LEILA: formAl tool for idEntifying mobille maLicious behAviour

Gerardo Canfora, Fabio Martinelli, Francesco Merca�do, Vittoria Nardone, Antonella Santone, Corrado Aaron Visaggio

Abstract—With the increasing diffusion of mobile technologies, nowadays mobile devices represent an irreplaceable tool to perform several operations, from posting a status on a social network to transfer money between bank accounts. As a consequence, mobile devices store a huge amount of private and sensitive information and this is the reason why attackers are developing very sophisticated techniques to extort data and money from our devices. This paper presents the design and the implementation of LEILA (formAl tool for idEntifying mobille maLicious behAviour), a tool targeted at Android malware families detection. LEILA is based on a novel approach that exploits model checking to analyse and verify the Java Bytecode that is produced when the source code is compiled. After a thorough description of the method used for Android malware families detection, we report the experiments we have conducted using LEILA. The experiments demonstrated that the tool is effective in detecting malicious behaviour and, especially, in localizing the payload within the code: we evaluated real-world malware belonging to several widespread families obtaining an accuracy ranging between 0.97 and 1.

Index Terms—Security, malware, model checking, testing, Android

1 Introduction

In Q3 2017, the researchers of Kaspersky Lab detected 1,598,196 mobile malicious installation packages, 1.2 times more than in the previous quarter\(^1\). The mobile malware phenomenon is exponentially growing; as a matter of fact, GData experts report that they identify a new Android malware strain every 11 seconds\(^2\).

Security patches are released as soon as a vulnerability is found, which make most of the existing malware obsolete. Then, malware writers search for new vulnerabilities and release new malware, or variants of old malware, to get around the new protections. New patches are released to fix the newly discovered vulnerabilities and the cycle starts over again. McAfee security experts define this process as “vicious cycle”\(^3\).

Variants of existing malware are usually generated by adding new behaviours or merging together parts of existing malware codes, since writing effective malware from scratch is costly and antimalware barely detects unknown variants of known malware\(^1, 2\). For this reason, malware analysts use to group malware samples belonging to the same family in order to share significant parts of a malicious behaviour.

This scenario is made more critical by enterprising malware authors who often package their malware as an exploit kit for sale to other cybercriminals, as for example the AndroRAT APK Binder tool, that allows a remote attacker to control the infected device by using a user friendly control panel\(^3\). A weak point of the current antiviruς solutions is their main operating mechanism: they mostly apply a signature-based detection. This approach requires the vendor’s awareness of the malware, it identifies the signature and sends out updates regularly. Signatures have traditionally been in the form of fixed strings and regular expressions, for this reason minimal variations are sufficient to alter the signature\(^2, 4, 5, 6\), making the malware no longer identifiable.

The current literature proposes several detection methods that do not rely on signatures; for instance, methods able to detect apps that exhibit malicious behaviours based on required resources, such as permissions, suspicious API calls and system calls retrieved using both static and dynamic analysis. For example, CrowDroid\(^7\) collects system call traces, while DroidAPIMiner\(^8\) statically mines API in order to catch malware behaviours in mobile environment.

However, in order to bypass detection, malware writers are developing more aggressive paradigms of attacks. A trend in this area is represented by splitting the payload in several methods\(^9\); current antimalware, that tends to search harmful actions within methods separately taken, fails in this case. Considering that the Android operating system is event-based, the several pieces composing the split payload do not require to be invoked by a chain of methods where, after an initial trigger, one calls the following one till the completion of the payload execution. On the contrary, each method of the payload needs only an external event to be triggered, but different triggers may be used for different methods: it is only needed that those triggers be somehow synchronized for performing correctly and completely the harmful action. For instance, it may happen that the malicious payload, able to send sensitive information to a Command and Control (C&C) server, is split in two different actions: the first one gathers the sensitive information, while the second one sends the information over the network. The two methods are not related by a static point of view, the call-graph analysis does not find a link between the two methods, but at runtime they are executed in a sequence. It

\(^1\) https://securelist.com/it-threat-evolution-q3-2017-statistics/83131/
\(^2\) http://pliki-gdata.pl/partner/materiały_prasowe/2016/02/MMWR_EN_Q42015.pdf
may happen that the first harmful action is invoked when the user boots the device (i.e., when the BOOT_COMPLETED event is launched), while the second one when the user receives a call: this mechanism makes the detection more difficult, considering that the full harmful action consists of the composition of different methods, stored in different classes and different packages that do not require to be called statically.

Inter-procedural analysis has been already explored by researchers in order to track the flow of privacy sensitive data by identifying all the dependent variables and statements [10], [11], [12]. LEILA, with respect to existing inter-procedural analysis techniques, is also able to identify other common widespread high-level malicious behaviors, like, for example, ransomware encryption or an attack perpetrated exploiting update mechanisms. As a matter of fact, the typical structure of Android apps offers many opportunities for malware writers: in Android, an app consists of a collection of several components, each in charge of some high-level functionality. LEILA overcomes the existing techniques, as it identifies the high-level malicious behaviors even if they are not implemented in methods that are referenced among each others: this is the reason why we present LEILA as a tool able to localize the malicious payload even if it is split cross different unrelated methods. A fragmented payload which is scattered over different methods without a reference among them cannot be detectable by Inter-component communication (ICC).

In addition, malware writers generally use obfuscation techniques [13] aiming at changing the signature of the code, preventing a virus scanner detecting the obfuscated virus with search strings; malware writers employ these techniques in order to maximize the window of opportunity that elapses between the release of a new threat and the generation of the signature suitable to its recognition.

In this landscape, the pernicious use of communication dislocated in different points of an app (for instance, the method $M_1$ of $C_1$ class retrieves sensitive information that $M_2$ method provided by $C_2$ class will send to a drop server by using network resources) allows to successfully perform a large number of harmful actions.

As resulted by an extensive analysis on data exfiltration done by Android malware [12], one of the main and common goals of malware for mobile devices is to steal sensitive information, by sending it to a drop server or a remote controller. This kind of data are useful to cyber-criminals for many reasons: identity theft, scams, phishing, harassment, espionage, sensitive information gathering. It is clear that the practice of splitting malicious behaviors into several small, and apparently not harmful, actions implemented in different parts of the app is a very appealing mechanism for malware writers in order to elude the many detection mechanisms adopted by antimalware. An example of malware presenting split payload across parts that are not referenced among each others, but implement a cooperation mechanism, is shown in Figs. 1 and 2.

The two code snippets are extracted by two classes (the “Waste” and the “Doctype” classes) of the same app belonging to the BaseBridge malware family: the first one (in Fig. 1) is looking for the execution of the following antimalware packages: “com.qihoo360.mobilesafe”, “com.tencent.qqpimsecure” and “com.lbe.security” (the package names are stored in the encrypted string at rows 3 and 4): whether anyone of them is detected, the malware will try to stop the corresponding process (of the antimalware). The second snippet (in Fig. 2) tries to download the malicious payload, possibly with the antimalware not enabled. As a matter of fact, the dynamic loading can be considered a critical operation by most antimalware. The malware writers often do their best to disable the antimalware before the malicious update is downloaded.

We propose a behavioural based method for detecting Android malware families. More precisely, we define a model-checking-based method able to identify malicious actions and localize the payload, regardless of whether they are implemented or not in same method or class of the application. Model checking [14] is an effective technique to automatically verify the correctness of a system with respect to a desired behaviour, by checking whether a mathematical model of the system satisfies a formal specification of such behavior, expressed using a temporal logic. In this paper we exploit the capabilities of model checking to detect Android malware. In particular, we use the Milner’s Calculus of Communicating Systems (CCS) [15]. CCS is one of the most well known process algebras and it is largely used for modeling complex systems. We specify each Android component in CCS but we do not check each component separately. As explained above, analyzing methods separately is not enough to detect some characteristics of malware. For this reason, in this paper we perform an inter-methods analysis of apps.

In summary, the original contributions of the paper are:
- the implementation of LEILA, a tool exploiting the proposed method;
- a set of rules for specifying many malicious behaviors characterizing different kinds of malware families spread with infamous malware campaigns;
- the evaluation of the effectiveness of our method through the application of LEILA to widespread real-world Android malware families. Our experimentation includes a dataset containing malware families released from 2011 to 2016. The samples of malware cover a huge spectrum of malicious payloads, from the ransomware encryption abilities to the newest Android root exploits exhibited by the HummingBad family;
- the localization of fragmented payloads, as well as the localization of malicious payloads placed in one only method or
referred methods;
• the use of the model checking technique for Android apps
to identify malware families with the management of the method
calls, which has never been investigated, to the best
of the authors’ knowledge;
• the automatic dissection of a mobile app: the events able to
toggle the malicious behavior can be caught and this is the
reason why LEIL.A can be also used as a sanitizer tool, by
disarming the payload;
• the management of multi-threading in Android applications
with a solution to tackle the state explosion problem arising
by the interleaving of the threads’ executions.

Besides the above novelties, other distinctive and relevant
strengths of our method are:
• the detection of malware through the analysis of the Java
Bytecode instead of the source code, which is an advantage
because source code could not be available or obtained by
reverse engineering (due to obfuscation, for instance);
• the identification of malware families, which can help when
dissecting a malware in order to make the picture about the
possible behavior and features of the malware;
• a behaviour-based detection as an approach to mitigate the
effect of code obfuscation techniques.

The reminder of the paper is organized as follows: Section 2
reports the preliminaries on formal methods, Section 3 introduces
the method, Section 4 illustrates the results of the experiment,
Section 5 discusses the distinctive features and the limitations
of our method. Section 6 thoroughly analyses related work. Finally,
conclusions and future works are given in Section 7.

2 Preliminaries on Formal Methods

Formal methods are mathematically based languages, techniques,
and tools for specifying and verifying complex systems. To apply
formal methods, we need:
• a precise notation for defining systems;
• a precise notation for defining properties.

We examine in details these two notions in the following two
subsections.

2.1 A precise notation for defining systems

Specification is the process of describing a system. We assume that
the system behaviour is represented as an automaton. It basically
consists of a set of nodes together with a set of labelled edges
between these nodes. A node represents a system state, while
a labelled edge represents a transition from one system state to
the next. That is, if the automaton contains an edge $s \xrightarrow{a} s'$,
then the system can evolve from state $s$ into state $s'$ by the
execution of action $a$. One state is selected to be the root state,
i.e., the initial state of the automaton. However, for the purpose
of mathematical reasoning, it is often convenient to represent the
automaton algebraically in the form of processes. For this aim,
we use Milner’s Calculus of Communicating Systems (CCS) [15].
CCS is one of the most well known process algebras and it is
largely used for modeling complex systems. Readers unfamiliar
with CCS are referred to [15] for further details. CCS contains
basic operators to build finite processes, communication operators
to express concurrency, and some notion of recursion to capture
infinite behaviour. The syntax of processes is the following:
$$p ::= \text{nil} \mid a.p \mid p + p \mid p|p \mid p\{L\} \mid p[f] \mid x$$

where $\alpha$ ranges over a finite set of actions $A = \{\tau, a, n, b, \ldots\}$. Input actions are labelled with “non-barred” names, i.e., $a$, while output actions are “barred”, e.g., $\tau$. The action $\tau \in A$ is called
internal action. The set $L$ ranges over sets of visible actions ($A -
\{\tau\}$), $f$ ranges over functions from actions to actions, while $x$
ranges over a set of constant names: each constant $x$ is defined by
a constant definition $x \overset{\text{def}}{=} p$.

We give the semantics for CCS by induction over the structure
of processes:

• the process $\text{nil}$ can perform no actions;
• the process $a.p$ can perform the action $a$ and thereby become
the process $p$;
• the process $p + q$ can behave either as $p$ or as $q$;
• the operator $|$ expresses parallel composition: if the process
$p$ can perform $\alpha$ and become $p'$, then $p|q$ can perform $\alpha$
and become $p'|q$, and similarly for $q$. Furthermore, if $p$ can perform
a visible action $I$ and become $p'$, and $q$ can perform
$I$ and become $q'$, then $p|q$ can perform $\tau$ and become $p'|q'$;
• the operator $\setminus$ expresses the restriction of actions. If $p$
can perform $\alpha$ and become $p'$, then $p\setminus L$ can perform $\alpha$
to become $p'\setminus L$ only if $\alpha, \overline{A} \notin L$;
• the operator $[f]$ expresses the relabelling of actions. If $p$
can perform $\alpha$ and become $p'$, then $p[f]$ can perform $f(\alpha)$
and become $p'[f]$;
• each relabelling function $f$ has the property that $f(\tau) = \tau$;
• a constant $x$ behaves as $p$ if $x \overset{\text{def}}{=} p$.

A (labelled) 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 $\delta =
\alpha_1 \ldots \alpha_n, n \geq 1$, we write $s \xrightarrow{\delta} s'$ to mean $s \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s'$. Moreover
$s \xrightarrow{\lambda} s$, where $\lambda$ is the empty sequence. Given $s \in S$, with $R(p) =
\{s' \mid s \xrightarrow{\delta} s'\}$ we denote the set of the states reachable from $s$
by $\rightarrow$. Given a CCS process $p$, the standard transition system
for $p S(p) = (R(p), A, \rightarrow, p)$. Note that, with abuse of notation,
we use $\rightarrow$ for denoting both the operational semantics and the
transition relation among the states of the transition system.

The operational semantics of a process $p$ is a labelled transition
system, i.e., an automaton whose states correspond to processes
(the initial state corresponds to $p$) and whose transitions are
labelled by actions in $A$ (see Table 1). This automaton is called
standard transition system of $p$ and denoted by $S(p)$.

2.2 A precise notation for defining properties

This need can be solved using a temporal logic. Temporal logics
present constructs allowing to state in a formal way that, for
instance, all scenarios will respect some property at every step,
or that some particular event will eventually happen, and so on.
The most noticeable examples are:

• safety properties, which state that an undesirable situation
will never arise;
• liveness properties, which state that some actions will always
be followed by some reactions.

A model checker then accepts two inputs, a system described, for
example, in process-algebraic notations and a temporal formula,
and returns “true” if the system satisfies the formula and “false”
otherwise. In this paper we use the logic selective mu-calculus
[16]. It was defined with the goal of reducing the number of states
of the transition systems in such a way that the reduction is driven
TABLE 1: Standard operational semantics of CCS

<table>
<thead>
<tr>
<th>Act</th>
<th>Sum</th>
<th>Par</th>
<th>Com</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha.p \xrightarrow{\alpha} p )</td>
<td>( p \xrightarrow{\alpha} p' ) (and symmetric)</td>
<td>( p \xrightarrow{\alpha} p' )</td>
<td>( p \xrightarrow{i} p', q \xrightarrow{r} q' )</td>
<td>( p \xrightarrow{\alpha} p' )</td>
</tr>
<tr>
<td>( p \xrightarrow{f} p'[f] )</td>
<td>( p + q \xrightarrow{\alpha} p' ) (and symmetric)</td>
<td>( p{q \xrightarrow{\alpha} p'{q } )</td>
<td>( p{q \xrightarrow{\alpha} p'{q } )</td>
<td>( p \xrightarrow{\alpha} p' )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Con</th>
<th>Comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p \xrightarrow{\alpha} p' )</td>
<td>( x \xrightarrow{\alpha} p' )</td>
</tr>
</tbody>
</table>

by the formulae to be checked, and in particular by the syntactic structure of the formulae. The selective mu-calculus is a variant of the mu-calculus [17], and differs from it in the definition of the modal operators. The syntax of the selective mu-calculus is the following, where \( K \) and \( R \) range over sets of actions, while \( Z \) ranges over a set of variables:

\[
\varphi ::= \top | \bot | \mathsf{ff} | Z | \varphi \lor \varphi | \varphi \land \varphi | (K)_R \varphi | (K)_R \varphi | \nu Z. \varphi | \mu Z. \varphi
\]

The satisfaction of a formula \( \varphi \) by a state \( s \) of a transition system is defined as follows:

- each state satisfies \( \top \) and no state satisfies \( \bot \);
- a state satisfies \( \varphi_1 \lor \varphi_2 \) (\( \varphi_1 \land \varphi_2 \)) if it satisfies \( \varphi_1 \) or (and) \( \varphi_2 \);
- \( [K]_R \varphi \) and \( (K)_R \varphi \) are the selective modal operators. They require that the formula \( \varphi \) is satisfied after the execution of an action of \( K \), provided that it is not preceded by any action in \( K \cup R \). More precisely: \( [K]_R \varphi \) is satisfied by a state which, for every performance of a sequence of actions not belonging to \( R \cup K \), followed by an action in \( K \), evolves in a state obeying \( \varphi \). \( (K)_R \varphi \) is satisfied by a state which can evolve to a state obeying \( \varphi \) by performing a sequence of actions not belonging to \( R \cup K \) followed by an action in \( K \).

The precise definition of the satisfaction of a closed formula \( \varphi \) by a state is given in Table 2.

As in standard mu-calculus, a fixed point formula has the form \( \mu Z. \varphi \) (\( \nu Z. \varphi \)) where \( \mu Z. (\nu Z. \varphi) \) binds free occurrences of \( Z \) in \( \varphi \). 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. \( \mu Z. \varphi \) is the least fix-point of the recursive equation \( Z = \varphi \), while \( \nu Z. \varphi \) is the greatest one. To give an intuition of their meaning, consider the formulae \( \varphi = \mu Z.(\psi \lor (a)_b Z) \) and \( \varphi' = \nu Z.(\psi \land (a)_b Z) \). A transition system satisfies \( \varphi \) if it can evolve to a state satisfying \( \psi \) after a finite number of occurrences of action \( a \) (ignoring all other actions), while it satisfies \( \varphi' \) if it satisfies \( \psi \) along any path containing \( a \) (ignoring all other actions).

A transition system \( T \) satisfies a formula \( \varphi \), written \( T \models \varphi \), if and only if \( q \models \varphi \), where \( q \) is the initial state of \( T \). A CCS process \( p \) satisfies \( \varphi \) if \( S(p) \models \varphi \).

The selective mu-calculus is equivalent to the mu-calculus. In fact it is easy to see that the standard mu-calculus modal operators \( [K]_R \varphi \) and \( (K)_R \varphi \) can be defined by means of the selective operators subscribed by the whole set of actions \( A \):

\[
[K]_R \varphi = [K]_R \varphi \text{ and } (K)_R \varphi = (K)_R \varphi
\]

On the other hand the selective operators can be expressed in standard mu-calculus as follows:

\[
\langle K \rangle_R \varphi \overset{\text{def}}{=} \mu Z. (\langle K \rangle \varphi \lor \langle A \rangle \varphi) Z
\]

\[
[K]_R \varphi \overset{\text{def}}{=} \nu Z. (\langle K \rangle \varphi \land \langle A \rangle \varphi) Z
\]

In the sequel we will use the following abbreviations (where \( K \) ranges over sets of actions and \( A \) is the set of all actions):

\[
\{\alpha_1, \ldots, \alpha_n\}_{R \varphi} \overset{\text{def}}{=} \{\alpha_1, \ldots, \alpha_n\} \{[A]_R \varphi \}
\]

\[
[-]_R \varphi \overset{\text{def}}{=} [A]_R \varphi
\]

This implies that to check selective mu-calculus formulae we can use all formal verification environments which accept mu-calculus as property specification language.

Example 2.1. We give some examples of selective mu-calculus formulae to explain the use of the selective operators.

\( \psi_1 = [a]_{[b]} ff \): “it is not possible to perform an action \( a \) if an action \( b \) has not been previously performed”.

\( \psi_2 = [a]_{tt} \): “it is possible to perform an action \( a \) preceded by any action”.

\( \psi_3 = \nu Z.[a] Z \land [a]_{[b]} ff \): “it always holds that, after an action \( a \) has occurred, a successive \( a \) cannot occur before either an action \( b \) or an action \( c \) has occurred”.

![Fig. 3: Three transition systems](image)

Let us consider the transition systems in Fig. 3. It holds that:

<table>
<thead>
<tr>
<th></th>
<th>( S_1 \models \psi_1 )</th>
<th>( S_2 \models \psi_1 )</th>
<th>( S_3 \models \psi_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 \models \psi_2 )</td>
<td>( S_2 \models \psi_2 )</td>
<td>( S_3 \models \psi_2 )</td>
<td></td>
</tr>
<tr>
<td>( S_1 \models \psi_3 )</td>
<td>( S_2 \notmodels \psi_3 )</td>
<td>( S_3 \notmodels \psi_3 )</td>
<td></td>
</tr>
</tbody>
</table>
TABLE 2: Satisfaction of a closed formula by a state

<table>
<thead>
<tr>
<th>Formula</th>
<th>Satisfaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>p \not= ff</td>
<td>p ⊨ \varphi ∧ \psi if \ p ⊨ \varphi \ and \ p ⊨ \psi</td>
</tr>
<tr>
<td>p ⊨ \varphi ∨ \psi</td>
<td>p ⊨ \varphi \ or \ p ⊨ \psi</td>
</tr>
<tr>
<td>p ⊨ [K]_R \varphi</td>
<td>\forall p', \forall \alpha \in K.p \xrightarrow{\alpha} K,R p' \implies p' \models \varphi</td>
</tr>
<tr>
<td>p ⊨ [K]_R \varphi</td>
<td>\exists p', \exists \alpha \in K.p \xrightarrow{\alpha} K,R p' \ and \ p' \models \varphi</td>
</tr>
<tr>
<td>p ⊨ vZ, \varphi</td>
<td>p \models vZ^n, \varphi \ for \ all \ n</td>
</tr>
<tr>
<td>p ⊨ \mu Z, \varphi</td>
<td>p \models \mu Z^n, \varphi \ for \ some \ n</td>
</tr>
</tbody>
</table>

where:

- for each n, vZ^n, \varphi \ and \ \mu Z^n, \varphi \ are \ defined \ as:
  \[ vZ^n, \varphi = \tau \tau \quad \mu Z^n, \varphi = \tau \tau \]
  \[ vZ^{n+1}, \varphi_1 = \varphi[vZ^n, \varphi_1/Z] \quad \mu Z^{n+1}, \varphi_1 = \varphi[\mu Z^n, \varphi_1/Z] \]

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

- \ p \xrightarrow{\alpha_{ij}} q \quad \ p \xrightarrow{\delta} q, \ \text{where} \ \delta \in (\mathcal{A} - I)^+ \ \text{and} \ I \subseteq \mathcal{A}. 

One of the most popular environments for verifying concurrent systems is the CAAL (Concurrency Workbench, Aalborg Edition) [18], which supports several different specification languages, among which CCS. In the CAAL, the verification of temporal logic formulae is based on model checking [19]. In this paper we use CAAL as formal verification environment. It is worth noting that CAAL represents output actions as ‘a instead of \(\alpha\). In this paper we use this notation.

### 3 The method

The method we propose consists of three main steps:

1. the creation of a formal model;
2. the definition of the temporal logic formulae that describe the characterizing behaviors of Android malware families;
3. the verification of the formulae with a formal verification environment.

#### Step 1: Creation of a formal model

The first step aims at generating a formal model by parsing the Java Bytecode of the app to be analyzed; it is represented in Fig. 4 (Box 1). More precisely, the Bytecode of the target app that resides in a class folder or in JAR files is fed to a custom parser, based on the Apache Commons Bytecode Engineering Library (BCEL)\(^4\). The .class files of the parsed Java Bytecode are successively translated into formal models. Since we use the Calculus of Communicating Systems (CCS) [15], we specify a Java Bytecode-to-CCS transform operator \(T\). The function \(T\) directly applies to the Java Bytecode and translates it into CCS process specifications. The function \(T\) is defined for each instruction of the Java Bytecode. We assume that a Java Bytecode program \(P\) is a sequence \(c\) of instructions, numbered starting from the address 0; \(\forall i \in \{0, \ldots, |c|\}\), and \(c[i]\) is the instruction at the address \(i\), where \(|c|\) denotes the length of \(c\). All the Java Bytecode instructions have been translated in CCS, i.e., we associate a new CCS process to each Java Bytecode instruction. This translation has to be performed only one time for each app and it has been completely automated. A preliminary function \(T\) has been defined in [20], [21]. However, the CCS generated model was very simple, since it did not consider many features, such as the try-catch construct and method invocations. Thus, all the methods were analyzed separately. It is worth noting that analyzing methods separately is not enough to detect malware: it is needed to perform an inter-methods analysis of the app. In fact, an Android app includes multiple components, i.e., activities, services, broadcast receivers, and content providers. ICC is performed using an intent, which is a messaging object that contains the destination component’s address or action string, and possibly data. While regular Java programs have a single entry point, Android apps can have multiple entry points.

In the light of all above, in this paper we enrich our model eliminating the previous limitations [20]. For clarity’s sake, the translation of some Java Bytecode instructions is provided.

#### Instruction: \(c[i] = \text{goto } j\)

\[ T(i) = x_i \overset{\text{def}}{=} \text{goto } j \]

The instruction \(c[i] = \text{goto } j\) is translated into a CCS process \(x_i\), that performs 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 translated, regardless of the type \(t\) and of the name of the variable \(z\), as \text{store} followed by the constant process \(x_i+1\) representing the CCS translation of the successive instruction.

#### Instruction: \(c[i] = \text{if } \text{cond } j\)

\[ T(i) = x_i \overset{\text{def}}{=} \text{if } \text{cond }_{\text{ff}} x_i+1 + \text{if } \text{cond }_{\text{tt}} x_j \]

Conditional jumps are instead specified as non-deterministic choices. In this case we define the CCS constant $x_i$ corresponding to the instruction $i$. The true (resp. false) condition is represented by the CCS action $\text{if}\_\text{cond}\_\text{tt}$ (resp. $\text{if}\_\text{cond}\_\text{ff}$), while $\text{cond} \in \{\text{eq}, \text{ne}, \text{icmeq}, \text{icmpne}, \ldots\}$.

Thus, if the condition is true, the CCS process $x_i$ performs the action $\text{if}\_\text{cond}\_\text{tt}$ 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 the address $i + 1$ (process $x_{i+1}$).

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

$$\mathcal{T}(i) = x_i \triangleq \text{athrow.nil}$$

The instruction $\text{athrow}$ throws an error or exception and this is represented by the CCS process $x_i$ which terminates after performing the action $\text{athrow}$.

**Instruction:** $c[i] = \text{invoke\_type MethodName}_i$

$$\mathcal{T}(i) = x_i \triangleq \text{callMethodName}_i.\text{returnMethodName}.x_{i+1}$$

Each $\text{invoke\_type}$ instruction, where

$$\text{type} \in \{\text{dynamic, interface, special, static, virtual}\}$$

is represented by the CCS process $x_i$ which calls the method $\text{MethodName}_i$, performing the action $\text{callMethodName}_i$. The control to the caller process is returned through the action $\text{returnMethodName}_i$ and the execution continues at the successive instruction at address $i + 1$ (process $x_{i+1}$).

Note that, in order to simplify the notation, we have reported only the name $\text{MethodName}_i$ in the CCS translation of a method’s invocation, neglecting the arguments, their types and all the class path. Let’s see, now, the translation of the method definition.

**Method definition:**

visibility type $\text{MethodName}_i$;

Code: $m$

where

- $m$ is a sequence of method’s instructions, numbered starting from address 0; $m[j]$ is the instruction at address $j$, $\forall j \in \{0, \ldots, \#m\}$.
- visibility $\in \{\text{public, protected, private}\}$;
- type $\in \{\text{Primitive data type and any object type}\}$;

For each method definition we specify a CCS process.
in the CCS process
P3 = ‘callclassPath_returnType_methodNameM1_argsList
returnclassPath_returnType_methodNameM1_argsListP4
P4 = ...
... = ...
Ph = ‘returnclassPath_returnType_methodNameP1_argsListP1

Example of the abstract CCS process of the Method M1
M1 = callclassPath_returnType_methodNameM1_argsListM2
M2 = firstInstructionM3
M3 = secondInstructionM4
M4 = ...
... = ...
Mm = ‘returnclassPath_returnType_methodNameM1_argsListM1

Fig. 5: Example of Method Call

MethodName; as follows:

MethodName = \text{def} \text{callMethodName}_i \text{MethodName}_n
where
T(0) = MethodName = p_0
...
T(m-1) = MethodName = p_{m-1}
T(m) = MethodName = \text{def} \text{returnMethodName}_i \text{MethodName}_n

where 0 is the first instruction of the MethodName. Every method corresponds to a process waiting to be invoked through the action \text{callMethodName}_i. The action \text{returnMethodName}_i causes the control being returned to the method caller. The instructions of the body, starting from 0, are translated in CCS processes through T. Then, the process goes back to the starting action so that it can be invoked again.

As already explained for the invocation of a method, also in the method definition the CCS translation shows only the name MethodName, but does not show the arguments, for readability and simplicity. An example of the complete definition of a method is given in Fig. 5, where the definition methods of P1 and M1 are shown. In addition to the name of the method, the class Path, the Return Type and the list of arguments are specified. In the example, P1 calls M1 through the action \text{callclassPath_returnType_methodNameM1_argsList} present in the CCS process P3, while the corresponding non-barred action \text{callclassPath_returnType_methodNameM1_argsList} is in the CCS process M1. These actions allow the communication between the two processes. The actions \text{returnclassPath_returnType_methodNameM1_argsList} in the CCS process P3 and the action \text{returnclassPath_returnType_methodNameM1_argsList} in the CCS process Mm cause the control being returned to the method caller.

Finally, the explicit inter-component communication is codified in CSS as methods invocation while multiple entry points are codified in CSS using the non-deterministic choice operator: in this way they are always active and analyzed by the model checker.

**Exception Table:**

<table>
<thead>
<tr>
<th>Exception table:</th>
<th>from</th>
<th>to</th>
<th>target type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td>k</td>
<td>t</td>
</tr>
</tbody>
</table>

Independently of the type t, all the CCS processes x_i, with i ≤ l < j are modified adding the process x_k as a choice of the body of the process x_j, i.e.,

\[ x_i = \text{def} \ p + x_k \]

where T(l) = x_i = p.

If an exception occurs during the execution of the instructions between indexes i and j (corresponding to the processes x_i, i ≤ l < j), the control is transferred to the Java Virtual Machine code at index k, which is the CCS process x_k implementing the block starting from k.

The final CCS process describing a Java Bytecode program is obtained through the proposed method as a parallel of the Start process and all the MethodName processes, where \( i \in [1..n] \) if n is the number of the methods in the analyzed program, restricted to all the actions used for communicating among such processes, i.e., \text{callMethodName}_i and \text{returnMethodName}_i. Formally,

\[ \text{Final} = \text{def} \{ \text{MethodName}_1 | \cdots | \text{MethodName}_n | \text{Start} \}, S \]

where S = \{ \text{callMethodName}_i, \text{returnMethodName}_i \}, \forall i \in [1..n]

The Start process is so defined:

\[ \text{Start} = \text{def} \ \text{callMethod}_{k_1}.\text{returnMethod}_{k_1}.\text{nil} \]
\[ \cdots \]
\[ \text{callMethod}_{k_n}.\text{returnMethod}_{k_n}.\text{nil} \]

Start is the parallel composition of CCS processes like \text{callMethod}_{i}.\text{returnMethod}_{i}.\text{nil}, with \( i \in \{k_1,\ldots,k_m\} \). Each of these processes has the only aim of starting the multiple entry points Method_i. Unlike Java programs, Android applications do not have a main method. In fact, they can contain multiple components, each one characterized by its own life cycle: the information about the component that will be launched when the app is started is provided by the Manifest file. In addition, considering that Android operating system is event-based, from a static point of view there are methods that are not referenced (and called) by other methods that are triggered when an event happens (for instance, when the user receives a phone call or an SMS). Our method takes into account these intrinsic characteristics of the mobile platform, as a matter of fact, we consider these methods in the CCS Start process. This makes particularly hard the static analysis in Android environment, as it has multiple entry points the analysis should start from. In order to consider all the possible app entry points, we need to introduce Start as parallel composition of all the CCS processes.
An example
To give the reader the flavour of the transformation function $T$, we consider a very simple code fragment that implements the traditional “Hello World” example. This app is composed only of two classes; the first class, shown in Listing 1, calls the method of the other one, shown in Listing 2. The only action performed is the visualisation of the string “Hello World” on the screen. The Java Bytecode of the two Java classes is shown in Listings 3 and 4. The resulting CCS processes are shown in Listings 5 and 6. For simplicity’s sake, we have used $x = p$ instead of $x \equiv p$ in the definition of a CCS process. To better understand the example, we also report the Java code in Listings 1 and 2, even if the transformation function $T$ operates only on the Java Bytecode.

The Listing 7 shows the Final CCS process of the correlation between the two classes. For the sake of clarity, we consider the Start process containing only the Main method call. The Final process does not involve the ExampleClass and Hello class Constructors into its specification, as shown in Listings 3 and 4 respectively. They are automatically generated during the Java Bytecode visualization, using the command javap (included with the JDK). This clearly emerges in the Listings 1 and 2 which contain the Java source code of the two classes respectively. Since these two methods specify the default Java constructors that we do not want to model checking, the Final process does not include them in its specification. Differently, the Final process will contain the CCS specification of a Java class constructor only if it is declared by the programmer.

Listing 8 and Listing 9 show an example of try-catch translation. In particular, the Listing 8 shows the Java Bytecode of the method containing a try-catch block, while the corresponding CCS translation is reported in Listing 9.

Step 2: Definition of Android malware families’ behaviors as temporal logic formulae
The second step aims at recognizing the characterizing behaviors of the Android malware families, expressed in temporal logic: an example is shown in Fig. 4 (Box 2). Identifying a behavior means verifying the algebraic specification describing the characteristics of malware families on the CCS processes obtained with the first step, which are a representation of the app’s code. Finally, CAAL verifies the logic formulae on the CCS processes, after that they are automatically mapped to labelled transition systems. Depending on the malware family to detect, different properties have been considered. For each family $F$, a selected set of samples has been manually inspected for defining the selective mu-calculus formulae characterising the malware behaviour of the family $F$. For example, in Table 3 is shown an example of Opfake Formula.

The formula means that it is always possible to perform this sequence of actions:

\begin{verbatim}
newandroidcontentIntent invokegetBroadcast invokegetDefault
invokesendTextMessage
\end{verbatim}

In other words, the formula is able to catch those malicious behaviours:
- \textit{newandroidcontentIntent}: the Intent is a mechanism for performing late runtime binding between the codes of different applications. They are mostly used for launching activities;
- \textit{invokegetBroadcast}: this represents an invocation of the get-Broadcast method belonging to PendingIntent class available in Android; it performs a broadcast. Releasing a PendingIntent to another app, corresponds to granting the right to perform the operation specified as argument to another app with the same right of the caller;
- \textit{invokegetDefault}: the getDefault method, belonging to the Locale class, gets the current value of the geographical region of the device; it is used to obtain user personal information;
- \textit{invokesendTextMessage}: the method sendTextMessage, provided by Android SMSManager class, sends a text based SMS; this is an action recurring within malicious payload with information gathering abilities. As a matter of fact, this rule is able to identify the malicious payload that gathers the sensitive information and creates a text SMS with this information to be sent to the attacker (i.e. the person that controls the malware and managed the infection campaign).

Step 3: Application of a formal verification environment
Finally, as shown in Fig. 4 (Box 3), a formal verification environment, including a model checker, is invoked to recognise the malware families through model checking. This step checks which algebraic formulae are verified on which CCS model, i.e., which malware characteristics are found on which app. We use the CAAL tool [18] as formal verification environment. When the result of the CAAL model checker is true, it means that the app belonging to that malware family (the verified algebraic formulae correspond to), false otherwise.

3.1 The LEILA implementation
The method introduced in the previous section has been implemented in a tool named LEILA. LEILA is a Java system. It is composed of four packages:

1) Preprocessing: The aim of this package is to extract the class files of the Android application. In particular, it integrates the dex2jar$^5$ tool in order to convert the dex (i.e., the Dalvik Executable) file into jar (i.e., Java Archive) file. Using the dex2jar tool, LEILA obtains the Android application in a compressed format. To extract the classes from the jar file, LEILA uses the command: jar -xvf provided by the Java Development Kit. The outputs of this package are the classes belonging to the Android app;

2) Parsing & Translation: It is the core of the tool since it builds the CCS model. Starting from the output of the first package and using the Bytecode Engineering Library (Apache Commons BCEL), LEILA parses the class files. Afterwards, starting from the main activity of the application, LEILA analyzes all the methods and all the multiple entry points of an Android application generating the Final CCS process, as explained above. Thus, LEILA considers all events able to trigger activities, services and any other Android component. This is obtained by implementing the transformation function $T$. The output of this package is the CCS model of the app;

3) Reduction: It implements the CCS model reduction. This package takes as input two files: the CCS model and the temporal logic formulae that describe the malicious behaviors. Starting from the logic formulae, LEILA is able to reduce the formal model of the Android application. The readers can

\footnote{https://sourceforge.net/projects/dex2jar/}
find more details about this process in Section 5. The output of this package is the reduced CCS model;

4) Verification: It realizes the formal verification of the model. It verifies the model, i.e., the reduced CCS model, through the CAAL model checker, checking the temporal logic formulae: this package invokes the CAAL model checker. The output of this package is the identification of the malware family, if any, along with with the payload localization.

The process described above is detailed considering one Android application as input, but LEILA is able to work on a set of applications, too.

4 EVALUATION

The goal of this study is to investigate the suitability and the limitations of the proposed method in recognizing the malware families targeting the Android operating system. The quality focus concerns the effectiveness and the performances of the detection performed by LEILA. The perspective is that of researchers interested in evaluating a method based on model checking that associates a malware to the family it belongs to and locates the actual payload in the code.

We evaluated the following capabilities of LEILA:

1) effectiveness of malware classification: the correctness in identifying the malware family a sample belongs to;

2) resiliency against obfuscation: how the classification capabilities are affected by the obfuscation of malware; and

3) performances of detection: which is the workload required to the computational resources when running LEILA.

To estimate the LEILA effectiveness of malware classification, we compute the metrics of precision and recall, F-measure (Fm) and Accuracy (Acc), defined as follows:

\[
\text{Precision (Pr)} = \frac{TP}{TP + FP}; \quad \text{Recall (Rc)} = \frac{TP}{TP + FN};
\]
null
**HummingBad** family was download by the ContagioMobile website.8

Here is a brief description of the 6 populous families in the dataset.

1) The Opfake samples make use of an algorithm that can change shape over time to evade the detection. The Opfake malware demands payment for the application content through premium text messages. The Opfake malicious payload is triggered by the user i.e., by an UI event. As a matter of fact, SMS messages to premium-rate numbers are sent when the wapxload.ru/opera/ web page is displayed on the infected device screen once the user opens it. The malicious payload application does not read the content of the HTTP response but it is able to enables the JavaScript functionality in the browser and registers a callback method named onJsPrompt. This family represents an example of polymorphic malware [26] in Android environment: it is written with an algorithm that can change shape over time to evade detection by signature based antimalware;

2) GinMaster family contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and send the stolen data to a remote website, as well as install applications without user interaction. It is also a trojan application (basically, the term “trojan” relates to an application that hides its true purpose i.e., the concealment of the malicious functions behind harmless ones) and starts its malicious services as soon as it receives a BOOT_COMPLETED or USER_PRESENT intent. The malware can successfully avoid detection by mobile antivirus software by using polymorphic techniques to hide malicious code, obfuscating class names for each infected object, and randomizing package names and self-signed certificates for applications (typically polymorphic malware attempts to evade detection by encrypting itself differently, and rewriting the decrypting module accordingly. The second variant of GinMaster is able to make use of the encryption as, but the decrypting module is static);

3) the main aim of FakeInstaller samples is to send SMS messages to premium rate numbers, without the consent of the user, passing itself off as the installer for a legitimate application. When a FakeInstaller sample is executed, it displays a service agreement: the user is forced to click an “Agree button”, which sends the premium SMS messages. There are also seen variants that send the messages before the user of the infected device clicks this “Agree button” 10. Furthermore, FakeInstaller samples are server-side polymorphic i.e., the server is able to provide different APK files for the same URL request. For instance, when the user requests an application from the fake market provided by FakeInstaller, the server redirects the browser to another server able to process the request in order to send a customized APK which has an associated ID in the generated URL: the installation package sent to the user is associated with the IP address of the infected device11.

4) the Plankton payload is able to silently forward information about the device to a drop server. It does not require root privileges and the payload is downloaded from a remote location. The malicious service checks the details of the installed application, including its security permissions, and it sends the details to a hardened HTTP server. The server replies with a URL that is used to download a JAR file that contains the malicious payload [27]. The downloaded Dex code launches a connection to the Command server and listens for commands to execute (for instance, send details like IMEI, browser history, permissions granted) [25];

5) Ransomware is one of the newest threats in mobile environment: basically the samples belonging to this category hinder the access to the device or the data within the device until a ransom is paid. They employ different techniques in order to lock the phone (i.e., to take control of the user interface either locking the screen always on the same view or encrypting all files of the sd-card) [28], [29]. Typically these samples masquerade as well-known applications, such as Adobe Flash and a number of antimalware applications, and pretend to scan the device upon launch. After completing the fake scan it locks the infected device [28]. The user is not able to navigate away and if he/she tries to reboot, the fake FBI message will be the first thing the user sees when the device turns on. Furthermore, it demands several hundred dollars in a MoneyPak voucher in order to release the infected device [29];

6) the HummingBad family represents the most recent threat in the evaluated malware dataset: as a matter of fact, it was discovered by Check Point analysts in February 201612. In July 2016, the same analysts discovered that this kind of malicious payload is able to install more than 50,000 fraudulent applications each day, displays 20 million malicious advertisements, and generates more than $300,000 per month in revenue13. The attacks use multiple exploits in an attempt to gain root access on a device. When rooting fails, a second component delivers a fake system update notification in hopes of tricking users into granting HummingBad system-level permissions14. Whether or not rooting succeeds, HummingBad downloads a large number of apps. In some cases, malicious components are dynamically downloaded onto a device after an infected app is installed. From there, infected phones display illegitimate ads and install fraudulent apps after certain events, such as rebooting, the screen turning on or off, a detection that the user is present, or a change in Internet connectivity. HummingBad also has the ability to infect code into Google Play in order to tamper with its ratings and statistics. This is done by using infected devices to imitate clicks on the install, buy, and accept buttons15.

In order to download trusted applications, we crawled the

9. https://usa.kaspersky.com/resource-center/threats/trojans#VvD3eOLdDrQ
Google’s official app store\textsuperscript{16}, by using an open-source crawler\textsuperscript{17}. The obtained trusted dataset includes samples belonging to all the different categories available on the market.

The legitimate applications were collected between January 2016 and April 2016.

The (malware and trusted) dataset is composed by 400 samples, we randomly selected 50 samples belonging to each family; we described and 50 applications retrieved from Google Play. Moreover, in order to better evaluate the families misclassification, we take into account additional 50 samples belonging to malware families different from the ones considered into the evaluation (i.e., 50 samples randomly selected among the top ten most populous families of Drebin project dataset not already considered).

Table 5 shows the number of samples considered for each family in the experiment. More precisely, the first column of the table shows the number the samples manually inspected to generate the selective mu-calculus logic formulae characterising the malicious behavior of the family. Clearly, for the “Trusted” apps and “Other Malware” ones, no sample has been manually inspected since for them no logic formula has been specified. The second column of the table shows the number of samples that have been evaluated by LEILA. Finally, the third column shows the total number of the samples considered in our experiment (which is the sum of the other two columns). With the aim of evaluating the misclassification, we applied the formulae to other malware families: the number of these samples is shown in “Other Malware” row.

<table>
<thead>
<tr>
<th>Family</th>
<th># Inspected Samples</th>
<th># Evaluated Samples</th>
<th># Samples Considered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opfake</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>GinMaster</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>FakeInstaller</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>Ransomware</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>HummingBad</td>
<td>5</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>Trusted</td>
<td>-</td>
<td>-</td>
<td>50</td>
</tr>
<tr>
<td>Other Malware</td>
<td>-</td>
<td>-</td>
<td>50</td>
</tr>
</tbody>
</table>

In order to validate the correctness of the dataset’s classification, we analyzed the dataset with the VirusTotal service\textsuperscript{18}, a service running 57 different antimalware software (i.e., Symantec, Avast, Kasperky, McAfee, Panda, and others): the analysis confirmed that the legitimate applications did not contain malicious payload while the malware ones were actually recognized as malicious.

### 4.2 Results

Table 6 shows the results of our evaluation. LEILA recognizes the evaluated Android malware families with an accuracy ranging between 0.97 and 1.

The precision varies between 0.84 and 1, while the recall between 0.86 and 1. It is worth noticing that we obtained very low FP values: 4 for Opfake (i.e., 1%), 2 for GinMaster (i.e., 5%); 9 for Ransomware (i.e., 2.2%); 0 for Plankton samples; 5 (i.e., 1.2%) for FakeInstaller samples and 0 for HummingBad.

Table 7 reports the achieved results with the obfuscated samples showing that the performances keep pretty unchanged.

In order to generate the obfuscated versions of the malware, we used the DroidChameleon\textsuperscript{5} tool. Basically the obfuscation process consists of code injections (i.e., junk code insertion, code reordering, call indirections and renaming) aimed at generating morphed versions of the applications belonging to the malware dataset.

The DroidChameleon tool applies the transformation techniques not only on the malicious payload but to all the application code. For instance, in case we apply the junk code insertion technique, all the methods of the application will be obfuscated with this technique, the same happens with the other injected obfuscations. For this reason we can state that 100% of the code will be obfuscated (more precisely considering for instance, the string encryption, the 100% of the strings will be obfuscated).

A previous work\textsuperscript{5} demonstrated that current antimalware solutions fail to recognize the malware after these transformations. We applied our method to the morphed dataset in order to verify if LEILA loses its effectiveness after the malware has been altered.

The analysis confirms that LEILA is resilient to the common code obfuscation techniques. LEILA keeps successful because it relies on behavioral features extracted by the Bytecode. As a matter of fact, LEILA detection is realized with logic formulae that describe (and so catch) some characterizing behaviors extracted from the Bytecode. A direct benefit is that many code obfuscation techniques are transparent to the Bytecode, as well as the “nop” insertion: “nop” are not included in the Bytecode.

Table 8 shows the comparison between LEILA and seven current well-know antimalware solutions. This comparison is performed on the original and morphed versions of the samples. The table reports the number of samples correctly identified as belonging to the right family. LEILA achieves the best identification performance if compared with the other antimalware solutions (with both the original and morphed samples). These results reveal that a model-checking-based method is able to recognize the morphed samples, too.

Finally, Table 9 shows, for each family involved in the experiment, the details about the number of samples identified by all the considered rules. More precisely, given the formula \( \phi_F \), characterizing the family \( F \), the table shows how many apps satisfy the formula, i.e. belong in the family \( F \). For instance, the Opfake formula (i.e., the formula \( \phi_{Opfake} \)) is able to:

- correctly identify 46 apps of the Opfake family;
- incorrectly identify 1 app of the GinMaster family;
- incorrectly identify 1 app of the Ransomware family;
- incorrectly identify 2 apps of the Plankton family;
- correctly identify no app of FakeInstaller family;
- correctly identify no app of HummingBad family;
- correctly identify no app of Trusted;
- correctly identify no app of Other Malware.

We highlight that all the considered formulae are not true on the trusted samples and on the other malware ones. This is symptomatic that the logic formulae are able to characterize the malicious behaviour of the considered malware families. Note that for the “trusted” samples and the “other malware” ones no formula characterizing their behaviour exists in the “Formula” column of Table 9, since in these cases we verify the conjunction of the negation of all the specified formulae. Furthermore, for the FakeInstaller (resp. HummingBad) family, no sample is misclassified, i.e., no sample belonging to the family is incorrectly recognized by

\textsuperscript{16} https://play.google.com/store
\textsuperscript{17} https://github.com/tiato/android-market-api-py
\textsuperscript{18} https://www.virustotal.com/
the formula $\varphi_F$, where the family $F$ is different from FakeInstaller (resp. HummingBad). In some cases, it is possible that the formula $\varphi_F$ identifies apps not belonging to the family $F$. The reason is that, as demonstrated in [30], those samples can share the malicious behaviour (i.e., $\varphi_{\text{FakeInstaller}}$ is true on 4 samples of the Opfake family). In other words, the formula is verified also in the apps of other families because the corresponding behavior occurs in those families. It is not infrequent that different families can share single behaviors. Thus, the problem is to choose the overall set of rules that identifies the single family.

#### 4.3 Performance Analysis

We evaluated the CPU load of a machine running LEILA, with the aim to evaluate the LEILA overhead from a computational point of view.

We resort to the Windows Management Instrumentation Command-line (WMIC) tool in order to obtain information about the machine carrying out the experiment and to gather the CPU load measurements. Basically, WMIC is a command-line and scripting interface that simplifies the use of Windows Management Instrumentation (WMI) and systems managed through WMI. We considered a tool working at the level of operating system instead of one working at the Virtual Machine level (i.e., JProfiler), because we need to collect also the time required to execute CAAL (that cannot be measured by a Java virtual machine profiler). WMI is installed by default on Windows desktop and server environments and it is useful to extract several data about the machine under analysis i.e., about the CPU and the architecture.

In order to take the measurements, we considered the following command: “wmic cpu get loadpercentage /every:1”: the “/every:1” parameter allows us to take the CPU load percentage every 1 second. We measured the CPU load in different phases i.e., the CPU load when LEILA is not running (idle), the decompilation phase (dex2jar), the jar to Bytecode process performed by the CEL library (Bytecode), the building automatons phase (automaton) and, finally, the verification phase (verification). We collected the measurements for all these phases every second on 100 different apps of different sizes (from 100 KB to 10 MB) and we computed the average between the measurement points of the same phase of the considered apps. The average CPU load in the idle phase is 7%, in the dex2jar one is 9%, in the Bytecode one is 8%, in the automaton one is 11% and finally in the verification one is equal to 23%. We highlight that the most representative LEILA phases (i.e., the automatons and the verification ones) are the less CPU intensive if compared with the other phases. In addition, if we consider that the CPU load of the machine under analysis in idle is equal to 7% we can state that the automaton phase requires the 3% of CPU load (11%-7%) while the verification one the 16% (23%-7%) of total CPU load average.

In addition, in order to measure the LEILA performances from a temporal point of view, we considered the System.currentTimeMillis() Java method that returns the current time in milliseconds. Table 10 shows the performances of our method (both the CPU load and time point of view). These two values are the average times, i.e., they are computed as the total time employed by LEILA to process the samples divided the number of samples evaluated.

The machine used to evaluate the LEILA CPU load has these characteristics: an Intel Core i3 CPU 540@3.04 GHz (information obtained using the following WIMC option: “wmic cpu get name, CurrentClockSpeed”), with Microsoft Windows 7 Ultimate Service Pack 1 64 bit on board (“wmic OS get Caption,CSVersion,OSArchitecture”). The CPU presents 2 cores and 4 logical processors (“wmic cpu get NumberOfCores,NumberOfLogicalProcessors”).

The times we have obtained seem to be still high for an usage in the real world, nevertheless LEILA is intended to be used to perform antimalware analysis on apps marketplace, before the publication of an app. The times could be improved a lot, considering the deployment on appropriate servers of the marketplace’s back end running LEILA; and as the analysis will not run on the device, LEILA will not affect the usability of the device.

### 5 Discussion

In the following subsections we discuss the distinctive features and the limitations of our method.

#### 5.1 Detailed discussion of the novelties

We can point out some key novelties of the proposed approach.

- We provide an automatic dissection of a mobile application, since checks whether an app performs a specific behaviour, with an additional benefit: the localization into the code of those instructions that implement the actions that determine the malicious behaviour. Our solution is able to identify the exact position of those instructions characterizing the malicious behaviour. The method indicates the class file(s) where the rule is verified, with a precision at method level. This is a very novel result in the malware analysis and it could be used by the current antimalware to build a new type of birthmark which is expressed by a behaviour, i.e., a set of rules, instead of a sequence of bytes, i.e., a signature. Fig. 6 shows the name of the classes involved for the satisfaction of the formula. In details, LEILA also reports the list of the called and the calling methods. In Fig. 7 we have graphically shown only a part of that list. The complete representation is omitted in the figure to save space.

A first attempt to localize the payload can be found in [31]. HookRanker tool provides ranked lists of potentially malicious packages taking into account the way in which malware behaviour is triggered. HookRanker is able to perform a dissection of a sample under analysis at the granularity level of packages, while LEILA analysis localizes the malicious behaviour at a lower level;

- in addition to the payload localization, we can catch the events that trigger the malicious payload. We focus on this aspect because at the best of authors’ knowledge the current antimalware technologies are able only to delete the detected samples: our method can be used as a sanitizer tool i.e., a method able to make the payload not activable;
TABLE 6: Results of the evaluation. The first column represents the logic formula of all the families involved in the experiments. The second and the third column show respectively the number of samples belonging and not belonging to the family associated to the formula. The fourth column shows the number of trusted samples. All the other columns contain True Positives, False Positives, False Negatives, True Negatives, Precision, Recall, F-measure, and Accuracy.

<table>
<thead>
<tr>
<th>Formula</th>
<th>#Malware ∈ Family</th>
<th>#Malware ∉ Family</th>
<th>#Trusted</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>Opfake</td>
<td>50</td>
<td>300</td>
<td>50</td>
<td>46</td>
<td>4</td>
<td>4</td>
<td>346</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.98</td>
</tr>
<tr>
<td>GinMaster</td>
<td>50</td>
<td>300</td>
<td>50</td>
<td>47</td>
<td>2</td>
<td>3</td>
<td>348</td>
<td>0.95</td>
<td>0.94</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>Ransomware</td>
<td>50</td>
<td>300</td>
<td>50</td>
<td>48</td>
<td>9</td>
<td>2</td>
<td>341</td>
<td>0.84</td>
<td>0.96</td>
<td>0.84</td>
<td>0.97</td>
</tr>
<tr>
<td>Plankton</td>
<td>50</td>
<td>300</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>350</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FakeInstaller</td>
<td>50</td>
<td>300</td>
<td>50</td>
<td>50</td>
<td>2</td>
<td>0</td>
<td>348</td>
<td>0.89</td>
<td>0.86</td>
<td>0.87</td>
<td>0.97</td>
</tr>
<tr>
<td>HummingBad</td>
<td>50</td>
<td>300</td>
<td>50</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>350</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 7: An Example of Payload Localization: the involved methods

TABLE 7: Resilience to the Obfuscation Techniques. The first column represents the families involved in the experiments. The second and the fourth column show respectively the number of samples belonging to the original and morphed family. The third and the fifth column report the number of samples correctly identified by LEILA (TP), respectively in the original form and in the morphed form.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Original Samples</th>
<th>Morphed Samples</th>
<th>TP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opfake</td>
<td>50</td>
<td>50</td>
<td>46</td>
</tr>
<tr>
<td>GinMaster</td>
<td>50</td>
<td>50</td>
<td>47</td>
</tr>
<tr>
<td>Ransomware</td>
<td>50</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td>Plankton</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>FakeInstaller</td>
<td>50</td>
<td>50</td>
<td>43</td>
</tr>
<tr>
<td>HummingBad</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

- another important novelty of our approach is the use of model checking techniques for Android applications to identify malware families with the management of the method calls. All these aspects have never been treated jointly, to the best of the authors’ knowledge. In addition, model checking, besides proving that a program is not correct, can also help debugging, making it possible to locate errors. In fact, if a property does not hold, the model checking algorithm generates a counter-example, i.e., an execution trace leading to a state in which the property is violated. This ability to generate counter-examples, which can be exploited to pinpoint the cause of an error, is the main advantage of model checking, as compared to other well-known techniques for software verification, as abstract static analysis. Thus, our formal model can be used (i) to identify Android malware family; (ii) to perform code obfuscation detection; (iii) to help in code analysis; and (iv) to get insight on liveness and safety properties of the code, using a single model-checking based framework;
- we propose a behaviour-based detection as an approach to mitigate the effect of code obfuscation techniques. This is against the backdrop of the limitations of signature-based approach commonly in use nowadays by most antivirus engines. Signature-based approaches require: the analysis of signature strings, to obtain signatures of malware, and storing them in databases with which strings of any new attacks are compared for possible detection. New attacks whose signatures have not been previously obtained and stored in databases can not be detected. Our behaviour-based approach is able to handle new malware payloads and detect real malware found in the wild, when the new malware exhibits a behavior similar to that of an analyzed payload;
- our approach can manage multi-threading in Android applications being based on CCS with its intrinsic notion of concurrency. Using the CCS parallel operator we can produce automatically all the interleavings of the executions of the threads, differently from the dynamic analysis where different executions of the same software systems will generally produce different traces depending on the execution
Our method derives CCS processes from the Java Bytecode and not directly on the Java source code. This has several advantages:

1. It permits to define a wide range of properties of program executions, such as security properties and more generally, invariant, liveness or safety properties.
2. It is considered malware if it contains a sequence of instructions that is matched by a regular expression. Recently, it has been shown that such malware detectors can be easily defeated using simple program obfuscations [32], that are already being used by the malware writers. Since the pattern-matching algorithm is not resilient to slight modifications, these malware detectors must use different patterns to recognize two malware instances that are minor obfuscations and modifications, as shown in [20] and in the previous section.

Our approach overcomes this kind of problem, being resilient to minor obfuscations and modifications, as shown in [20] and in the previous section.

On the other side, our method requires that a set of logic rules be defined for each family. Actually, they have been defined manually by inspecting very few samples for each family, but we are studying approaches to automatically deduce the logic rules. Some hints are given in Section 5.3.

Our method leverages the Bytecode representation of Java programs produced by a Java compiler. Performing Android malware families detection on the Bytecode and not directly on the Java code has several advantages:

1. The behaviour of a program. Commercial malware detectors use simple pattern matching approaches to malware detection. A code is considered malware if it contains a sequence of instructions that is matched by a regular expression. Recently, it has been shown that such malware detectors can be easily defeated using simple program obfuscations [32], that are already being used by the malware writers. Since the pattern-matching algorithm is not resilient to slight modifications, these malware detectors must use different patterns to recognize two malware instances that are minor obfuscations and modifications, as shown in [20] and in the previous section.

2. The fundamental drawback of a pattern-matching approach to malware detection is that it is purely syntactic and it ignores the behaviour of a program. Commercial malware detectors use simple pattern matching approaches to malware detection. A code is considered malware if it contains a sequence of instructions that is matched by a regular expression. Recently, it has been shown that such malware detectors can be easily defeated using simple program obfuscations [32], that are already being used by the malware writers. Since the pattern-matching algorithm is not resilient to slight modifications, these malware detectors must use different patterns to recognize two malware instances that are minor obfuscations and modifications, as shown in [20] and in the previous section.

On the other side, our method requires that a set of logic rules be defined for each family. Actually, they have been defined manually by inspecting very few samples for each family, but we are studying approaches to automatically deduce the logic rules. Some hints are given in Section 5.3.

Our method leverages the Bytecode representation of Java programs produced by a Java compiler. Performing Android malware families detection on the Bytecode and not directly on the Java code has several advantages:
• independence of the source programming language (e.g., Kotlin Language\(^\text{19}\));
• detection of malware families without decompilation even when source code is missing;
• ease of parsing a lower-level code;
• independence from some obfuscation techniques.

Another relevant feature of our method is the detection of payload fragmentation, as shown in Fig. 8. In particular, the payload is related to formula \( \varphi \) in Table 3. The two snippets of code belong to the Opfake family. The payload is highlighted by the big red rectangle both in Sample A and in Sample B. The difference is that, in sample A the payload occurs in the for instruction, while in the sample B the payload is located in the sendSMS method that is invoked in the for instruction. According to the formula \( \varphi \) of Table 3, a sample is infected by OpFake malware if the actions involved in the formula \( \varphi \) occur many times in the sample’s code. By analyzing in isolation only the body of the method of the Sample B, it is not possible to recognize the malicious payload, since the malicious actions occur only one time. To recognize the Sample B as malware, it is needed to implement the method call, as done in our method.

As a simple example, Fig. 9 shows a fragment of the automaton generated by the code of Sample B in Fig. 8. Our model is able to characterize the code behaviour. In fact, the automaton mimics both the cyclic behaviors of a code and the method calls. By using a model checker tool, it is possible to verify, in an automatic manner, if the model exhibits a specified behaviour. The model checker verifies if the behaviour specified by the formula is true (or not) on the model. Both the model and the formula are the inputs of the model checker tool. The advantage is that no responsibility of the human verifier of the system is required, i.e., user intervention is not needed. Using this kind of approach is simple and automatic to verify every behaviour of a model. Considering the example of Fig. 9, the formula \( \varphi \) is true on the automaton, recognizing in this way the malicious payload.

5.2 Motivation of the use of the selective mu-calculus logic

It is well-known that most current formal methods are mainly applicable to small-scale applications, but do not scale up well. However, the state explosion problem, typical of model checking, can be a real problem when analyzing Android applications, since the produced CCS specifications can have a large number of states and transitions. In fact, our new formal model takes into account also methods invocations specified as a parallel composition. To alleviate the state explosion problem, we use the selective mu-calculus logic to express Android malware families. In this way we improve both the performance, i.e., the speed at which the model checker returns its results, and its scalability, i.e., the extent to which the model checker can manage increasingly large apps. In the following, we discuss the results of these tests.

The basic characteristic of the selective mu-calculus is that the actions relevant for checking a formula \( \varphi \) are those ones explicitly mentioned in the modal operators used in the formula itself. Thus, we define the set \( O(\varphi) \) of occurring actions of a formula \( \varphi \) as the union of all sets \( K \) and \( R \) appearing in the modal operators \( (\bigcup K_R \varphi, K_R \varphi) \) occurring in \( \varphi \).

In the work \([16]\) \( \rho \)-equivalence is defined, formally characterizing the notion of “the same behavior with respect to a set \( \rho \) of actions”:

\[ \text{two transition systems are } \rho\text{-equivalent if a } \rho\text{-bisimulation relating their initial states exists.} \]

The definition of \( \rho \)-bisimulation is based on the concept of \( \alpha \)-ending path: an \( \alpha \)-ending path is a sequence of transitions labelled by actions not in \( \rho \) followed by a transition labelled by the action \( \alpha \) in \( \rho \). Two states \( S_1 \) and \( S_2 \) are \( \rho \)-bisimilar if and only if for each \( \alpha \)-ending path starting from \( S_1 \) and ending into \( S_1' \), there exists an \( \alpha \)-ending path starting from \( S_2 \) and ending into a state \( \rho \)-bisimilar to \( S_1' \), and vice-versa. In \([16]\) it is proved that

\[ \text{two transition systems are } \rho\text{-equivalent if and only if they satisfy the same set of formulae with occurring actions in } \rho. \]

As a consequence, a formula of the selective mu-calculus with occurring actions in \( \rho \) can be checked on any transition system \( \rho \)-equivalent to the standard one. Thus, improvements in model checking can be obtained by minimizing the transition system with respect to the actions in \( \rho \). Obviously, the degree of reduction depends on the size of \( \rho \) with respect to the size of the whole set \( A \) of actions.

Consider again the transition systems illustrated in Fig. 3. \( S_1 \) is \( \{a,b\} \)-equivalent to \( S_2 \). The two transition systems give the same value for the formulae containing only actions in \( \{a,b\} \). In particular, they satisfy \( \psi_2 \), while they do not satisfy \( \psi_1 \). Note that \( O(\psi_1) = \{a,b\} \) and \( O(\psi_2) = \{a\} \subseteq \{a,b\} \). On the contrary \( S_2 \) is not \( \{a,b\} \)-equivalent to \( S_3 \), since it can perform an action \( b \) without performing an action \( a \). In \([33]\) a method is defined, which, given a process \( p \) and a set of interesting actions \( \rho \) occurring in a formula \( \varphi \) to be checked, transforms \( p \) into another process \( q \), corresponding to a smaller transition system than that of \( p \), on which \( \varphi \) can be equivalently checked. The tool is based on a set of syntactic transformations rules. A prototype tool which implements the algorithm has also been defined; it is written in SICStus Prolog\(^{20}\).

We discuss our experience with the above method. The aim is to evaluate the performances of the approach and compare it against the standard formal verification environment. First, we apply our tool to transform the specification into a smaller one, where the reduction is driven by the actions occurring in the formulae. Then, we check the properties on the reduced specification. All properties have been checked on reduced CCS processes; below we will show only the application on a single property, just to give the reader the flavor of the approach followed.

Consider the following property:

\[ \psi = (\text{invokesqlEscapeString})_0 (\text{newjavalangStringBuilder})_0 (\text{invokeappend})_0 (\text{invokeexecSQL})_0 (\text{invokerawQuery})_0 (\text{invokemoveToNext})_0 (\text{invokeclose})_0 \tt \]

\( \psi \) is a selective mu-calculus logic formulation characterizing the Gin Master family. It holds that the set of interesting actions of the above property is:

\[
\rho = O(\psi) = \{ \text{invokesqlEscapeString}, \text{newjavalangStringBuilder}, \text{invokeappend}, \text{invokeexecSQL}, \text{invokerawQuery}, \text{invokemoveToNext}, \text{invokeclose} \}
\]


\(20\). http://www.sics.se/isl/sicstus.html
Table 11 shows the number of states and transitions of the \( \psi \)-reduction, i.e., is the reduced CCS process obtained by applying the property-based method with \( p \). It is worth noting that the reduction we perform of the states and transitions is significant. Note that also for all the other properties we obtain great reductions. Obviously, a reduction of the state-space implies also a reduction of the time employed by model checker to check the formulae. In most cases we obtain a reduction more than 95%. In general, the usefulness of our tool depends both on the number of actions occurring in the formula and on the structure of the formula to be checked.

5.3 Limitations

This section discusses the limitations of our method.

- **Non-Detectable by Static Analysis obfuscation techniques.** As stated in [34], obfuscation transformations can be classified into two classes: (i) those which result in variants that can still be detected by advanced static analyses involving data-flow and control-flow analysis, like for example, call indirections, code reordering; (ii) and those which can render malware undetectable by static analysis, like for example, reflection [35], Bytecode encryption. Our method, being static, cannot manage obfuscations belonging to the undetectable by static analysis type. In fact, LEILA is not able to resolve the target of reflected method calls if these targets are stored encrypted or created dynamically.

- **Virtual calls.** LEILA does not handle dynamic runtime behaviors of Java programs, such as virtual method invocations. In presence of inheritance, the problem of resolving virtual methods is not managed, since our method flattens all possible flows into a single one that is based on the static declaration of the reference used to access the object. However, our method also checks all the other methods not occurring in the analyzed flow (i.e., never referred in the Bytecode), since they are specified as a non-deterministic choice in our model. Obviously, this solution breaks down the complexity of our method, reducing the number of virtual call targets, but may introduce incorrectness and incompleteness, even if in reality, as shown in the “Results” section, this has not massively occurred;

- **Manual definition of the formulae.** The logic rule-set characterizing a malware family needs to be designed and defined. Writing the correct rules can be a rather complex task. Currently, we are studying an approach based on clone detection to automatically deduce the logic rules. Another viable solution could be the use of natural language processing tools for the analysis of technical reports. Furthermore, to help the designer to write simple temporal properties, it is possible to use the user-friendly interface (UFI) developed by one of the authors and al. in [36]. UFI has the aim of simplifying the writing of the logic properties;

- **Performances.** LEILA is currently a research prototype tool whose main aim is to demonstrate the effectiveness of our method in malware families identification, thus the performances are not the core issue of development of LEILA. Although the times we have obtained seem to be still high for an usage in the real world, it should be considered how LEILA is intended to be applied. LEILA was conceived to perform antimalware analysis on apps marketplace, before the publication of an app. Consequently appropriate architectures can be deployed on the marketplace back end in order to improve performances, and, in any case, it does not affect the device’s usability;

- **Native code and cross-site-scripting.** Another weakness of our method is represented by the fact that LEILA prototype considers in its analysis the Bytecode extracted by the dex file of the mobile application excluding the native code and the javascript code possibly embedded into the application.

Overall, we think that these limitations do not severely restrict
the applicability of our method. Our method, as the experiment demonstrated, should be considered as a good check to get a reasonable trust in the correctness of detecting malicious behaviour and, especially, localizing the payload within the code.

6 RELATED WORK

Previous researches have demonstrated that the formal verification can help to detect malware. In the following we describe malware detection methodologies employing formal methods.

In [37] an extension CTPL (Computation Tree Predicate Logic) of CTL (Computation Tree Logic) has been proposed to express some malicious properties that are used to detect the malicious behavior of thirteen Windows malware variants using as dataset a set of worms dating from 2002 to 2004. Further, Song and Touili extend CTPL to SCTPL for better description on stack-based actions of viral behaviors [38].

Song et al. [38] present an approach to model Microsoft Windows XP binary programs as a PushDown System (PDS). They evaluate 200 malware variants (generated by NGVCK and VCL32 engines) and 8 benign programs.

The tool PoMMaDe [39] implements a custom-made model-checking algorithm that it is able to detect 600 real malware for PCs, 200 malwarees generated by two malware generators (NGVCK and VCL32), and proves the reliability of benign programs: a Microsoft Windows binary program is modeled as a PushDown System which allows to track the stack of the program.

First of all, our main objective is to reuse existing model checkers avoiding the design of custom-made model checker. Model checkers are extremely sophisticated programs that have been crafted over many years by experts in the specific techniques employed by the tool. A re-implementation of the algorithms in these tools could likely yield worst performance. Secondly, we focus on Android Mobile malware, instead on PC malware. Thirdly, our approach is able to identify the localization of the malicious payload, especially if it is broken up into several methods, classes and packages of the application under analysis. Finally, we propose a method for the identification of family malware.

Song and Touili in [40] apply model checking to Android apps. However, they model mobile applications using a PushDown System in order to discovery only private data leaking working at Smali code level.

Jacob and colleagues [41] provide a basis for a malware model, founded on the Join-Calculus: they consider the system call sequences to build the model. It is a theoretical paper that shows the intractability of the malware detection problem providing a formalization of malware in a Turing complete model of computation.

Recently, in [42] the authors have built a general framework named DROIDPF upon Java PathFinder (JPF), towards model checking Android apps. DROIDPF focuses on common security vulnerabilities in Android apps including sensitive data leakage involving a non-trivial flow- and context-sensitive taint-style analysis. Differently, LEILA focuses on the detection of malicious family identification.

The possibility to identify the malicious payload in Android malware using a model checking based approach has been already explored by some the authors of these papers [43], [20], [21]. Starting from the payload behavior definition the authors formulate logic rules and then test them by using a real world dataset composed by real world Android malware samples. The main
difference with the method presented in this paper is represented by the fact that the cited methodologies are not able to catch a malicious action divided in different small parts of the application, i.e., the payload using these methods is recognizable only if it is fully included in a single method.

6.1 A comparison with Machine Learning based approaches

In last years machine learning was largely employed to generate the classifiers able to discriminate malware samples from the goodware. The main task of machine learning based approaches is represented by the feature(s) selection phase, once extracted the feature representing the malware payload, the training process (and the consequently testing phase) is automatically provided by well-known algorithms. In the following we briefly describe the most representative efforts done by researchers in this direction.

Canfora et al. [44]: the authors have proposed a machine learning based approach to identify Android malware families. The features are extracted using two techniques: the Hidden Markov Model (HMM) and the Structural Entropy. The first technique considers HMMs with the aim to determine if an application is similar to malware, while the second one is based on the estimation of structural entropy of an executable (i.e., the .dex file) using the wavelet transform to obtain a representation of the entropy series. The features gathered using the two techniques are used to classify malware families with following algorithms: J48, LabTree, NBTree, RandomForest, RandomTree and RepTree.

Madam [45]: MADAM is an Android malware detector that concurrently monitors the device under analysis both at kernel and at user level; machine learning techniques are employed to distinguish between standard and malicious behaviors. At kernel-level, features related to the following system parameters are extracted: system calls, running processes, free RAM and CPU usage. At user-level, features related to the following system parameters are considered: idle/active, key-stroke, called numbers, sent/received SMS and Bluetooth/Wi-Fi analysis. The developed prototype is able to detect several real-world malware applications found in the wild with a low number of false positives.

ICCDetector [46]: the authors have proposed a machine learning based detection method, called ICCDetector. This approach recognizes a malware using its Inter-Component Communication (ICC) patterns. Starting from an APK file the authors model how the ICC mechanism. At the end of this process the ICC patterns of the APK file are defined. Afterwards, they trained ICCDetector with the ICC patterns belonging to benign and malicious applications. According to the ICC patterns identified, ICCDetector further has classified the malicious application in five new malware categories.

DroidClone [1]: the authors use a clone detection to recognize Android malware variants. In order to find code clone MAIL language is used, a new Malware Analysis and Intermediate Language. MAIL is used to specify control flow patterns which reduces the effect of obfuscations and provides automation and platform independence.

DroidScribe [47]: the authors use machine learning techniques to automatically classify Android malware samples into families observing their runtime behavior i.e., performing a dynamic analysis. They have considered several feature sets in order to increase the accuracy: the first one, based on IP, port and network traffic size; the second one, based on file name and type, name classes; the third one, based on method name and parameters and the last one, based on user permission to execute a file. The trained classifiers fuses Support Vector Machines with Conformal Prediction obtaining an accuracy equal to 84%.

StormDroid [48]: StormDroid is a machine learning based framework for malware detection. The authors have defined two kind of features: well received features and newly-defined features. The first one are the required permissions by an application and the sensitive API calls. The second one are the newly defined “Sequences” and Dynamic behaviours. The Sequence is the quantity of sensitive API calls (i.e., the number of sensitive API calls requested by the malicious applications and the benign ones, respectively). Dynamic behaviour analysis aims to observe the malicious activities performed by an application, for instance a malicious app request sensitive information more frequently than the benign one. The top features of dynamic behaviors used in this work are network activity, file system access and interaction with the operating system. In order to collect these features StormDroid performs an hybrid analysis, it employs both static and dynamic analysis.

DroidSieve [49]: the proposed approach is an Android malware classifier based on static analysis. The authors identify two high-level classes: resource-centric features are derived from resources used by the application under analysis, while syntactic features are derived from the code and metadata of the mobile application. Furthermore, the authors consider obfuscation-invariant features and artifacts introduced by obfuscation mechanisms commonly used by malware writers.

Nix and Zhang [50]: the authors have proposed a classification methods based on Deep Neural Networks (DNNs). In particular, they design a Convolutional Neural Network (CNN) to detect malware. The features used in the classification are the Android system-API calls. They extract these feature without any execution of the code. Authors track the program instructions following the all possible execution of the code without computing any state information of the program. Authors also compare with the recurrent neural network with Long Short Term Memory (LSTM) and other n-gram based methods.

In order to highlight the novelty of LEILA, we compare our tool with the above approaches. There are plenty of possible criteria for such comparison. We formulate seven essential criteria:

- C1: Approach. The approaches can be static, dynamic or hybrid;
- C2: Family Identification Capability. The approach is able to categorize the malware sample in the right family, i.e., to recognize the family which the malware belongs to;
- C3: Resilience to Code Obfuscation. The approach relies on common code obfuscation techniques;
- C4: Behavioural Analysis. The approach performs some behavioural analyses on the sample under analysis;
- C5: Statement-level Localization. The approach performs malicious payload localization at lower level i.e., it provides some information about the malicious payload and its position in the code of the sample;
- C6: Completely Automatic. The approach performs its analysis without human involvement;
- C7: Multi-threading. The approach supports multi-thread analysis. This analysis is important since Android is event-based and it has several components able to interact with each other.

Table 12 highlights the differences between LEILA and the other considered approaches. In Table 12 the rows are the...
considered approaches and the columns are the formulated criteria for the comparison. Note that, whether a criterion is not explicitly specified in the work, the Table 12 presents the label N/A standing for information Not Available. Table 12 shows that LEILA is the only method able to both localize the malicious behaviours and to perform multi-thread analysis.

Furthermore, the main criticisms related to machine learning techniques are the following: (i) the usage of an extensive training set in order to reach high precision and recall, (ii) if the features related to a malware behaviour are not represented in the training set, the application will be not successfully recognized, (iii) the high false positives rate, (iv) the need to continually update the training set in order to catch the new threats.

Differently, our method requires a small subset of samples in order to extract the logic rules representing the malware behaviour and it has a few false positives. In addition, with the features extracted it is not possible to build classifiers able to localize the malicious payload, as a matter of fact the aim in these cases is the binary malware detection (i.e., the aim of the authors is to discriminate malware applications from legitimate ones).

6.2 A comparison with Flow Analysis Approaches

Differently from the previous techniques there is another kind of approach related to the detection of the ICC issue. The main differences with our work is represented by the fact that these tools do not identify malicious payload, but the source and the destination of exfiltrated data, for this reason they are not considered as malware detector but as tools that help the security expert to analyze if there is a connection from source of private and confidential data to the sink (i.e., the channel used to send data out of device for instance, a method writing the information to a socket). The following tools perform ICC detection.

Amandroid [11] performs an ICC analysis to detect ICC leaks, and has been developed concurrently with IccTA. Amandroid needs to build an Inter-component Data Flow Graph (ICDFG) and an Data Dependence Graph (DDG) to perform ICC analysis. It is basically a general framework to enable analysts to build a customized analysis on Android apps.

FlowDroid [10] adequately models Android-specific challenges like the application lifecycle or callback methods. It helps reduce missed leaks or false positives. Novel on-demand algorithms allow FlowDroid to maintain efficiency despite its strong context and object sensitivity.

Epicc [51] identifies a specification for every ICC source and sink. This includes the location of the ICC entry point or exit point, the ICC Intent action, data type and category, as well as the ICC Intent key/value types and the target component name. Note that where ICC values are not fixed Epicc infer all possible ICC values, thereby building a complete specification of the possible ways ICC can be used. The specifications are recorded in a database as flows detected by matching compatible specifications.

IccTA [52] uses a static taint analysis technique to find privacy leaks, i.e., paths from sensitive data, called sources, to statements sending the data outside the application or device, called sinks. A path may be within a single component or cross multiple components. To verify their approach, authors developed 22 apps containing ICC-based privacy leaks.

TaintDroid [53] is an extension to the Android operating system that tracks the flow of privacy sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors in realtime how these applications access and manipulate users’ personal data.

PCLeaks [54] performs data-flow analysis to detect potential component leaks, which not only include component hijacking vulnerabilities, but also component launch (or injection) vulnerabilities.

7 CONCLUDING REMARKS AND FUTURE WORK

Malware in last years is plagiarizing mobile users: threats are quickly becoming more and more aggressive in gathering sensitive and private user information and stealing credit. Current antimalware technologies are not able to protect users also when a malware is recognized by antimalware, by using trivial obfuscation techniques the attackers can easily elude the detection. In order to overcome these limitations, in this paper we propose LEILA, a model checking-based method able to identify Android malware introducing new features, as the localization of the malicious classes into the infected application, family and fragmented payload recognition. We evaluated LEILA using real-world Android malware belonging to six widespread families (Opfake, GinMaster, FakInstaller, HummingBad, Plankton and Ransomware), obtaining an accuracy ranging between 0.97 and 1. We produced morphed variants of the dataset we tested in order to demonstrate the effectiveness of LEILA in obfuscated malware identification, finding that even when signature-based antimalware are eluded by the obfuscated samples, LEILA is anyway able to recognize the malicious payloads.

The main disadvantage of the proposed technique is represented by the manual rule generation. This is the reason why we plan to design an approach to automatically deduce logic rules. As stated in Section 5.3, we are studying an approach based on clone detection to automatically deduce the logic rules.

As future work, we plan to test LEILA with other malware families, with particular regards with colluding malware [55] i.e., malicious applications with the ability to exploit covert or overt channels. In addition, we will improve our method in order to make ineffective the malicious payload once localized by LEILA.

Another trend in Android malware is the exploit of MultiDex applications i.e., applications that use two Dalvik Executable (DEX) files to deliver the final payload. The current antimalware and the state-of-the-art detectors fail to recognize this kind of malware as they focus on a single DEX file as well. Our approach can identify the malicious payload also in MultiDex applications.

In addition, we plan to evolve LEILA and make it able to inspect also ARM instructions (this requires to build the automaton related to the ARM libraries in order to verify whether the malicious payload is implemented). With regard to the cross-site-scripting (i.e., the JavaScript injection), considering that Android application can embed urls into the application code, a countermeasures in order to mitigate this kind of attacks can be represented by the usage of a sandbox to visit the url and download the malicious JavaScript in order to analyze it.

Finally, the efficiency of LEILA will be tested also on malware provided by Mystique [56], a framework to automatically generate malware with specific features. This will allow us to verify specific behavior on samples ad-hoc generated.

ACKNOWLEDGMENT

This work was partially supported by the H2020 EU funded project NeCS [GA #675320], by the H2020 EU funded project CISP [GA #700294].
The Authors thank Maurizio Viscusi and Raffaele Sanzari for helping in the LEILA implementation.

REFERENCES


Francesco Mercaldo received his master degree in computer engineering from the University of Sannio (Benevento, Italy), with a thesis in software testing. He obtained his Ph.D. in 2015 with a dissertation on malware analysis using machine learning techniques. The research areas of Francesco are software testing, verification, and validation, with the emphasis on the application of empirical methods. Currently, he is working as post-doctoral researcher at the Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche (CNR) in Pisa (Italy). He is also involved as lecturer in Database, Operating Systems (Bachelor Degree) and Software Security (Master Degree) courses at the University of Molise (Italy).

Vittoria Nardone obtained the MsC in Computer Engineering at the University of Sannio. Vittoria is a PhD student at the same University under the supervision of Antonella Santone. Her current research is focused on the application on formal verification methods, in particular on the model checking technique. She applies the following technique on the Android malware analysis environment in order to identify malicious behaviours.

Antonella Santone received the Laurea degree in Computer Science at the University of Pisa, Italy, in April 1993. In September 1997 she received the Ph.D. degree in Computer Systems Engineering at the Dipartimento di Ingegneria della Informazione, University of Pisa. She is an Associate Professor at the University of Molise since September 2017. She has been Assistant Professor at the University of Pisa from November 1998 to October 2001. She has been an Associate Professor at the Department of Engineering of the University of Sannio from November 2001 to August 2017. She was involved in several research activities and projects. Antonella Santone’s current research is focused on formal verification methods. Her research interests include formal description techniques, temporal logic, concurrent and distributed systems modelling, heuristic search, formal methods in systems biology and in software security. She has written more than hundred papers for international journals and conferences.

Corrado Aaron Visaggio obtained the PhD in computer engineering at University of Sannio (Italy). He is assistant professor of Software Security of the MsC in Computer Engineering at the University of Sannio, Italy. His research interests include: empirical software engineering, software security, and data privacy. He is author of more than 50 papers published on international journals, international and national conference’s proceedings, and books. He serves in many program committees and editorial boards of international conferences and journals.