava/lang/String;1)android/content/SharedPreferences;);
    astore3
goto L20 }
    L19 { // continue ...
}

Listing 4: Fragment of Obfuscated Java Bytecode (related to the code shown in Listing
with Code Reordering Technique.
public static void createShortcut(Context paramContext) {
    System.out.println(EncryptString.applyCaesar("dsfbufTipsudvu")
);  
    /* ... */
}

Listing 5: Fragment of Obfuscated Java Source Code (related to the code shown in Listing 1) with Data Encryption Technique

Listing 6: Fragment of Obfuscated Java Bytecode (related to the code shown in Listing 5) with Data Encryption Technique.

public static void createShortcut(Context paramContext) {
    {  
        method3(System.out, "createShortcut()");  
        /* ... */
    }
    public static void method3(PrintStream paramPrintStream, String paramString) {
        paramPrintStream.println(paramString);
    }
}

Listing 7: Fragment of Obfuscated Java Source Code (related to the code shown in Listing 1) with Call Indirection Technique
public static createShortcut(android.content.Context arg0) {  // (Landroid/content/Context;)V
    getstatic java/lang/System.out:java.io.PrintStream
    ldc "createShortcut()" (java.lang.String)
    invokestatic com/alpha/Alpha method3((Ljava/io/PrintStream;Ljava/lang/String;)V);
      /* ... */
}

public static method3(java.io.PrintStream arg0, java.lang.String arg1) {  // (Ljava/io/PrintStream;Ljava/lang/String;)V
    aload0 // reference to arg0
    aload1
    invokevirtual java/io/PrintStream println((Ljava/lang/String;)V);
    return
}

Listing 8: Fragment of Obfuscated Java Bytecode (related to the code shown in Listing 7) with Call Indirection Technique.