Localization and Inhibition of Malicious Behaviors through a Model Checking based methodology

Mario Cimino and Gigliola Vaglini
Department of Information Engineering, University of Pisa, Pisa, Italy
{*mario.cimino, gigliola.vaglini*}*@unipi.it*

Keywords: Malware; Android; Security; Formal Methods; Process Algebras.

Abstract: Mobile malware is increasing more and more in complexity; current signature based antimalware mechanisms are not able to detect attacks, since trivial code transformations may evade detection. Furthermore, antimalware, when correctly label an application as malicious, are able to quarantine or delete the application, but not to allow the user to install and safely use it. Here we present a model checking based approach to locate and inhibit malicious behaviors: we suppose the specification of programs in terms of process algebra language LOTOS, malicious behaviors specified by temporal logic formulae, and define a method to retrieve, from the specifications, the description of the infected part of the program. We refer as example to some Android malware and derive LOTOS specification automatically from the Java Bytecode corresponding to Android's app. The method consists of a set of rules building the LOTOS processes mirroring the behavior of the malware possibly contained in the app; besides the description of the infected part of the code, we give also a way to block the malware attack, putting the basis to disinfect the application. The method can be applied at any level of complexity, so allowing the precise location of malicious behaviors.

1 INTRODUCTION

Malware, in order to evade the detection by antimalware, evolves and the majority of newly detected ones are simple variants of well-known code (Bailey et al., 2009; Hu et al., 2009; Jang et al., 2011) that maintain the typical behavior of the initial malware. As a matter of fact, attackers use to modify existing malicious code, by adding new behaviors or merging together parts of different existing malware's codes. Existing Android malware, for example, can be embedded in apparently benign programs (usually popular apps) with repackaging (Zhou and Jiang, 2012): malware authors locate and download popular apps, disassemble them, enclose malicious payloads, re-assemble and then submit the new apps to official and/or alternative Android markets. For this reason the repackaged malware applications are legitimate application with a small portion of malicious behaviors injected by the attacker. Usually the malicious payload exploits the same permissions used by the legitimate application in order to act in a silent way. Researchers (Zhou and Jiang, 2012) grouped malware in so called *families*, where a family defines a set of behaviors common to all its members.

Current literature provides several approaches to detect Android malware (Canfora et al., 2013; Arp et al., 2014), but malware that is well recognized by detectors turns it in a version that is not anymore recognized by the most detectors after simple code transformations (Canfora et al., 2015). Starting from these considerations, new techniques are due which are able to effectively recognize the fundamental malicious behavior common to all malware codes of a given family. In particular static techniques are useful since they do not require the code execution (Schmidt et al., 2009).

In this paper we investigate whether model checking could help to properly locate malicious payload, where each payload can be individuated by a temporal logic specification of the basic family malicious behaviors. This specification could help also in defining the way in which the malware can be inactivate thus making the code safely usable. The method we present applies to LOTOS (Bolognesi and Brinksma, 1987) specifications and to formulae of the temporal logic selective mu-calculus (Barbuti et al., 1999b; Barbuti et al., 2005); the outcome of the method is a new LOTOS specification satisfying those formulae. Then it is required a translation from Java Bytecode, in the case of Android's app: to LOTOS, this translation is really very easy when not considering values, but only the method calls. The use of the selective mu-calculus, which is an equi-expressive variant of mu-calculus (Stirling, 1989), helps in obtaining a description of the affected behavior of the system containing only the dangerous events. We take examples of formulae describing Android malware from (Battista et al., 2016) and here we use their negation. More precisely, given a LOTOS component P and a malware temporal logic specification, described by a selective mu-calculus formula ψ, the technique constructs a LOTOS component X and a set of events B such that the parallel composition of P and X with synchronization on the events in B satisfies $\varphi = \mathit{not}\ \psi$. At the end X contains the description of the events of P potentially dangerous; actually, X shows also a way to block the effect of the malware, in fact the composition of P and X verifies $\varphi = \mathit{not}\ \psi$, i.e. the absence of the malware.

The technique can be applied in a component-based program development at various level of complexity of the system: for each level the description of the contained malware is obtained. In fact, distributed systems are typically too large and complex to perform verification after construction as a monolithic, one-time process. The scale of these systems demands the development of methods for security assurance that are more modular, compositional, and incremental. Modularity is needed so that formal methods can be applied to individual system components rather than requiring that the verifier confronts the entire complexity of the system at once. Modularity also demands compositionality: if separately verified components are combined to form a larger system, the desired security properties of the larger system should follow from the formally verified properties of the individual component modules, rather than requiring that modules be verified again for their new context. In this regards, the presented method guarantees that the properties of the code independent from the malware are maintained by the modifications. In fact, the new process $(P\ |[B]|\ X)$ preserves the properties of P, but only if they do not "contrast" with φ: roughly speaking, if χ is the property that P verifies, $(P\ |[B]|\ X)$ verifies $\chi \wedge \varphi$ only if χ and φ do not contain existential and universal conditions on the same event. In this case, we say that the formulae are not interfering. The class of non-interfering formulae takes into account many properties that can be interesting in practice, such as "safety" and "liveness" properties. Intuitively, a safety property (like for example "between two successive events a no event b is allowed") expresses the fact that *nothing bad will ever happen*, while a liveness property (like for example "if a ever happens then eventually b will happen later") expresses the fact that *something good must eventually happen*. However, the non-interference criterion can rule out some mixture of safety and liveness formulae.

The paper proceeds as follows: comparisons with related work are made in Section 2. Section 3 is a review of the basic concepts of formal methods, while Section 4 describes our methodology. In Section 5 results are discussed and conclusions are drawn, while Section 6 concludes and gives some hints on future work.

2 RELATED WORK

Some works about malware detection exist where formal methods and static analysis are used. Authors in (Kinder et al., 2005) introduce the specification language CTPL (Computation Tree Predicate Logic) which extends the logic CTL, and describe an efficient model checking algorithm.

Song et al. (Song and Touili, 2001) present an approach to model Microsoft Windows XP binary programs as a PushDown System (PDS). In particular, the tool PoMMaDe (Song and Touili, 2013) uses PDS to track the stack of the program; while Song et al. (Song and Touili, 2014) model mobile applications by a PDS to discovery private data leaking.

In (Jacob et al., 2010) a basis for a malware model is provided founded on the Join-Calculus: the process-based model supports the notions interaction, concurrency and non-termination to cover evolved malware. They consider the system call sequences to build the model.

An approach aiming at the derivation of algebraic specifications from the Java code of an Android's app can be found in (Battista et al., 2016; Mercaldo et al., 2016c; Mercaldo et al., 2016a; Mercaldo et al., 2016b): the result is a CCS (Milner, 1989) specification.

The problem we are trying to solve can be reduced to that of the synthesis of concurrent programs from temporal logic specifications; in particular, in (Pnueli and Rosner, 1989) the new program is obtained as side-effect of the proof of the logic formula by a theorem prover. The time complexity of such algorithms is exponential in the size of the old and the new process.

In (Santone and Vaglini, 2003) a tableau-based method is proposed to integrate a LOTOS process with a new concurrent one, so that it became able to satisfy a given temporal logic formulae; in this case all potentialities of the existing module are exploited, and in the new process only the actions (or sequence of actions) not included in it, or differently specified, are added. The method requires exponentiality but only in the nesting recursion level of the formula when different recursive operators are interwined; while, the use of alternation free formulae leads to a polynomial cost of the construction. Moreover the method exploits the potentialities of existing process to reduce the dimension of the part newly designed. In the present context, this is a very valuable characteristic, since we try to describe and locate exactly the malware present in the code and not other not dangerous actions.

3 PRELIMINARIES ON FORMAL METHODS

In this section we introduce the basic concepts of formal methods. For applying formal methods, we need two things:
1. A precise notation for defining systems: For this purpose we can use directly the notion of automaton describing the state transitions of the system, or, alternatively, we can represent that automaton in the more compact form of a process in some algebraic language.
2. A language for defining properties: We consider concurrent and distributed systems, then a suitable language is temporal logic.

3.1 The specification language LOTOS

Basic LOTOS is the version of LOTOS without value-passing widely used in the specification of concurrent and distributed systems to describe the synchronization aspects of the system. We assume the reader familiar with Basic LOTOS, and so we recall only some main concepts. The reader can refer to (Bolognesi and Brinksma, 1987) for further details. From now on we write LOTOS instead of Basic LOTOS. A LOTOS program is defined as:

```
process ProcName := P
      where ℰ
endproc
```

where `P` is a *process*, `ProcName := P` is a *process declaration* and $\mathcal{E}$ is a *process environment*, i.e. a set of process declarations. A process is the composition, by means of a set of operators, of a finite set $\mathcal{A} = \{i, a, b, ...\}$ of atomic *actions*. The action i is called the *unobservable action*. Only the operators useful for our purpose are presented in the following syntax for LOTOS processes:

$$P ::= \texttt{stop} \mid \alpha;P \mid P[]P \mid P|[S]|P \mid X$$

where X ranges over a set of process names, α ranges over $\mathcal{A}$, $S \subseteq \mathcal{A} - \{i\}$. `stop` denotes a process that cannot show any action. The other operators are *action prefix* ($a;P$), *choice* ($P_1[]P_2$), *parallel composition* ($P_1|[S]|P_2$), *process instantiation* (X). $\mathcal{P}$ denotes the set of all possible processes. By means of the function $\mathcal{L}$ below, a process can be associated with the set of the actions it can perform.

Definition 3.1. *Let P be a LOTOS process and $\mathcal{E}$ a set of process declarations, $\mathcal{L}_{\mathcal{E}}(P) \subseteq \mathcal{A}$ is the set of actions obtained as the least solution of the following recursive definition:*

$$\begin{array}{lcl}
\mathcal{L}_{\mathcal{E}}(\texttt{stop}) & = & \emptyset \\
\mathcal{L}_{\mathcal{E}}(\alpha.P) & = & \begin{cases} \mathcal{L}_{\mathcal{E}}(P) \cup \{\alpha\} & \textit{if } \alpha \neq i \\ \mathcal{L}_{\mathcal{E}}(P) & \textit{if } \alpha = i \end{cases} \\
\mathcal{L}_{\mathcal{E}}(P[]Q) & = & \mathcal{L}_{\mathcal{E}}(P|[S]|Q) \ = \ \mathcal{L}_{\mathcal{E}}(P) \cup \mathcal{L}_{\mathcal{E}}(Q) \\
\mathcal{L}_{\mathcal{E}}(X) & = & \mathcal{L}_{\mathcal{E}}(P) \ \ \textit{if } X := P \in \mathcal{E}
\end{array}$$

When clear from the context, $\mathcal{L}(P)$ is used instead of $\mathcal{L}_{\mathcal{E}}(P)$; moreover, the process $P \ |[\mathcal{L}(P) \cap \mathcal{L}(Q)]| \ Q$ is denoted by $P \ |/ \ Q$, and the process $P \ |[\emptyset]| \ Q$ by $P|||Q$.

Given a set $\mathcal{E}$ of process declarations, the standard *operational semantics* of a process is given through a relation $\longrightarrow_{\mathcal{E}} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ ($\longrightarrow$ for short), that is the least relation defined by the rules in Table 1, where, for the sake of simplicity, the symmetric rules for choice and parallel composition are not shown; this fact does not alter the completeness of the method, which exploits the definition of $\longrightarrow$. For each $P, Q \in \mathcal{P}$, it holds that $P \xrightarrow{\lambda} P$, while, if $\gamma = \alpha_1 \dots \alpha_n, n \geq 1$, $P \xrightarrow{\gamma} Q$ means $P \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} Q$; γ is also called a *finite computation* of P. If $P \xrightarrow{\gamma} Q$, the process Q is called a *derivative* of P; when P has a finite number of syntactically different derivatives, P is called *finite state*, or simply *finite*. We consider only finite LOTOS processes.

The operational semantics of a process 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 $\mathcal{A}$.

3.2 The language to define properties

Temporal logic presents constructs allowing to state in a formal way that, for instance, all scenarios will

$$\textbf{pre}\quad \frac{}{\alpha; P \xrightarrow{\alpha} P}$$

$$\textbf{choice}\quad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \,[]\, P_2 \xrightarrow{\alpha} P_1'}$$

$$\textbf{inst}\quad \frac{P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'}\; X := P \in \mathcal{E}$$

$$\textbf{par}\quad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \,|[S]|\, P_2 \xrightarrow{\alpha} P_1' \,|[S]|\, P_2}\; \alpha \notin S$$

$$\textbf{com}\quad \frac{P_1 \xrightarrow{\alpha} P_1',\; P_2 \xrightarrow{\alpha} P_2'}{P_1 \,|[S]|\, P_2 \xrightarrow{\alpha} P_1' \,|[S]|\, P_2'}\; \alpha \in S$$

Table 1: *Standard operational semantics*

respect some property at every step, or that some particular event will eventually happen, and so on. Here we use the logic called *selective mu-calculus* (Barbuti et al., 1999b) that is a variant of the mu-calculus (Stirling, 1989) with a slight different definition of the modal operators. It was defined with the goal of reducing the number of states of the transition system corresponding to a pocess P in such a way that the reduction is driven by the formulae to be checked, and in particular by the syntactic structure of the formulae.

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:

$$\begin{aligned}\phi \quad ::= \quad & \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \vee \phi \mid \phi \wedge \phi \mid \\ & [K]_R\,\phi \mid \langle K\rangle_R\,\phi \mid \nu Z.\phi \mid \mu Z.\phi\end{aligned}$$

As in standard mu-calculus, a fixed point formula has the form $\mu Z.\varphi$ ($\nu Z.\varphi$) where μZ (νZ) *binds* free occurrences of Z in φ. An occurrence of Z is free if it is not within the scope of a binder μZ (νZ). 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. The precise definition of the satisfaction of a closed formula φ by a finite LOTOS process P, written $P \models \varphi$, is in Table 2 and relies on the transition relation $\Longrightarrow_I$, parametric with respect to $I \subseteq \mathcal{A}$, with the following meaning.

Definition 3.2. *Given $I \subseteq \mathcal{A}$, the relation $\Longrightarrow_I \subseteq \mathcal{P} \times I \times \mathcal{P}$ is such that, for each $\alpha \in I$ and $P, Q \in \mathcal{P}$*

$$P \overset{\alpha}{\Longrightarrow}_I Q \quad \textit{iff} \quad P \xrightarrow{\gamma\alpha} Q, \textit{ where } \gamma \in (\mathcal{A} - I)^*$$

By $P \overset{\alpha}{\Longrightarrow}_I Q$ we express the fact that it is possible to pass from P to Q by performing a (possibly empty) sequence of actions not belonging to I and then the action α in I. Note that $\Longrightarrow_{\mathcal{A}} = \longrightarrow$.

$P \not\models \mathtt{ff}$		
$P \models \mathtt{tt}$		
$P \models \varphi \wedge \psi$	iff	$P \models \varphi$ and $P \models \psi$
$P \models \varphi \vee \psi$	iff	$P \models \varphi$ or $P \models \psi$
$P \models [K]_R\,\varphi$	iff	$\forall P'.\forall\alpha \in K.P \overset{\alpha}{\Longrightarrow}_{K\cup R} P'$ implies $P' \models \varphi$
$P \models \langle K\rangle_R\,\varphi$	iff	$\exists P'.\exists\alpha \in K.P \overset{\alpha}{\Longrightarrow}_{K\cup R} P'$ and $P' \models \varphi$
$P \models \nu Z.\varphi$	iff	$P \models \nu Z^n.\varphi$ for all n
$P \models \mu Z.\varphi$	iff	$P \models \mu Z^n.\varphi$ for some n

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

$$\begin{aligned}&\nu Z^0.\varphi = \mathtt{tt} && \mu Z^0.\varphi = \mathtt{ff}\\ &\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z] && \mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]\end{aligned}$$

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

Table 2: Satisfaction of a closed formula by a process

Informally, $[K]_R\,\varphi$ is satisfied by a process which, after each finite computation $\gamma\alpha$, where $\gamma \in \mathcal{A} - (R \cup K)^*$ and $\alpha \in K$, evolves into a process satisfying φ.
$\langle K\rangle_R\,\varphi$ is satisfied by a process P having at least one finite computation $\gamma\alpha$, where $\gamma \in \mathcal{A} - (R \cup K)^*$ and $\alpha \in K$, after which P evolves into a process satisfying φ.

As shown in (Barbuti et al., 1999b), selective mu-calculus is equi-expressive to mu-calculus, in particular, it is possible to give a correspondence between each modal selective operator and a mu-calculus formula as follows:

$$[K]_R\,\varphi = \nu Z.[K]\varphi \wedge [\mathcal{A} - (K \cup R)]Z$$

$$\langle K\rangle_R\,\varphi = \mu Z.\langle K\rangle\varphi \vee \langle \mathcal{A} - (K \cup R)\rangle Z$$

In the following, to give an easy description of the presented method, we will consider only selective formulae without explicit recursive operators: the limitation is made less meaningful by the presence of the implicit recursion inside the selective operators. Without loss of generality, we shall assume that a selective mu-calculus formula contains only modal operators of the form $\langle\alpha\rangle_R$ and $[\alpha]_R$, with $\alpha \in \mathcal{A}$ and $R \subseteq \mathcal{A}$: in fact, $\langle\{\alpha_1,\ldots,\alpha_n\}\rangle_R\,\varphi = \langle\alpha_1\rangle_R\,\varphi \vee \cdots \vee \langle\alpha_n\rangle_R\,\varphi$ and $[\{\alpha_1,\ldots,\alpha_n\}]_R\,\varphi = [\alpha_1]_R\,\varphi \wedge \cdots \wedge [\alpha_n]_R\,\varphi$. The following definition is useful to express the notion of non-interference between formulae.

Definition 3.3. *Let φ be a selective mu-calculus for-*

Table 3: Families Description and Corresponding Logic Rules.

DroidKungFu	**Rule (selective mu-calculus formulae)**
	$\varphi = \varphi_1 \vee \varphi_2 \vee \varphi_3$ where:
device ID	$\varphi_1 = \langle pushphone\rangle_\emptyset \langle invokegetSystemService\rangle_\emptyset \langle checkcastandroidtelephonyTelephonyManager\rangle_\emptyset \langle invokegetDeviceId\rangle_\emptyset \mathtt{tt}$
IMEI	$\varphi_2 = \langle pushIMEI\rangle_\emptyset \langle load\rangle_\emptyset \langle invokeinit\rangle_\emptyset \langle invokeadd\rangle_\emptyset \mathtt{tt}$
device rooting	$\varphi_3 = \langle pushchmod\rangle_\emptyset \langle invokeinit\rangle_\emptyset \langle store\rangle_\emptyset \langle load\rangle_\emptyset \mathtt{tt}$
Opfake	**Rule (selective mu-calculus formulae)**
	$\psi = \psi_1 \vee \psi_2 \vee \psi_3$ where:
SMS sending	$\psi_1 = \langle load\rangle_\emptyset \langle invokesendTextMessage\rangle_\emptyset \mathtt{tt}$
download file	$\psi_2 = \langle push\rangle_\emptyset \langle anewarray\rangle_\emptyset \langle invokegetMethod\rangle_\emptyset \mathtt{tt}$
SMS sending by reflection	$\psi_3 = \langle pushsendTextMessage\rangle_\emptyset \langle load\rangle_\emptyset \langle invokegetMethod\rangle_\emptyset \mathtt{tt}$

mula.

$$\begin{array}{lcl} box(\varphi) & = & \{\alpha \mid [\alpha]_R \textit{ occurs in } \varphi\} \\ diamond(\varphi) & = & \{\alpha \mid \langle\alpha\rangle_R \textit{ occurs in } \varphi\} \\ act(\varphi) & = & box(\varphi) \cup diamond(\varphi) \end{array}$$

The property of non-interference is of relevance when managing conjunctions of sub-formulae, such as $\varphi_1 \wedge \varphi_2$.

Definition 3.4. *Let φ be a selective mu-calculus formula, non-interference holds if $\mathcal{N}(\varphi) = true$, where $\mathcal{N}$ is inductively defined as follows.*

$$\begin{array}{lclcl} \mathcal{N}(\mathtt{tt}) & = & \mathcal{N}(\mathtt{ff}) & = & true \\ \mathcal{N}(\langle\alpha\rangle_R\varphi) & = & \mathcal{N}([\alpha]_R\varphi) & = & \mathcal{N}(\varphi) \\ \mathcal{N}(\varphi_1 \vee \varphi_2) & = & \mathcal{N}(\varphi_1) \textit{ and } \mathcal{N}(\varphi_2) & & \end{array}$$

$$\mathcal{N}(\varphi_1 \wedge \varphi_2) = \begin{cases} \mathcal{N}(\varphi_1) \textit{ and } \mathcal{N}(\varphi_2) \\ \quad \textit{if } box(\varphi_1) \cap act(\varphi_2) = \emptyset \textit{ and} \\ \quad box(\varphi_2) \cap act(\varphi_1) = \emptyset \\ false \\ \quad \textit{otherwise} \end{cases}$$

Intuitively, a formula is interfering if it contains a subformula $\varphi_1 \wedge \varphi_2$ such that $\varphi_i, i = 1,2$ contains an operator $[\alpha]_R$ and $\varphi_j, j \neq i$ contains either $[\alpha]_{R'}$ or $\langle\alpha\rangle_{R'}$.

It is possible to apply the *not* operator, as usual for dual operators, to selective formulae without recursive operators with the following meaning:
$not(\langle\alpha\rangle_R\varphi) = [\alpha]_R(not\ \varphi)$
$not(\varphi_1 \wedge \varphi_2) = (not\ \varphi_1) \vee (not\ \varphi_2)$

One of the most popular environments for verifying concurrent systems through model checking of temporal logic formulae is CADP (Garavel et al., 2013), which supports several different specification languages, among which LOTOS, and different temporal logics, among which mu-calculus.

4 THE METHODOLOGY

In this section we present the steps of our methodology for the localization of Android malware: it is based on the concepts of the model checking and defines some constructive rules able to obtain a description of the wrong behavior of the system in terms of a LOTOS process. While efficient model checking techniques (Barbuti et al., 2005; Barbuti et al., 1999a) have been proposed to verify the correct behavior of a system, recently they have been also applied in other disciplines such as clone detection (Santone, 2011), biology (Ruvo et al., 2015), secure information flow (Barbuti et al., 2002), among others. In this paper, model checking technique has been applied in the security field.

Step 1: Java Bytecode-to-LOTOS

The first step generates a LOTOS specification from the Java Bytecode of the .class files derived by the analysed apps. This is obtained by defining a Java Bytecode-to-LOTOS transform operator $\mathcal{T}$. The function $\mathcal{T}$ directly applies to the Java Bytecode ($\mathcal{T}$ is defined for each instruction of the Java Bytecode) and translates it into LOTOS process specifications. A Java Bytecode program P is seen as a sequence s of instructions, numbered starting from address 0; $\forall i \in \{0,\ldots,\sharp s\}$ $s[i]$ is the instruction at address i, where $\sharp s$ denotes the length of s. The translation is very similar to that sketched in (Battista et al., 2016) for *CCS*: in essence, each bytecode instruction is translated into a process performing only the action representing that instruction (i.e. assignment, goto and so on) and then transforming into the process corresponding to the next executable instruction.

Step 2: Android malware as temporal logic formulae

The second step aims at expressing android malware by temporal logic formulae. The processes obtained in the first step can be model checked against the formulae describing the malware. LOTOS processes are mapped into labelled transition systems by the CADP, and, if the result of the verification is false, we have the need of a precise localization of the malware. We shall see in the next section that our methodology can actually substitute the model checking since, when the malware is not present in the code, we obtain an empty process X.

Table 3, taken from (Battista et al., 2016), elicits the malicious behaviors for some malware families and shows the resulting translation into logic rules. The meaning of the table is the following:
For the rules characterizing DroidKungFU:
φ_1 = device ID, it represents, the malware ability to retrieve the specific alpha-numeric Identification code associated with the mobile device, employed by Google to identify the user when she/he logs on the Google Play in order to download and/or buy applications;
φ_2 = IMEI, i.e., the ability of the malicious payload to stealthy gather the International Mobile Equipment Identity (IMEI), a numeric code that uniquely identifies a mobile terminal;
φ_3 = device rooting, the use of tricks in order to try to obtain root privileges, in this case the payload tries to change permissions to files and/or folders using the chmod (change mode) command derived by Unix operating systems.
While for the rules characterizing Opfake:
ψ_1 = SMS sending, this behavior represents the ability to send SMS messages to premium-rate numbers. In this case the payload is also able to parse the incoming SMS for a sender's number containing 088011 or 000100 (considering that malware variants usually encrypt hard-coded phone numbers, in the rule we do not consider the specified phone numbers but the malicious behaviour able to parse the incoming SMS), if this condition is satisfied the SMS body is parsed through a regular expression. If the sequence is identified, the group matching it is set as the secret code and the sms is hidden from the user's eyes;
ψ_2 = download file, this behavior is one of the most representative of Opfake family: it adds an Opera browser icon on the menu and displays a fake download progress bar to make it appear that the application (with a fake license) is actually downloading, but in the meanwhile the malware downloads the malicious payload;
ψ_3 = SMS sending by reflection, i.e. the ability to invoke at run-time a method (contained in a jar o dex file) able to send SMS messages using the reflection mechanism provided by Java environment. This mechanism is usually involved to update application without shutdown and restart the app, but in this case it is used to evade the static detection by antimalware.

Here we consider only formulae, as those shown in Table 3, containing the selective operator $\langle\alpha\rangle_R$ and the logical operator $\vee$; anyway, it is worth noting that any formula describing malware contains as terminal a sub-formula of the kind $\langle\alpha\rangle_R\,\mathtt{tt}$, since any malware requires that an action is executed. Then any formula expressing the absence of malware will terminates with the sub-formula like as $[\alpha]_R\,\mathtt{ff}$.

Step 3: Localization and description of malware

The third step defines a set of rules to build the integration needed by a finite state LOTOS process P, so that the resulting process $(P\;|[B]|\;X)$, for some set of actions B, satisfies a selective mu-calculus formula. For the sake of simplicity, the rules in Table 4 build X taking into account only formulae having the structure of those in Table 3. When considering the malware families described in Table 3, the rules will apply to the formulae *not* φ and *not* ψ expressing the absence of malware. The set B to start the construction is $box(not\ \varphi)$ (or $box(not \psi)$).

The method applies successive transformations to the original process and to the formula, building a set of integrations with the related environments and composing them into the final process X with its environment. Each rule is of the form:

$$\frac{P\;|[B]|\;X \vdash_{\mathcal{E}} \varphi}{P_1\;|[B]|\;X_1 \vdash_{\mathcal{E}_1} \varphi_1 \quad \cdots \quad P_n\;|[B]|\;X_n \vdash_{\mathcal{E}_n} \varphi_n}$$

where $n > 0$ and side conditions may exist. Moreover, it is $B \subseteq \mathcal{A}$, P a finite LOTOS proces, $P_1,\ldots,P_n$ derivates of P, $\mathcal{E}$ a set of declarations $\{X_1 := P_1,\ldots,X_m := P_m\}$, such that $X_i \neq X_j$, if $i \neq j$, X does not occur in $\mathcal{E}$ and, moreover, $\mathcal{E}$ includes a process environment for P. The premise is the goal to be achieved, the consequents are the sub-goals determined by the structure of the formula, by the environment and the possible derivatives of P.

At a point we will obtain a set of terms as that below, for $i \in [i_1..i_m]$, for which no rule can be more applied

$$P_i\;|[B]|\;X_i \vdash_{\mathcal{E}_i} \varphi_i$$

and thus a final environment must be given for each X_i. The possible situations in which no rule can be applied are listed in Table 5.

box	$\dfrac{P\;\vert[B]\vert\;X \vdash_{\mathcal{E}} [\alpha]_R\,\varphi}{P_1\;\vert[B]\vert\;Y_1 \vdash_{\mathcal{E}'} \varphi \cdots P_n\;\vert[B]\vert\;Y_n \vdash_{\mathcal{E}'} \varphi}$	$\varphi \neq \mathtt{ff}$ $\{P_1,\ldots,P_n\} = \{P' \mid P \overset{\alpha}{\Longrightarrow}_{\{\alpha\}\cup R} P'\}$ $\mathcal{E}' = \mathcal{E} \cup \{X := \alpha; (\cdots(Y_1 \;\vert/\; Y_2)\cdots) \;\vert/\; Y_n)\}$
and	$\dfrac{P\;\vert[B]\vert\;X \vdash_{\mathcal{E},\Delta} \varphi_1 \wedge \varphi_2}{P\;\vert[B]\vert\;Y_1 \vdash_{\mathcal{E}',\Delta} \varphi_1 \qquad P\;\vert[B]\vert\;Y_2 \vdash_{\mathcal{E}',\Delta} \varphi_2}$	$\mathcal{E}' = \mathcal{E} \cup \{X := Y_1 \;\vert\vert\vert\; Y_2\}$

Table 4: Rules

1. $\varphi_i = \mathtt{tt}$.
2. $\varphi_i = [\alpha]_R\,\varphi$ and φ is logically equivalent to $\mathtt{ff}$.
3. $\varphi_i = [\alpha]_R\,\varphi$ and $\{P' \mid P_i \overset{\alpha}{\Longrightarrow}_{\{\alpha\}\cup R} P'\} = \emptyset$.

Table 5: Possible terminations

The final process produced through the procedure is the following

process $P_F := (P\;|[B]|\;X)$
  where $\mathcal{E}_0 \cup \mathcal{E}_1 \cup \cdots \cup \mathcal{E}_n \cup \mathcal{E}_{stop}$
endproc

Where $\mathcal{E}_0$ is the environment of P, $\mathcal{E}_1 \cdots \mathcal{E}_n$ are the environments built by successive rule applications, and, for $i \in [i_1..i_m]$,

$$\begin{aligned}\mathcal{E}_{stop} = \;& \{X_i := \mathtt{stop} \mid \\ & \text{for each } P_i\;|[B]|\;X_i \vdash_{\mathcal{E}_1} \varphi_i \\ & \text{matching one situation of Table 5 }\}\end{aligned}$$

5 RESULTS AND DISCUSSION

Now we show a simple system P possibly affected by malware represented by the formula ψ of Table 3. The formula $\psi' = not\;\psi$ is as follows (we consider a shorthand of the names of the events in ψ taking the initial letter or the initial letter followed by the capital letters in the name of the action),

$$\psi' = [l]_\emptyset\,[iTM]_\emptyset\,\mathtt{ff} \wedge [p]_\emptyset\,[a]_\emptyset\,[iM]_\emptyset\,\mathtt{ff}$$

$$\wedge\,[pTM]_\emptyset\,[l]_\emptyset\,[iM]_\emptyset\,\mathtt{ff}$$

while the process P is as below:

process $P := (P_1|[f]|P_2)$
  where $P_1 := d;l;b;stop[]g;l;h;f;iTM;P_1$
    $P_2 := f;p;c;a;iM;P_2$
endproc

Starting with $\mathcal{E}_0 = \{P_1, P_2\}$ and $B = \{l, iTM, p, a, iM\}$, we apply the rules of Table 4 obtaining

process $P_F := (P\;|[B]|\;X)$
  where $P_1 := d;l;b;stop[]g;l;h;f;iTM;P_1$
    $P_2 := f;p;c;a;iM;P_2$
    $X := (Y_1|||Y_2|||Y_3)$
    $Y_1 := l;(Y_4|/Y_7)$
    $Y_2 := p;Y_5$
    $Y_3 := stop$
    $Y_5 := a;Y_6$
    $Y_4 := stop$
    $Y_6 := stop$
    $Y_7 := stop$
endproc

We can note that the complexity of X depends on the formula, and on the complexity of the process only for what regards the number of possible derivatives (operator $|/$) following a same action (but not on the complexity of the derivatives). X contains a set of synchronizations with all the actions contained in the malware, apart the actions iTM and iM that do not occur in X; so, in some sense, the malicious behavior is inactivated in P_F and cannot completely obtain its purpose since P is blocked on a synchronization not performed by X. On the other hand, also the action pTM does not occur in X, since the behavior specified by the formula ψ_3 of Table 3 cannot start when pTM is not present in P; the actions l and iM are present in P, but their occurrence alone does not indicate the presence of malware.

If the procedure is applied separately to P_1 and P_2, we will obtain
process $P_F := (P_{F1}|[f]|P_{F2})$
... endproc

and X_1 and X_2 will describe different infections
process $P_{F1} := (P_1\;|[B]|\;X_1)$

  where $X_1 := (Y_1'|||Y_2'|||Y_3')$
    $Y_1' = Y_1$
    $Y_2' := stop$
    $Y_3' := stop$
endproc
process $P_{F2} := (P_2\;|[B]|\;X_2)$

  where $X_2 := (Y_1''|||Y_2''|||Y_3'')$
    $Y_1' = stop$
    $Y_2' := Y_2$

```
    $Y_3' := stop$
endproc
```

It is possible to note that, at any level the methodology is applied, the actions *iTM* and *iM* are not allowed and this knowledge can be an useful suggestion for modifying the Java bytecode to make the malicious behavior ineffective. In fact, we can transfer the behavior of X in a bytecode component to be associated to the bytecode described by P to avoid the effect of the malware.

Finally, as proved in (Santone and Vaglini, 2003), we recall that all properties of P, not interfering with ψ', are maintained in the resulting processes.

6 CONCLUDING REMARKS AND FUTURE WORK

Considering that current antimalware and prototypes proposed by researchers try to discriminate a mobile malware application from a legitimate one, we propose a methodology to localize the malicious payload with the aim to make it ineffective, to permit to the user to install and to run the disinfected application. We use model checking concepts in order to apply our methodology against two of most diffused malware family in Android environment: the *DroidKungFu* and the *Opfake* families.

As future work we are going to extend this preliminary methodology to include all selective operators and other formulae specifying malware; moreover, we intend to implement our rules and obtain a deep evaluation of the capability of using the result to inactivate the malware beside to precisely localize it.

REFERENCES

Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*. IEEE.

Bailey, U., Comparetti, P., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*. IEEE.

Barbuti, R., De Francesco, N., Santone, A., and Tesei, L. (2002). A notion of non-interference for timed automata. *Fundamenta Informaticae*, 51(1-2):1–11. cited By 6.

Barbuti, R., De Francesco, N., Santone, A., and Vaglini, G. (1999a). Loreto: A tool for reducing state explosion in verification of lotos programs. *Software - Practice and Experience*, 29(12):1123–1147. cited By 12.

Barbuti, R., Francesco, N. D., Santone, A., and Vaglini, G. (1999b). Selective mu-calculus and formula-based equivalence of transition systems. *J. Comput. Syst. Sci.*, 59(3):537–556.

Barbuti, R., Francesco, N. D., Santone, A., and Vaglini, G. (2005). Reduced models for efficient CCS verification. *Formal Methods in System Design*, 26(3):319–350.

Battista, P., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Identification of android malware families with model checking. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016), Rome, Italy, February 19-21, 2016.*, pages 542–547. SciTePress.

Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59.

Canfora, G., Di Sorbo, A., Mercaldo, F., and Visaggio, C. (2015). Obfuscation techniques against signature-based detection: a case study. In *Proceedings of Workshop on Mobile System Technologies*. IEEE.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2013). A classifier of malicious android applications. In *Proceedings of the 2nd International Workshop on Security of Mobile Applications, in conjunction with the International Conference on Availability, Reliability and Security*. IEEE.

Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2013). CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107.

Hu, X., Chiueh, T., Shin, K., Kruegel, C., and Kirda, E. (2009). Large-scale malware indexing using function call graphs. In *ACM Conference on Computer and Communications Security*. ACM.

Jacob, G., Filiol, E., and Debar, H. (2010). Formalization of viruses and malware through process algebras. In *International Conference on Availability, Reliability and Security (ARES 2010)*. IEEE.

Jang, J., Brumley, D., and Venkataraman, S. (2011). Bitshred: feature hashing malware for scalable triage and semantic analysis. In *ACM Conference on Computer and Communications Security*. ACM.

Kinder, J., Katzenbeisser, S., Schallhart, C., and Veith, H. (2005). Detecting malicious code by model checking. Springer.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016a). Download malware? no, thanks: how formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2016, Austin, Texas, USA, May 15, 2016*, pages 22–28. ACM.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016b). Hey malware, I can find you! In *25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2016, Paris, France, June 13-15, 2016*, pages 261–262. IEEE Computer Society.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016c). Ransomware steals your phone. formal methods rescue it. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 212–221. Springer.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190. ACM Press.

Ruvo, G., Nardone, V., Santone, A., Ceccarelli, M., and Cerulo, L. (2015). Infer gene regulatory networks from time series data with probabilistic model checking. pages 26–32. cited By 7.

Santone, A. (2011). Clone detection through process algebras and java bytecode. pages 73–74. cited By 10.

Santone, A. and Vaglini, G. (2003). Modifying LOTOS specifications by means of automatable formula-based integrations. *J. Autom. Reasoning*, 30(1):33–58.

Schmidt, A., Bye, R., Schmidt, H., Clausen, J. H., Kiraz, O., Yüksel, K. A., Çamtepe, S. A., and Albayrak, S. (2009). Static analysis of executables for collaborative malware detection on android. In *Proceedings of IEEE International Conference on Communications, ICC 2009, Dresden, Germany, 14-18 June 2009*, pages 1–5. IEEE.

Song, F. and Touili, T. (2001). Efficient malware detection using model-checking. Springer.

Song, F. and Touili, T. (2013). Pommade: Pushdown model-checking for malware detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM.

Song, F. and Touili, T. (2014). Model-checking for android malware detection. Springer.

Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In Yonezawa, A. and Ito, T., editors, *Concurrency: Theory, Language, And Architecture*, LNCS, pages 2–20. Springer.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*. IEEE.