International Journal of Information Security (2018) 17:719–738
https://doi.org/10.1007/s10207-017-0398-5

REGULAR CONTRIBUTION

Talos: no more ransomware victims with formal methods

Aniello Cimitile[1] · Francesco Mercaldo[2] · Vittoria Nardone[1] · Antonella Santone[3] · Corrado Aaron Visaggio[1]

Published online: 19 December 2017

Abstract
Ransomware is a very effective form of malware that is recently spreading out on an impressive number of workstations and smartphones. This malware blocks the access to the infected machine or to the files located in the infected machine. The attackers will restore the machine and files only after the payment of a certain amount of money, usually given in the form of bitcoins. Commercial solutions are still ineffective to recognize the last variants of ransomware, and the problem has been poorly investigated in literature. In this paper we discuss a methodology based on formal methods for detecting ransomware malware on Android devices. We have implemented our method in a tool named Talos. We evaluate the method, and the obtained results show that Talos is very effective in recognizing ransomware (accuracy of 0.99) even when it is obfuscated (accuracy still remains at 0.99).

Keywords Malware · Mobile · Ransomware · Security · Model checking · Android

1 Introduction

Ransomware is a malware that impedes the access to a device or to files and demands a payment for restoring the functionality of the infected smartphone.

Mobile ransomware improves its capabilities at a very fast pace, adding more aggressive features to new variants emerging in the wild [2]. It usually comes to the victim's machine through sophisticated forms of phishing, and leverages some hiding techniques [45] that are able to recognize the environments. Ransomware can be also enriched with anti-debugging techniques, which make antimalware scanning harder [51]. Ransomware is also using techniques known as process hollowing [70] for hiding the actual ransomware process by nestling in host legitimate processes to lock the device and to gather sensitive information [18,25].

As the evidence of the high infections rate demonstrates, commercial antimalware solutions are mainly ineffective to detect ransomware [11,13,59,77]. As a matter of fact, in 2016 Kaspersky labs found that the number of mobile malicious installation packages grew considerably, amounting to 8,526,221: three times more than the previous year. As a comparison, from 2004 to 2013 the security analysts detected over 10,000,000 malicious installation packages [39].

In this paper we propose a technique for detecting ransomware on smartphone devices that is based on model checking [23]. Model checking is a useful method to verify automatically 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 this behaviour, expressed using a temporal logic. An Android application is modelled as an automaton, and the ransomware malicious behaviour is expressed with a temporal logic formula. If the formula is true on the automaton, then we consider the application belonging to the ransomware family. The technique has shown to be very effective. In our evaluation we obtained an accuracy equal to 0.99. The technique has two additional advantages, due to the use of the

✉ Antonella Santone
antonella.santone@unimol.it

Aniello Cimitile
cimitile@unisannio.it

Francesco Mercaldo
francesco.mercaldo@iit.cnr.it

Vittoria Nardone
vnardone@unisannio.it

Corrado Aaron Visaggio
visaggio@unisannio.it

[1] Department of Engineering, University of Sannio, Benevento, Italy

[2] Institute for Informatics and Telematics, CNR, Pisa, Italy

[3] Department of Bioscience and Territory, University of Molise, Pesche, IS, Italy

model checking. First, it is able to localize in the code of the ransomware those instructions that implement specific malicious behaviours, which can particularly help in the phase of malware dissection and reconnaissance. Secondly, it is robust with respect to the obfuscation. Obfuscation techniques [58,59] are used for hiding the specific implementation of a code from automatic or human analysis [57]. In goodware, obfuscation helps to protect intellectual properties of the code or data and information that the author of the code does not want to be known. Obfuscation is largely employed by malware writers [11,13] for hindering human or automatic detection or reverse engineering. Being a behaviour-based approach, we are able to detect ransomware malware codes also in presence of minor obfuscations and modifications. This is a great result since nowadays Android malware detectors are able to identify a malware using its signature, which is based on code structure. Thus, it is very easy to evade the mechanism of signature recognition by code obfuscation. For this reason it is very important to develop malware detection techniques which are able to recognize malware even if it is obfuscated. A few papers propose specific detectors for ransomware, since at the best knowledge of the authors only four papers face this problem [2,17,69,76], even if in [76] only a theoretical solution has been illustrated, without any implementation.

A very preliminary work has been presented as short paper at FORTE 2016 [49], which, as far as we know, is the first attempt to exploit model checking technique for detecting ransomware malware.

This paper extends that initial work introducing the following distinctive features:

- we present the design and the implementation of Talos, an efficient and practical tool based on our formal approach;
- we analyse a more complete and relevant dataset;
- we localize the malicious payload in ransomware samples, i.e., we are able to identify the class and the method that performs the malicious action;
- we show and demonstrate how our method is resilient with respect to the obfuscation;
- we provide a characterization and a dissection of the ransomware, which is a further application of the proposed detection technique;
- we have validated our results by comparing them to those obtained by a state-of-the-art approaches for ransomware detection [2,17,69];
- we present some logic formulae describing a ransomware malicious behaviour.

The paper proceeds as follows: Sect. 2 discusses related work; Sect. 3 explains the background related to the formal methods; Sect. 4 describes and motivates our method; Sect. 5 illustrates the results of experiments; finally, conclusions are drawn in Sect. 6.

2 Related work

In this section we review the current literature related to malware detection in mobile devices. We first discuss recent works mainly based on the detection on Android to further analyse in depth the literature addressing the issues of ransomware detection and malware identification using formal methods.

2.1 Malware detection

Concerning Android malware detection, several different approaches have been proposed in the recent past, due to the rising trend of malware spreading: two recent surveys in [27,74] summarize the main techniques for securing the Android platform. Malware detectors on Android are basically based on features extraction using static and/or dynamic analysis.

Authors in [44] propose a permission-based approach: they extract requested and used permissions and make combinations of them to build a classifier to test 28,548 benign applications and 1563 malicious ones, obtaining a precision equal to 89.8%.

Song et al. [68] propose a framework to statically detect Android malware, consisting of four layers of filtering mechanisms: the message digest values (i.e., the cryptographic hash functions containing a string of digits created by a one-way hashing formula), the combination of malicious permissions, the dangerous permissions, and the dangerous intention. They validate their method, using 83 real mobile devices and achieving a 98.8% pass rate, where the versions of Android range from 2.3 to 5.1.

Canfora et al. [12] propose a method for detecting Android malware which is based on the analysis of system call sequences and machine learning behavioural analysis [62]. The experimentation on 20,000 execution traces of 2000 applications (1000 of them being malware belonging to different malware families), performed on a real device, obtained a detection accuracy equal to 97%.

Researchers in [14] propose an Android malware detector evaluating permissions and system calls related to process management and to *I*/*O* operations. They obtain a precision equal to 74%, a recall equal to 80%, and a ROC area equal to 72%, using a dataset of 200 malicious applications.

The authors in [10,15,16,50] evaluate the effectiveness of the opcodes frequencies to discriminate Android malicious samples from legitimate ones. The best accuracy obtained in the malware identification is equal to 96.88% . The detection rate among the analysed malware families (i.e., FakeInstaller, DroidKungFu, Plankton, Opfake, GinMaster, BaseBridge, Kmin, Geinimi, Adrd and DroidDream) ranges from 88.6 to 100%.

Authors in [64] proposed a behaviour-based and multi-level Android malware detection systems: their experiments reported a detection accuracy of 96.9% on a dataset composed by 2784 malicious apps.

Crowdroid [9] is a framework for the collection of traces from an unlimited number of real users based on crowdsourcing, monitoring system calls and sending them preprocessed to a centralized server.

2.2 Ransomware detection

In this section we report related work mainly focused on ransomware detection in mobile environment.

HelDroid [2] tool includes a text classifier based on NLP features, a lightweight Smali emulation technique to detect locking strategies, and the application of taint tracking for detecting file-encrypting flows. The main weakness of HelDroid is represented by the text classifier: the authors train it on generic threatening phrases, similar to those that typically appear in ransomware or scareware samples. More precisely, they train the classifier by using phrases labelled as threat, law, copyright, porn, and money, which typically appear in scareware or ransomware campaigns, identifying rightly 375 ransomware on a dataset formed by 443 samples. This method can be evaded by altering the occurrences of such words. In addition, like whatever machine learning approach, HelDroid needs to train the classifier in order to label a samples as a ransomware: the detection capability of the model is related to the training dataset.

In [17] the authors experimentally evaluate two techniques for detecting Android malware: the first one is based on Hidden Markov Model (HMM), while the second one exploits Structural Entropy. They demonstrate that these methods obtain a precision of 0.96 to discriminate a malware application. In addition they also analyse ransomware samples, obtaining a precision of 0.961 with LADTree classification algorithm using the structural entropy method, and a precision of 0.824 with J48 algorithm using HMM in ransomware identification.

Song et al. [69] propose a technique which consists in processes and file directories monitoring by using statistical methods based on Processor usage, Memory usage, and *I*/*O* rates so that the process with ransomware behaviours can be detected. They developed a ransomware sample able to encrypt files with 40-byte keys using the AES algorithm for validating their method. The self-made ransomware has the function of opening all the files on the input path and encrypting them.

Another work in literature exploring the ransomware detection in mobile world is [76]. The authors illustrate a possible design of a static and dynamic analysis based solution, without implementing it. Their goal is to build a tool that helps to understand how to support the approach for a successful detection of Android ransomware.

We compare our method with HelDroid [2] and the work presented in [17]. We are not able to compare our method with other cited works (i.e., the papers [69,76]) because the first one [69] proposes a dynamic technique evaluated using just one ransomware sample, developed by the authors (i.e., they do not test their solution using a real-world ransomware dataset); while the second one [76] illustrates a possible methodology to identify ransomware in Android environment, but authors do not implement the proposed method.

With regard to HelDroid [2] and the paper proposed in [17], both the works evaluate the proposed solutions on an extended real-world dataset of ransomware applications for Android environment.

HelDroid [2], among 443 total samples, is able to correctly detect 375 as ransomware or scareware, while authors in [17] state that the first method they propose (i.e., the one HMM-based) obtains a precision of 0.824 and a recall of 0.704 with J48 algorithm, while the second one, based on the structural entropy, obtains a precision of 0.961 and a recall of 0.879 with LADTree classification algorithm to recognize ransomware mobile applications.

A detailed comparison with the above techniques will be made in Sect. 5.6, using in some cases the same dataset.

As we explain in the evaluation section, Talos is able to reach a precision equal to 0.99, outperforming previous works (that we compare in terms of true positive and recall). Furthermore, we highlight that we obtain better performances testing the same application dataset.

2.3 Malware detection using formal methods

We now review related literature about malware detection which relies on formal methods.

In [40] the authors introduce the specification language CTPL (Computation Tree Predicate Logic) confirming the malicious behaviour of thirteen Windows malware variants using as dataset a set of worms dating from 2002 to 2004.

Song et al. [65] 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 [66] is able to detect 600 real malware for PCs, 200 malwares generated by two malware generators (NGVCK and VCL32), and proves the reliability of benign programs: a Microsoft Windows binary program is modelled as a PDS which allows to track the stack of a program.

Song et al. [67] model mobile applications using a PDS in order to discover private data leaking working at Smali code level.

Jacob et al. [37] provide a basis for a malware model, founded on the Join-Calculus: they consider the system call sequences to build the model.

At the best knowledge of the authors, the idea of exploiting model checking for recognizing the ransomware has never been used before. The authors of this paper have recently proposed an approach based on model checking that is able to identify the malicious behaviour at method grain [7,47,48]. Using the mu-calculus temporal logic, logic rules are formulated in order to detect the maliciousness of a real-world malware dataset.

Considered the widespread application of obfuscation for hiding malware's code and hindering the detection, many solutions have been proposed for detecting obfuscated malware.

Semantics templates were defined in [21] to detect malicious traits in obfuscated malware: these templates were generated based on instruction, variable and symbolic constants. In [20], authors introduce a system for detecting malicious patterns in executables that is resilient to common obfuscation. In [73] researchers developed a method relying on the assumption that all the versions of the same malware share a common core signature which is a combination of several features of the code. Some limits of static analysis for the detection of malicious code are explored in [53]. Authors in [42] propose a method to detect malware obfuscation which makes use of maximal patterns. A maximal pattern is a sub-sequence of system calls, which frequently occurs in program execution and can be used to describe the program specific behaviour. Metamorphism is a set of techniques for obfuscating malware, by morphing the malicious code at each infection. The detection of Metamorphic malware was investigated by different papers. A common approach [8,75] tries to detect metamorphic malware through some characterizing traces of API calls. Rieck et al. [61] use Support Vector Machine, while Hidden Markov Model was largely explored in [1,3,32].

Some works have specifically focused on obfuscated malware written in JavaScript. Caffeine Monkey [28] is a customized version of the Mozilla's SpiderMonkey [33], which allows the analysis of obfuscated malicious JavaScript. Wepawet [30] is an online service that can check for known exploits within a submitted JavaScript and offers also services like deobfuscation and capturing network activity. Jsunpack from iDefense [35] and "The Ultimate Deobfuscator" from WebSense [19] are two additional tools to automate the process of deobfuscating malicious JavaScript. Likarish [43] and colleagues propose a technique based on a classification engine for detecting malicious JavaScript when obfuscated, which demonstrated some limits with packed JavaScripts.

Our method improves the current body of knowledge in mobile malware detection with respect to three aspects: very high detection rate, the possibility to automatically localize specific malicious behaviour within the malicious code, and a great robustness with respect to the obfuscation.

3 Preliminaries

It could be useful to recall some preliminaries on formal methods.

3.1 Calculus of communicating systems

The calculus of communicating systems of Milner (CCS) [52] is one of the most well-known process algebras. Below we present only a brief overview of the main features of CCS. Readers unfamiliar with CCS are referred to [52] for further details. The syntax of *processes* is defined by the BNF equation:

$$p ::= nil \mid x \mid \alpha.p \mid p + p \mid p|p \mid p\backslash L \mid p[f]$$

where α ranges over a finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, \ldots\}$. The action $\tau \in \mathcal{A}$ is called the *internal action*. The set of *visible actions*, $\mathcal{V}$, ranged over by $l, l' \ldots$, is defined as $\mathcal{A} - \{\tau\}$. Each action $l \in \mathcal{V}$ (resp. $\overline{l} \in \mathcal{V}$) has a *complementary action* $\overline{l}$ (resp. l). The restriction set L, in the processes of the form $p\backslash L$, is a set of actions such that $L \subseteq \mathcal{V}$. The relabelling function f, in processes of the form $p[f]$, is a total function, $f : \mathcal{A} \to \mathcal{A}$, such that the constraint $f(\tau) = \tau$ is respected. The constant x ranges over a set of constant names: each constant x is defined by a constant definition $x \stackrel{def}{=} p$. Briefly, the interpretation of these expressions is as follows:

- nil represents a process which performs no actions;
- x represents the process bound to the variable in some assumed environments, i.e., if $x \stackrel{def}{=} p$, x behaves as its definition p;
- $\alpha.p$ represents the process which performs the action α and then it evolves into the process p;
- $p + q$ represents a choice between p and q;
- $p|q$ represents the (concurrent) composition of the process p and q, which allows the two processes to evolve independently or to synchronize on complementary actions;
- $p\backslash L$ represents the process which behaves as p except cannot perform any action l or $\overline{l}$ with $l \in L$;
- $p[f]$ represents the process p in which all actions are relabelled by f.

We denote the set of processes by $\mathcal{P}$. The standard *operational semantics* is given by a relation $\longrightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$. $\longrightarrow$ is the least relation defined by the rules in Table 1. Roughly

Table 1 Standard operational semantics of CCS

Act	$\dfrac{}{\alpha.p \xrightarrow{\alpha} p}$	**Sum**	$\dfrac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'}$ (and symmetric)
Rel	$\dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$	**Par**	$\dfrac{p \xrightarrow{\alpha} p'}{p\|q \xrightarrow{\alpha} p'\|q}$ (and symmetric)
Con	$\dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{def}{=} p$	**Com**	$\dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\bar{l}} q'}{p\|q \xrightarrow{\tau} p'\|q'}$
Res	$\dfrac{p \xrightarrow{\alpha} p'}{p\backslash L \xrightarrow{\alpha} p'\backslash L}\ \alpha, \bar{\alpha} \notin L$		

Table 2 Satisfaction of a closed formula by a state

$p \not\models \mathtt{ff}$

$p \models \mathtt{tt}$

$p \models \varphi \wedge \psi$ 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 $\forall p'. \forall \alpha \in K. p \xrightarrow{\alpha}_{K \cup R} p'$ implies $p' \models \varphi$

$p \models \langle K \rangle_R\, \varphi$ iff $\exists p'. \exists \alpha \in K. p \xrightarrow{\alpha}_{K \cup R} p'$ 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

where:

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

$\nu Z^0.\varphi = \mathtt{tt}$ $\quad$ $\mu Z^0.\varphi = \mathtt{ff}$

$\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z]$ $\quad$ $\mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]$

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

speaking, these conditional rules describe the transition relation of the automaton (transition system) corresponding to the process defining p.

A *(labelled) transition system* is a quadruple $\mathcal{T} = (S, \mathcal{A}, \longrightarrow, p)$, where S is a set of states, $\mathcal{A}$ is a set of transition labels (actions), $p \in S$ is the initial state, and $\longrightarrow \subseteq S \times \mathcal{A} \times S$ is the transition relation. If $(p, \alpha, q) \in \longrightarrow$, we write $p \xrightarrow{\alpha} q$. If $\delta \in \mathcal{A}^*$ and $\delta = \alpha_1 \ldots \alpha_n, n \geq 1$, we write $p \xrightarrow{\delta} q$ to mean $p \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} q$. Moreover $p \xrightarrow{\lambda} p$, where λ is the empty sequence. Given $p \in S$, with $\mathcal{R}(p) = \{q \mid p \xrightarrow{\delta} q\}$ we denote the set of the states reachable from p by $\longrightarrow$. Given a CCS process p, the *standard transition system* for p is defined as $\mathcal{S}(p) = (\mathcal{R}(p), \mathcal{A}, \longrightarrow, p)$.

3.2 Model checking and Mu-calculus logic

Given a transition system $\mathcal{T}$, we can ask questions such as the following:

- Are any "undesired" states reachable in $\mathcal{T}$, such as states that represent a deadlock?
- Are there runs of $\mathcal{T}$ such that, from some point on-wards, some "desired" state is never reached or some actions never executed?
- Is some initial system state of $\mathcal{T}$ reachable from every state?

Temporal logics are logical formalisms designed for expressing the above properties. More precisely, in the model checking framework, systems are modelled as transition systems and requirements are expressed as formulae in temporal logic. A model checker then accepts two inputs, a transition system and a temporal formula, and returns *true* if the system satisfies the formula and *false* otherwise. In this paper, we use the modal mu-calculus [41] in the usual extended form [72] as a branching temporal logic to express behavioural properties. The syntax of the extended mu-calculus (from now on, mu-calculus for short) is the following, where K ranges over sets of actions (i.e., $K \subseteq \mathcal{A}$) and Z ranges over variables:

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \wedge \phi \mid \phi \vee \phi \mid [K]\phi \mid \langle K \rangle \phi \mid \nu Z.\phi \mid \mu Z.\phi$$

A fixpoint formula may be either $\mu Z.\phi$ or $\nu Z.\phi$ where μZ and 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 (resp. νZ). A formula is *closed* if it contains no free variables. $\mu Z.\phi$ is the least fixpoint of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one. From now on we consider only closed formulae.

Scopes of fixpoint variables, free and bound variables, can be defined in the mu-calculus in analogy with variables of first-order logic.

The satisfaction of a formula ϕ by a state s of a transition system is defined as follows: each state satisfies $\mathtt{tt}$ and no state satisfies $\mathtt{ff}$; a state satisfies $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) if it satisfies ϕ_1 or (and) ϕ_2. $[K]\,\phi$ is satisfied by a state which, for every performance of an action in K, evolves to a state obeying ϕ. $\langle K \rangle\, \phi$ is satisfied by a state which can evolve to a state obeying ϕ by performing an action in K.

For example, $\langle a \rangle\, \phi$ denotes that there is an a-successor in which ϕ holds, while $[a]\, \phi$ denotes that for all a-successors ϕ holds.

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

A fixed point formula has the form $\mu Z.\phi$ ($\nu Z.\phi$) 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. $\mu Z.\phi$ is the least fixpoint of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one.

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

In the sequel we will use the following abbreviations:

$$\begin{aligned}
\langle \alpha_1, \ldots, \alpha_n \rangle \, \phi &= \langle \{\alpha_1, \ldots, \alpha_n\} \rangle \, \phi \\
\langle - \rangle \, \phi &= \langle \mathcal{A} \rangle \, \phi \\
\langle -K \rangle \, \phi &= \langle \mathcal{A} - K \rangle \, \phi \\
[\alpha_1, \ldots, \alpha_n] \, \phi &= [\{\alpha_1, \ldots, \alpha_n\}] \, \phi \\
[-] \, \phi &= [\mathcal{A}] \, \phi \\
[-K] \, \phi &= [\mathcal{A} - K] \, \phi
\end{aligned}$$

We provide some examples of logic properties. The simplest formulae are just those of modal logic:

$$\langle a \rangle \, \mathtt{tt}$$

means that "there is transition labelled by a".

With one fixpoint, we can talk about termination properties of paths in a transition system. The formula:

$$\mu Z. \, [a] \, Z$$

means that "all the sequences of a-transitions are finite".

The formula:

$$\nu Y. \, \langle a \rangle \, Y$$

means that "there is an infinite sequence of a-transitions".

We can then add a predicate p and obtain the formula:

$$\nu Y. p \wedge \langle a \rangle \, Y$$

saying that "there is an infinite sequence of a-transitions, and all states in this sequence satisfy p".

With two fixpoints, we can write fairness formulae, such as:

$$\nu Y. \mu X. (p \wedge \langle a \rangle \, Y) \vee \langle a \rangle \, X$$

meaning that "on some a-path there are infinitely many states where p holds".

Changing the order of fixpoints we get:

$$\mu X. \nu Y. (p \wedge \langle a \rangle \, Y) \vee \langle a \rangle \, X$$

saying "on some a-path almost always p holds."

In this paper, for ransomware detection, we will use mu-calculus logic formulae with the following pattern, where $1 \leq i < n$:

$$\begin{aligned}
\varphi_i &= \mu X. \, \langle a_i \rangle \, \varphi_{i+1} \vee \langle -a_i \rangle \, X \\
\varphi_n &= \mu X. \, \langle a_n \rangle \, \mathtt{tt} \vee \langle -a_n \rangle \, X
\end{aligned}$$

The formula φ_1 means that a sequence of actions $a_1, \ldots, a_n$ occurs in the software system under analysis, where each action a_i, with $1 \leq i \leq n$, can be preceded by any other actions, different from a_i.

In the paper, we will also use this other pattern, where $1 \leq i < n$:

$$\begin{aligned}
\psi_i &= \mu X. \, \langle a_i \rangle \, \psi_{i+1} \vee \langle -S_i \rangle \, X \\
\psi_n &= \mu X. \, \langle a_n \rangle \, \mathtt{tt} \vee \langle -S_n \rangle \, X
\end{aligned}$$

The formula ψ_1 means that a sequence of actions $a_1, \ldots, a_n$ occurs in the software system under analysis, where each action a_i, with $1 \leq i \leq n$, cannot be preceded by any actions in the set S_i. Finally, also the following pattern is of interest, where $1 \leq i < n$:

$$\begin{aligned}
\phi_i &= \mu X. \, \langle S_i \rangle \, \phi_{i+1} \vee \langle -S_i \rangle \, X \\
\phi_n &= \mu X. \, \langle S_n \rangle \, \mathtt{tt} \vee \langle -S_n \rangle \, X
\end{aligned}$$

The formula ϕ_1 means that a sequence of actions $b_1, \ldots, b_n$ occurs in the software system under analysis, where each action $b_i \in S_i$, with $1 \leq i \leq n$, cannot be preceded by any actions in the set S_i.

One of the most popular environments for verifying software systems is the Concurrency Workbench of New Century (CWB-NC) [24], which supports several different specification languages, among which CCS. In the CWB-NC the verification of temporal logic formulae is based on model checking [23]. In this paper we use CWB-NC as formal verification environment. The CWB-NC is available by North Carolina State University (NCSU) under a site license.

3.3 Motivation of the use of the CCS and of the mu-calculus logic

We think that the use of CCS is a viable solution, due to the availability of efficient formal verification methodologies. Furthermore, being a process algebra language, it has been proven valuable in the specification and design of software systems, for formal reasoning and for rapid prototyping. From the CCS textual specification it is possible to automatically generate the corresponding labelled transition system, which can be then used for model checking. We prefer textual syntax since it is more adapted to proof-writing and formal reasoning, as well as to the description of large-scale systems. Graphical notations are anyway complementary and can be used by user-friendly front-ends. Moreover, process algebraic operators are desirable to easily expressed code. Our approach exploits also the power of the mu-calculus logic which is able to recognize not only the presence of a given sequence, like the pattern-matching approach, but also checking a branching temporal logic formula, that permits to define a wide range of properties of

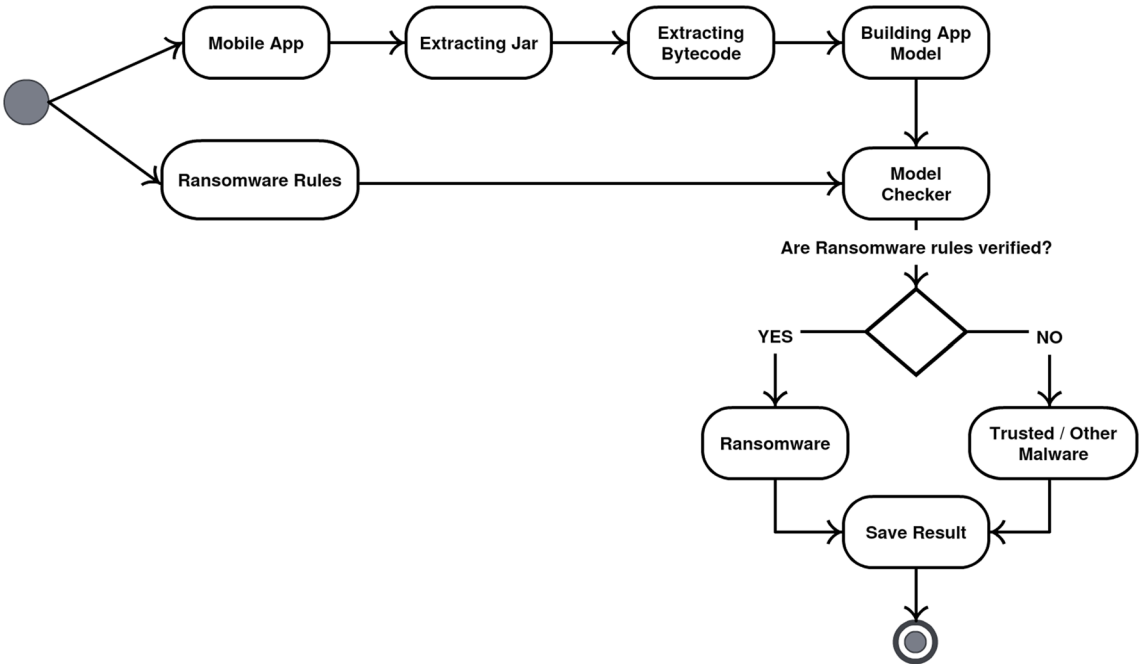

Fig. 1 UML activity diagram representing the detection ransomware apps methodology

program executions, security properties and more generally, invariant, liveness or safety properties. Thus, by using the mu-calculus logic it is also possible to detect not trivial sequences. For example, we are able to recognize two actions that are not syntactically consecutive, while this falls under pure pattern matching. Finally, our choices have been also dictated by the usage of the CWB-NC tool that supports mu-calculus as temporal logic and CCS as specification language.

4 Overall description of our method

In this section we present our method for the detection and the dissection of Android ransomware malware which applies model checking. As starting point, we represent the app behaviour as a CCS process. Then, from the CCS textual specification, using the CWB-NC, the corresponding labelled transition system is automatically generated.

The UML Activity diagram that describes all the phases of our method is shown in Fig. 1. The first phase is the construction of an automaton starting from the Java Bytecode of the app ("building app model" in Fig. 1). The automaton mimics the behaviour of the program. An Android application, the so-called *.apk* (i.e., Android Package) is a variant of the well-known *.jar* archive file. An *.apk* file typically contains the executable code for the Dalvik Virtual Machine (i.e., the *.dex* file), the resource folder (i.e., images, icons and sounds) and the Manifest file. Our method works at Bytecode level: we obtain Bytecode instructions starting from the *.apk* file by employing the tool chain shown in Fig. 1.

Since Android 5.0 Lollipop, Google replaced the Dalvik Virtual Machine (DVM) with ART (Android RunTime) [36]. The main difference between the ART and the DVM approach is represented by the fact that the first one considers the Just-In-Time (JIT) approach in which the compilation is performed on demand (i.e., the dex files are converted into their respective native representations only when the operating system needs them), while the second one employs the Ahead-Of-Time (AOT) approach, in which the dex files are compiled before they are demanded. Google considered this new approach in order to improve the performance and the battery life of mobile devices [54,55]. Considering that Talos works at bytecode level analyzing the dex file (used in both the approaches) and it does not require the application execution, its analysis is not affected by the different usage environment.

The process for transforming an apk file into a CCS process includes three steps:

- the *.jar* file is generated from the *.apk* file the with the dex2jar[1] tool;
- once obtained the *.jar* file, using the Java Archive Tool[2] utility, we extract the class files and directories from the *.jar* file related to the Android application with the only

[1] https://sourceforge.net/projects/dex2jar/.

[2] https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jar.html.

exception of the native code (i.e., the code able to invoke function compiled in C, C++ and assembly.);
- from the classes invoking the Byte Code Engineering Library[3] (i.e., Apache Commons BCEL) we retrieve the Bytecode of the Android application. With BCEL, classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions.

The automaton is obtained using an inference algorithm that for each Java Bytecode instruction produces a CCS process. Thus, the automaton representing the app is the standard transition system of the CCS processes. The nodes represent instruction addresses while the edges (labelled with opcodes) represent the control-flow transitions from one instruction to its successor(s). To give the reader the flavour of the approach followed, for example, suppose that at the address i we have the $goto\ j$ instruction. The instruction i is translated into a CCS process x_i that performs the action $goto_j$ and then jumps to the instruction address j (process x_j). In other words, we have the following transition:

$$x_i \xrightarrow{goto_j} x_j$$

Conditional jumps are instead specified as non-deterministic choices. Thus, in the same way all the Java Bytecode instructions are translated with CCS processes. The second phase of our method is the formal representation of the ransomware malicious behaviour (see "ransomware rules" in Fig. 1). We propose to specify malicious behaviours using temporal logic formulae where the actions range over the set of all the opcodes occurring in the program. Thus, this step tries to recognize specific and distinctive features of the ransomware behaviour with respect to all the other malware families and to the goodware too. This specific behaviour is written as a set of properties. To specify the properties, we manually inspected a few samples in order to find the different implementations of ransomware malicious behaviours at Bytecode level. Moreover, also ransomware technical reports have been analysed for obtaining additional information [46] [4], [5], [6]. In our approach, the mu-calculus logic [72] is used, which is a branching temporal logic to express behavioural properties.

In Table 3 we show two sample ransomware formulae to give the reader the flavour of the approach followed. The first one, i.e., φ_1, encodes the request to obtain admin privileges,

Table 3 Two formulae for ransomware detection

$$\varphi_1 = \mu X.\ \langle newandroidcontentIntent\rangle\ \varphi_{1_1} \vee \langle -newandroidcontentIntent\rangle\ X$$
$$\varphi_{1_1} = \mu X.\ \langle pushandroidappactionADDDEVICEADMIN\rangle\ \varphi_{1_2} \vee \langle -pushandroidappactionADDDEVICEADMIN\rangle\ X$$
$$\varphi_{1_2} = \mu X.\ \langle pushandroidappextraDEVICEADMIN\rangle\ \varphi_{1_3} \vee \langle -pushandroidappextraDEVICEADMIN\rangle\ X$$
$$\varphi_{1_3} = \mu X.\ \langle invokeputExtra\rangle\ \varphi_{1_4} \vee \langle -invokeputExtra\rangle\ X$$
$$\varphi_{1_4} = \mu X.\ \langle pushandroidappextraADDEXPLANATION\rangle\ \varphi_{1_5} \vee \langle -pushandroidappextraADDEXPLANATION\rangle\ X$$
$$\varphi_{1_5} = \mu X.\ \langle invokeputExtra\rangle\ \mathtt{tt} \vee \langle -invokeputExtra\rangle\ X$$
$$\varphi_2 = \mu X.\ \langle newjavaxcryptospecSecretKeySpec\rangle\ \varphi_{2_1} \vee \langle -newjavaxcryptospecSecretKeySpec\rangle\ X$$
$$\varphi_{2_1} = \mu X.\ \langle invokegetBytes\rangle\ \varphi_{2_2} \vee \langle -invokegetBytes\rangle\ X$$
$$\varphi_{2_2} = \mu X.\ \langle pushAES\rangle\ \varphi_{2_3} \vee \langle -pushAES\rangle\ X$$
$$\varphi_{2_3} = \mu X.\ \langle S\rangle\ \varphi_{2_4} \vee \langle -S\rangle\ X$$
$$\varphi_{2_4} = \mu X.\ \langle invokegetInstance\rangle\ \mathtt{tt} \vee \langle -invokegetInstance\rangle\ X$$
$$where\ S = \{pushAESCBCNoPadding, pushAESCBCPKCS5Padding, invokedecodevokeputExtra\}$$

while the second formula, i.e., φ_2, concerns the encryption. More precisely, φ_1 formula is able to catch the following behaviour:

- "$newandroidcontentIntent$" an intent in Android is an abstract description of an operation to be performed. It provides a facility for performing late run-time binding between the code in different applications;
- "$pushandroidappactionADDDEVICEADMIN$" the Device Administration API provides a public interface for managing policies enforced on the device. By invoking this API it is possible to write device admin applications that the user install on the device. The ADD_DEVICE_ADMIN action asks the user to add a new device administrator to the system;
- "$pushandroidappextraDEVICEADMIN$" and the desired policy is in the EXTRA_DEVICE_ADMIN field. This will invoke a user interface to bring the user through adding the device administrator to the system (or allowing them to reject it).
- "$pushandroidappextraADDEXPLANATION$" and "$invokeputExtra$" the developer can optionally include the EXTRA_ADD_EXPLANATION field to provide the user with additional explanation about what is being added but in several ransomware this message is used to deviate the user's attention.

On the other hand φ_2 means that the following behaviour occurred:

[3] https://commons.apache.org/proper/commons-bcel/.

[4] https://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf.

[5] https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2017.pdf (last visit: 18th December 2017).

[6] https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf (last visit: 18th December 2017).

- "$newjavaxcryptospecSecretKeySpec$" the invocation of the SecretKeySpec class provided by the javax.crypto package specifies a secret key. This class is only useful for raw secret keys that can be represented as a byte array and have no key parameters associated with them;
- "$invokegetBytes$": this method is invoked to retrieve the byte array used by the SecretKeySpec class to construct the secret key;
- "$pushAES$" The SecretKeySpec takes as argument a String variable containing the name of the secret key algorithm to be associated with the given key material that will be ciphered, in this case the developer uses the Advanced Encryption Standard (AES) algorithm. AES is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation, and it is fast in both software and hardware. It supports a 128-bit block cipher supporting keys of 128, 192, and 256 bits [29].
- "$pushAESCBCNoPadding$", "$pushAESCBCPKCS5Padding$", "$invokedecodevokeputExtra$", "$invokegetInstance$" the above sequence of instructions provides the functionality of a cryptographic cipher for encryption and decryption. In order to create a Cipher object, the application calls the Cipher's getInstance method and passes the name of the requested transformation to it. A transformation is a string that describes the operation (or set of operations) to be performed on the given input, to produce a certain output. A transformation always includes the name of a cryptographic algorithm (i.e., AES) and may be followed by the padding scheme. In the public key cryptography the padding is the process of preparing a message for encryption or signing using a specification or scheme such as [63].
  The transformation is defined in one of the two forms: "algorithm/mode/padding" or simply "algorithm". In our case two different transformations were defined: $AES/CBC/NoPadding$ and $AES/CBC/PKCS5Padding$, both with 128 bit key sizes. In the transformation AES represents the algorithm, CBC the Cipher Block Chaining Mode (as defined in [26]), PKCS5Padding the padding scheme (PKCS is the acronym for Public-Key Cryptography Standards and are specifications produced by RSA Laboratories inc., in this case it is used PKCS#5 that identifies the encryption standards with the use of a password [63]), while NoPadding expresses that the padding is not requested for the transformation.

As we stated in the introduction, the second formula is able to intercept ciphering operation that is the most discriminating behaviour that differentiates ransomware from the others existing malware. On the other hand, the first formula is able to catch the admin privileges behaviour employed recently by malware: as a matter of fact typically Android malware requires to run an external script by the asset folder to obtain root privileges, while in ransomware samples the application itself tries to obtain admin privileges without downloading an external resource.

Finally, the detection algorithm (see "model checker" in Fig. 1) receives two inputs, the automaton A that mimics the app, and the logic formula φ, representing the ransomware behaviour. Using a model checker, in particular the Concurrency Workbench of New Century (CWB-NC) [24], if A satisfies φ ($A \models \varphi$), then we consider the analysed app belonging to the ransomware family; otherwise, the app is either trusted or belonging to another malware family.

4.1 Localization of the payload

An important and novel feature of our method is that we can also achieve an automatic dissection of a mobile application, since it checks whether an app performs a specific behaviour, with an additional benefit: localization of the code of those instructions that implement the actions that determine the malicious behaviour. The payload can be localized without the help of a manual inspection. In fact, our solution is able to identify the exact position of those instructions characterizing the malicious behaviour. The method indicates the class file where the rule is verified, with a precision at method level, i.e., it is the CCS file that satisfies the formula. This is a very novel result in the malware analysis and it could be used from 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 byte, i.e., a signature.

Figure 2 shows an example of the CCS model obtained by applying our method starting from a Java Bytecode. For the sake of clarity, in Fig. 3 we have also reported the Java code related to the Java Bytecode at top box in Fig. 2.

Figure 4 shows four different implementations related to the same behaviour. We extracted the behaviour from four different applications belonging to the ransomware category. In particular, in these four samples, the identified behaviour is described by the φ_1 property of Table 3. As previously shown, our method is able to identify the Java class and the Java method where the payload is localized even when the behaviour is implemented in a different manner. In addition, using our method we localize the payload at method grain.

Belonging to different applications, the code snippets exhibit different package name and different implementation of the malicious method. We stress that also in these cases we are able to identify the malicious behaviour of the ransomware. The process is completely automatized, and for this reason our method is also useful to help malware analysts to perform automatic dissecting. Thus, we are able to know whether a sample has a particular malicious behaviour, without manual inspection.

```
public setDeviceAdmin() { //()V
    TryCatch: L1 to L2 handled by L3: java/lang/Throwable
    new android/content/Intent
    dup
    ldc "android.app.action.ADD_DEVICE_ADMIN" (java.lang.String)
    invokespecial android/content/Intent <init>((Ljava/lang/String;)V);
    astore1
    aload1
    ...
}
```

```
proc COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin0=newandroidcontentIntent
.COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin3

proc COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin3=dup
.COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin4

proc COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin4=pushandroidappactionADDDEVICEADMIN
.COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin6

proc COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin6=store
.COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin9

proc COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin9=load
.COREEFILETESTCLASSCOMASMBYASMADMEENpublicvoidsetDeviceAdmin10
```

Fig. 2 From Java Bytecode to CCS: an example

```
public void setDeviceAdmin()
  {
    Intent localIntent = new Intent("android.app.action.ADD_DEVICE_ADMIN");
    localIntent.putExtra("android.app.extra.DEVICE_ADMIN", mComponentName);
    localIntent.putExtra("android.app.extra.ADD_EXPLANATION",
        "To continue, you need activate the application. Click to activate / enable");
    localIntent.putExtra(new String(new AsmSmog().derooke("d277cb96862b96cf0db982c598d32d4e")), 3);
    ...
}
```

Fig. 3 Java code related to the Java Bytecode of Fig. 2

5 Evaluation of Talos

The method presented in the previous section has been implemented with the Talos tool. Talos is a Java system: the Java Bytecode is translated into CCS processes according to our method. Then Talos invokes the CCS model checker. The results of the model checker are fed back into the tool, which, in case of a positive answer, i.e., the rules are verified, states that the app belongs to the ransomware family. The tool is freely available at the following url (www.ing.unisannio.it/santone/Talos).

5.1 Goals of the evaluation

The validation consisted of three phases, each one aiming at verifying two properties of the proposed solution, which are:

1. the effectiveness of the ransomware detection.
2. the robustness with respect to the obfuscation.

In order to evaluate the effectiveness of ransomware detection, we collected a labelled dataset of malware (ransomware and other malware) and goodware. In order to evaluate the second property of our tool, we obfuscated all the samples belonging to the ransomware set and demonstrate that our method is resilient with respect to the obfuscation. Moreover, we enrich our analysis by comparing the performances obtained with our tool with those obtained by state-of-the-art tools for ransomware detection [2,17,69]. Finally, we compare the detection performance of our approach with that of the top 10 ranked mobile antimalware solutions from AVTEST[7], which is a benchmarking service of the quality of antimalware solutions and ranks them on the basis of a numerical evaluation.

The next sections describe all the experiments that have been conducted, compare Talos with the state-of-the-art tools and antimalware solutions for ransomware detection and discuss the advantages/disadvantages of our methodology.

5.2 Evaluation

To estimate the detection performance of our method, we compute the metrics of precision and recall, F-measure (Fm)

[7] https://www.av-test.org/en/antivirus/mobile-devices/.

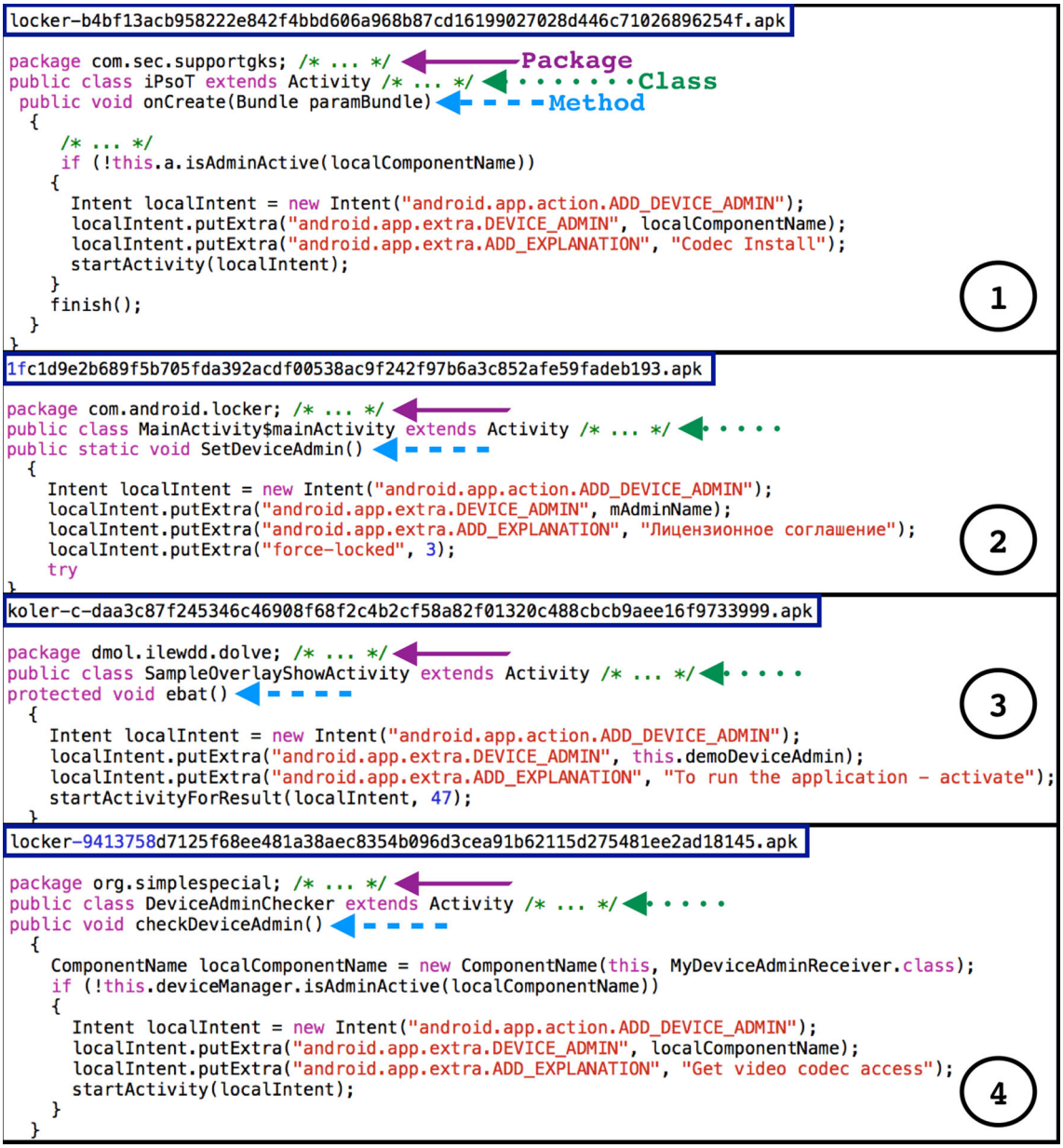

Fig. 4 Example of localization and dissection

and accuracy (Acc), defined as follows:

$$\mathrm{PR} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}; \quad \mathrm{RC} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}};$$
$$\mathrm{Fm} = \frac{2\mathrm{PR}\,\mathrm{RC}}{\mathrm{PR}+\mathrm{RC}}; \quad \mathrm{Acc} = \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{FN}+\mathrm{FP}+\mathrm{TN}}$$

where TP is the number of malware that was correctly identified in the ransomware family (true positives), TN is the number of malware correctly identified as not belonging to the ransomware family (true negatives), FP is the number of malware that was incorrectly identified in the ransomware family (false positives), and FN is the number of malware that was not identified as belonging to the ransomware family (false negatives).

5.3 The dataset

The real-world samples examined in the experiment were gathered from three different datasets. The first one is a collection of freely available samples of Android ransomware, formed by 672 taken from a source[8] and 11 taken from another source[9]. The samples are labelled as ransomware

[8] http://ransom.mobi/.

[9] http://contagiominidump.blogspot.it/.

Table 4 The dataset used in the experiment

Dataset	#Samples for category
Ransomware	683
Other malware	3872
Trusted	1500
Total	6055

and appeared from December 2014 to June 2015. The second one is the Drebin project's dataset [4,71], a very well-known collection of malware used in many scientific works, which includes the most widespread Android families.

Each malware sample in these datasets is labelled according to the *malware family* it belongs to: each family comprehends samples which have in common the same payload. This collection does not contain ransomware samples: we use this dataset to check the false positives and the true negatives.

The last one is a dataset of trusted applications crawled from the Google Play,[10] by using a script which queries a python API[11] to search and download apps. The downloaded applications belong to all the 26 different available categories (i.e., Books and Reference, Lifestyle, Business, Live Wallpaper, Comics, Media and Video, Communication, Medical, Education, Music and Audio, Finance and News, Magazines, Games, Personalization, Health and Fitness, Photography, Libraries and Demo, Productivity, Shopping, Social, Sport, Tools, Travel, Local and Transportation, Weather, Widgets). The applications retrieved were among the most downloaded in their category and were free.

The trusted applications were collected between April 2015 and January 2016 and were later analyzed with the VirusTotal service,[12] a service able to run 57 different anti-malware software (i.e., Symantec, Avast, Kasperky, McAfee, Panda, and others) on the app: the analysis confirmed that the crawled applications did not contain malicious payload. We use this dataset to check the false positives and the true negatives.

Table 4 provides the details of the full collection of 6055 samples used to test the effectiveness of our method. In order to test the capability of our rules to identify exclusively ransomware samples, we include in the dataset both trusted and malware samples from other families (respectively, *Trusted* and *Other Malware*).

The malware was retrieved from the Drebin project [4,71] (we take into account the top 10 most populous families). Table 5 shows the results obtained using our method.

[10] https://play.google.com.

[11] https://github.com/egirault/googleplay-api.

[12] https://www.virustotal.com/.

5.4 Results

Results in Table 5 seem to be very promising: we obtain an Accuracy equal to 0.99 and a F-measure equal to 0.97. Concerning the ransomware results, we are not able to identify the malicious payloads of just 3 samples on 683. This result, which reflects the strength of the proposed approach, is measured by the Recall (0.99).

In the following we discuss the reason why we obtain false positive and negatives: with regard to the false positive, we manually inspected the samples under analysis and we found that the samples contain the payload considered by the logic rule. In particular the verified rule is the φ_1 one explained in Sect. 4 i.e., the one expressing the ability to encode the request in order to obtain admin privileges, for this reason we think that these samples (downloaded from the Google Play) can be considered as vector of a malicious behaviour.

Relating the second one, we manually inspected the false negative samples and we found that the malicious payload was embedded into the samples, but it was split in the try-catch block (several instructions into the try section, while the remaining ones into the catch one). We think from the false negative analysis that the malicious payload is evolving by splitting their harmful code in different part of the application (including the exception catch blocks).

In order to measure Talos performances, we used the *System.currentTimeMillis()* Java method that returns the current time in milliseconds. Table 6 shows the performance of our method. In particular, we consider the overall time to analyse a sample as the sum of two different contributions: the average time required to extract the class files of the application using the dex2jar tool ($t_{dex2jar}$) and the time required to obtain the response from Talos ($t_{response}$). These two values are the average times, i.e., they are computed as the total time employed by Talos to process the samples divided the number of samples evaluated.

The machine used to run the experiments and to take measurements was an Intel Core i5 desktop with 4 gigabyte RAM, equipped with Microsoft Windows 7 (64 bit).

We highlight that the Talos performances do not depend on the application size. As a matter of fact, the total time required for a response is determined only by the size of the class files contained into the application under analysis. Table 7 shows seven different applications. We consider apps with size ranging between 1.2 and 20.2 MB. In Table 7 we report the following information:

– column **App** indicates the application name, for simplicity's sake we rename every application with an identification number. The apps are listed in increasing order taking into account the number of the states and the transitions of their CCS model;

Table 5 Performance evaluation

Formula	# Samples	TP	FP	FN	TN	PR	RC	Fm	Acc
Ransomware	6055	680	28	3	5344	0.96	0.99	0.97	0.99

Table 6 The performance evaluation (values are expressed in seconds)

$t_{dex2jar}$	t_{response}	Total time
89,937 s	1,569,376 s	1,659,313 s

– column **Size** reports in the three sub-columns **bin**, **apk** and **jar** the size of the folder containing the extracted classes, the size of the apk and the size of the jar, respectively;
– columns **States** and **Transitions** point out the size of the CCS model for each app, respectively, in terms of number of states and transitions;
– columns **Total Time** shows the time employed by Talos to verify the app.

As shown in Table 7 the size of the model, and consequently the related total time, is not depending by the apk size, but it depends only by the size of the folder containing the extracted classes from the jar file. This happens since our model is built starting from the class files. Indeed, the apk file contains also external resources such as images and sounds.

The last row of the table shows also a worst case of execution time related to our evaluation. In the worst case our approach reaches an execution time equal to 306 s.

5.5 Robustness with respect to the obfuscation

In order to demonstrate the effectiveness of our method in ransomware identification, we applied to the dataset a set of well-known code transformations [11,58] adopted by malware writers to evade the signature-based detection provided by the current antimalware technologies [59,77]. We report the transformation techniques used by the two tools used for modifying the samples of this experimentation: Droidchameleon [58] and the tool described in [11].[13]

1. *Disassembling and reassembling* The compiled Dalvik Bytecode in *classes.dex* of the application package may be disassembled and reassembled through *apktool*. This allows various items in a *.dex* file to be re-arranged or represented in a different way. In this way signatures relying on the order of different items in the *.dex* file will likely be ineffective with this transformation.
2. *Repacking* Every Android application has got a developer signature key that will be lost after disassembling the application and then reassembling it. Using the *signapk*[14] tool it is possible to embed a new default signature key in the reassembled app to avoid detection signatures that match the developer keys.
3. *Changing package name* Each application is identified by a unique package name. This transformation is aimed at renaming the application package name in both the Android Manifest and all the classes of the app.
4. *Identifier renaming* This transformation renames each package name and class name by using a random string generator, in both Android Manifest and *smali* classes, handling renamed classes' invocations.
5. *Data encoding* Strings could be used to create detection signatures to identify malware. To elude such signatures, this transformation encodes strings with a *Caesar cipher*. The original string will be restored during application run-time, with a call to a *smali* method that knows the *Caesar key*.
6. *Transform manifest* The manifest file presents essential information about your app to the Android system, information the system must have before it can run any of the app's code. Basically it describes the components of the application (i.e., the activities, services, broadcast receivers, and content providers that the application is composed) and which permissions the application must have in order to access protected parts of the API and interact with other applications. This transformation adds to the Manifest of the application the following permissions:
$android.permission.INTERNET$, that allows applications to open network sockets;
$android.permission.READ\_PHONE\_STATE$, that allows read only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls;
$android.permission.CHANGE\_WIFI\_STATE$, that allows applications to change Wi-Fi connectivity state;
and $android.permission.ACCESS\_WIFI\_STATE$, that allows applications to access information about Wi-Fi networks.
This is the unique transformation related to the Manifest xml file and not directly to source code: considering that there exist signature-based antimalware that calcu-

[13] https://github.com/faber03/AndroidMalwareEvaluatingTools.

[14] https://code.google.com/p/signapk/.

Table 7 Applications name, uncompressed classed extracted by the jar file (identified by the "bin" column), jar size, apk size, states, transitions and total time for seven applications

App	Size bin	jar	apk	States	Transitions	Total time (s)
1	29.7 kB	13.1 kB	4.6 MB	745	776	1
2	31.0 kB	11.0 kB	1.2 MB	2100	2178	2
3	108.3 kB	42.4 kB	3.5 MB	6201	6439	5
4	1.9 MB	932.3 kB	2.1 MB	103, 363	113,744	74
5	3.3 MB	1.6 MB	18.6 MB	179, 443	193,560	95
6	6.7 MB	3.2 MB	13.0 MB	453, 992	484,332	246
7	8.5 MB	4.1 MB	2.5 MB	530, 660	567,035	306

late the hash, using this simply morphing technique we are able to evade this kind of (trivial) identification.

7. *Call indirections* Some detection signatures could exploit the call graph of the application. To evade such signatures a transformation can be designed which mutates the original call graph, by modifying every method invocation in the *smali* code with a call to a new method inserted by the transformation which simply invokes the original method.
8. *Code reordering* This transformation is aimed at modifying the instructions order in *smali* methods. A random reordering of instructions has been accomplished by inserting *goto* instructions with the aim of preserving the original run-time execution trace.
9. *Junk Code Insertion* These transformations introduce those code sequences that have no effect on the function of the code. Detection algorithms relying on instructions (or opcodes) sequences may be defeated by this type of transformations.
10. *Encrypting payloads and native exploits* In Android, native code is usually made available as libraries accessed via JNI. However, some malware such as DroidDream also pack native code exploits meant to run from a command line in non-standard locations in the application package. All such files may be stored encrypted in the application package and be decrypted at run-time. Certain malware such as DroidDream also carry payload applications that are installed once the system has been compromised. These payloads may also be stored encrypted. Payload are categorized and encryption as DSA is exploited because signature-based static detection is still possible based on the main application's Bytecode. These are easily implemented and have been seen in practice as well (e.g., DroidKungFu malware uses encrypted exploit).
11. *Function outlining and inlining* In function outlining, a function is broken down into several smaller functions. Function inlining involves replacing a function call with the entire function body. These are typical compiler optimization techniques. However, outlining and inlining can be used for call graph obfuscation also.

Table 8 Between the transformation techniques implemented in the two tools

Transformation	Obfuscation engine	DroidChamelon
Dissassembling	X	X
Repacking	X	X
Changing package name	X	X
Identifier renaming	X	X
Data encoding	X	X
Transform manifest		X
Call indirections	X	X
Code reordering	X	X
Junk code insertion	X	X
Encrypting payloads		X
Function outlining		X
Reflection		X

12. *Reflection* The transformation converts any method call into a call to that method via reflection. After this a successive encryption of the method name is applied in order to make it more difficult for any static analysis to recover the call. Basically the transformation is able to convert any method call into a call to that method via reflection.

We apply the transformation set to our samples with Droidchameleon [58]. Furthermore, we use the framework[15] able to inject six different obfuscation techniques in Android applications. Table 8 shows the obfuscation techniques implemented by the two tools.

We combine together all the transformations provided by the two tools to generate stronger obfuscation. We have obfuscated the 683 ransomware of the dataset, reported in Table 4: Talos was only not able to identify the 3 samples that in the original form had not been detected. The remaining ransomware was correctly detected. The new results with the dataset of Table 4 including the 683 morphed ransomware samples (6738 samples overall) are shown in Table 9.

[15] https://github.com/faber03/AndroidMalwareEvaluatingTools.

Table 9 Results with the obfuscated samples

Formula	# Samples	TP	FP	FN	TN	PR	RC	Fm	Acc
Ransomware	6738	1360	28	6	5344	0.97	0.99	0.97	0.99

5.6 Comparison with the state-of-the-art tools for ransomware detection

In this section we compare the solutions provided by research community to detect ransomware in mobile environment: we discuss the methodologies provided in literature, and we present the results they obtained compared with ours.

Literature offers three methods to solve the problem of ransomware detection.

The first one is HelDroid [2], which analyses Android ransomware using both static and dynamic analysis. In particular, HelDroid uses static taint analysis and lightweight emulation to find flows of function calls that indicate device-locking or file-encryption behaviours. The approach to detect threatening behaviour is learning-based, natural language processing technique that recognizes menacing phrases. Although most of their analysis is static, the threatening-text detector does execute the sample in case no threatening text is found in the static files. Basically HelDroid proposes three generic indicators of compromise for detecting Android ransomware activity by recognizing its distinguishing features.

The first indicator is the Threatening Text Detector that uses text classification to detect coercion attempts. If the result of this classifier is positive, but the others are not, the sample is simply labelled as "scareware", i.e., the application limits itself to display some threatening text to convince the victim in performing some actions. If also the Encryption Detector and/or the Locking Detector (respectively, the second and the third indicators) are triggered, the application is actively performing either action on the infected device. In this case, the assigned label is "ransomware".

The second work is [17], which proposes two methods to discriminate Android malware application in the family they belong to using the Hidden Markov Model and the Structural Entropy methods.

The approach adopts HMMs to represent the statistical properties of the full malware dataset and of malware families. Trained HMMs can then be used to determine whether an application is similar to the malware families contained in the training set. The model is produced by collecting only the malware belonging to the ransomware family. They apply the Baum-Welch algorithm [5] for finding the values of the unknown parameters of the model. After this step, they compare a generic opcodes sequence (corresponding to the app to classify) with the opcodes sequence of the learned model: they use the Forward algorithm [5] for finding the likelihood of the sequence, i.e., the probability that the (learned) model generates the sequence to classify. If this is true, then the app is considered a malware instance.

The second method, the Structural Entropy one, works directly on the Dalvik executable file of the samples. As a matter of fact, the entropy-based method is based on the estimation of structural entropy of an Android executable (.dex file). The first step is the extraction of an entropy series: once the .dex file has been divided into blocks of fixed sizes, authors compute the Shannon entropy for each block. The second phase of the method is the comparison between the segments of two .dex files to compute a similarity score. The similarity score is based on the Levenshtein distance. This value represents the percentage of similarity between two .dex files based on the corresponding entropy areas.

The last work provided by literature is [69]: it proposes a technique to identify ransomware based on dynamic analysis that employ three modules: Configuration, Monitoring, and Processing.

The first module collects the information of priority protection area (i.e., PPA, the location of the files needed to be protected from the attacks of the ransomware), registers them to the watch list table for the monitoring module, and protects the corresponding files in real time.

The second module is responsible to detect the ransomware by monitoring the PPA area and the process. It monitors input/output events such as reading, writing, copying, and deleting of a file belonging to PPA. In addition it monitors processes, memory usage, I/O count, and storage to detect ransomware.

The aim of the last module is to stop the process suspicious of ransomware in the monitoring module and inquires users about the appropriate handling of the process. Once the handling is determined, the information of the corresponding process will be stored and used in the configuration module subsequently.

Table 10 shows the comparison in terms of samples correctly identified (true positive), number of ransomware samples (# Ransomware Samples) tested and Recall between the state-of-the-art tools and the method we propose.

We highlight that all the described methods use the same ransomware dataset, with the exception of Song et al. [69] who analyse only one application developed by the authors. Talos is able to detect 669 ransomware on a dataset composed by 672 samples (with a recall of 0.995); it overtakes the other methods in literature evaluating real-world malware. Note that the Structural Entropy-based method proposed in [17] identifies 628 True Positives and reaches a Recall equal to 0.935. Even if these results are little lower than ours, our

Table 10 Comparison among state-of-the-art tools in ransomware detection

Authors	Method	#Ransomware samples	True positive	Recall
Andronio et al. [2]	String matching	672	582	0.866
	HMM3	672	515	0.766
Canfora et al. [17]	HMM4	672	478	0.711
	HMM5	672	499	0.743
	Entropy	672	628	0.935
Talos	Model checking	672	669	0.995
Song et al. [69]	CPU and *I/O* usage	1 (self-made)	1	1

Table 11 Top 10 signature-based antimalware evaluation against Talos

Antimalware	Original			Morphed		
	%ident.	#ident.	#unident.	%ident.	#ident.	#unident.
AhnLab	13.76	94	589	9.95	68	615
Alibaba	0.44	3	680	0.44	3	680
Antiy	13.18	90	593	7.46	51	632
Avast	27.52	188	495	10.24	70	613
AVG	3.22	22	661	1.61	11	672
Avira	19.76	135	548	15.95	109	574
Baidu	14.34	98	585	9.95	68	615
BitDefender	28.26	193	490	19.18	131	552
ESET-NOD32	20.35	139	544	11.85	81	602
GData	27.96	191	492	11.27	77	606
Talos	99.56	680	3	99.56	680	3

method gains also in the robustness with respect to the obfuscation and in the localization of the payload.

5.7 Comparison with top 10 ranked mobile antimalware solutions from AVTEST

Finally, as a baseline for evaluating the performances of our solution, we compare the results obtained with Talos with those produced by the top 10 ranked mobile antimalware solutions from AVTEST[16], an independent Security Institute which each year provides an analysis of the performances of the antimalware software. We built this baseline, by submitting on January 2016 our original samples to the VirusTotal API[17], which allows to run the above-mentioned antimalware.

Table 11 shows the evaluation between the top 10 antimalware solutions and our method with the original ransomware samples and with the morphed ones.

We consider only the samples and the percentage identified in the right family (column "ident" and the percentage in column "%ident") in Table 11. We also report the samples detected as malicious but not identified in the right family and the samples not recognized as malware (column "unident").

With regard to Table 11 we notice that BitDefender shows better performances in family identification for *ransomware* original samples. Instead, with regard to morphed samples, antimalwares' performances decrease dramatically, indeed BitDefender is able to identify only 131 samples. The worst antimalware in *ransomware* identification is Alibaba, able to correctly classify just 3 original samples and 3 morphed samples.

Due to the novelty of the problem, antimalware solutions are still not specialized in family identification; this is the reason why most of antimalwares are unskilled to detect families. Another problem is that current antimalwares are not able to detect malware when the signature mutates: they fail to detect morphed samples. On the contrary, the detection done by Talos is barely affected by the code transformations, so it is independent from the signature.

5.8 Discussion

The advantages of our formal based methodology derive from the power of automated model checking to search a state space. Also the generation of the formal model is automated. Moreover, our method detects ransomware by recognizing behaviours characterizing certain ransomware families, rather than matching patterns or signatures. If we consider that malware evolves by merging or modifying

[16] https://www.av-test.org/en/antivirus/mobile-devices/.

[17] https://www.virustotal.com/.

existing malware [22,38,56,58], it is extremely likely to find known behaviours also in new and unknown malware. This means that this method is not a kind of over-fitting, but leverages behaviours' abstractions for detecting also new malware. Thus, our approach is able to handle new malware types and so to detect real malware found in the wild, when they have a behaviour similar to the analysed one. On the other side, the logic rule-set needs to be formed and defined. Writing the correct rules can be rather a complex task. Actually, we are studying an approach based on clone detection to automatically deduce the logic rules. 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 in [31]. UFI has the aim of simplifying the writing of the logic properties.

Another weakness of the proposed method is represented by the fact that the Talos research prototype considers in its analysis the bytecode extracted by the dex file of the mobile application excluding the native code: we plan to integrate also the native code in Talos analysis in order to make our properties verifiable also on C, C++ source code (through a reverse engineering process) and on JavaScript code.

Talos is currently a research prototype whose main aim is to demonstrate the detection' effectiveness of the proposed method; thus, the performances are not the core aspect at this point of development of Talos. Although the times we have obtained still seem to be high for an usage in the real world, it should be considered how Talos is intended to be applied. Talos was conceived to perform antimalware analysis on apps marketplace, before the publication of an app. In this scenario we can consider that: (i) the times already could not be a problem, as the analysis is not run on the device, so does not affect the usability; and (ii) the computational capability could be improved a lot, by using specific architectures on the back end of the marketplace's server.

Furthermore, Talos is able to localize the malware payload and this is the reason why it can be used by malware analysts as a tool to support the malicious behaviours discovery and localization avoiding the manual and time effort code inspection. This is useful for malware analysts to better understand the malicious behaviour implementation and to study the malicious payload evolution. As previously stated in order to make Talos able to recognize the malicious payload the logic formulae must be representative of the malware behaviour. In case the malicious payload is invoked through reflection, whether the invoked method exhibits the malicious payload expressed by the logic formulae, Talos is able to identify it. We highlight that Talos is not able to resolve the target of reflected method calls if these targets are stored encrypted or created dynamically.

As a future work we plan to inhibit the malicious code starting from its localization. For example, we can disable the malicious payload activation ensuring that the conditions that it triggers are never verified: in this case the malicious payload will become dead code that will never be called and/or triggered by any function.

6 Conclusion

Ransomware is not a new form of malware, but it appeared already in 2010, although in the last couple of years it has infected a very high number of smartphones and workstations around the world [2,6,34,46]. Considered the number of known infections, it is straightforward to conclude that the commercial antimalware is not effective to detect or block this threat. The problem of recognizing the ransomware is still open in literature, since there is a very little number of papers facing it. We have developed Talos based on formal methods which has the following advantages.

6.1 High precision in the detection of malicious behaviours

Talos is able to detect mobile ransomware with a very high accuracy (0.99). In particular, we define temporal logic rules in order to catch the malicious behaviour. Thus, a rule represents a single characteristic malicious behaviour that was previously codified as illicit or harmful. The rule contains the malicious actions performed by the malware in order to carry out the malicious behaviour.

6.2 Payload localization into the malware code

Talos allows us to know whether a rule is verified on the model which represents the mobile application. Then through the model checker tool, the ransomware rule is verified on the model. By this way, Talos can achieve an automatic dissection of a mobile application, since Talos checks whether an app performs a specific behaviour, with an additional benefit: Talos is able to localize into the code those instructions that implement the actions that determine the malicious behaviour.

6.3 Transparency to code obfuscation

Talos is robust with respect to the obfuscation. In the last years, obfuscation techniques have become popular and widely used in many commercial products. Namely, obfuscators create a program which is semantically equivalent to the original one, but syntactically different. Being behaviour-based, our method is transparent to obfuscation. This is an important result, since malware writers continuously make

use of obfuscation methods so that the malware may evade detections.

The method promises to open new roads to the detection of any kind of malware. For this reason our future work will concern the evolution of this method to the detection of botnet and rootkit. In addition we plan to explore the effectiveness of formal methods to identify emerging threats in IoT environment [60].

Acknowledgements The authors thank Gerardo Canfora for his valuable comments and suggestions.

Authors' contribution AC, FM, VN, AS and CAV are all responsible for the concept of the paper, the results presented and the writing. All the authors have read and approved the final published manuscript.

References

1. Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic analysis. J. Comput. Virol. **7**(4), 247–258 (2011)
2. Andronio, N., Zanero, S., Maggi, F.: Heldroid: Dissecting and detecting mobile ransomware. In: International Workshop on Recent Advances in Intrusion Detection, pp. 382–404. Springer (2015)
3. Annachhatre, C., Austin, T.H., Stamp, M.: Hidden markov models for malware classification. J. Comput. Virol. Hacking Tech. **11**(2), 59–73 (2015)
4. Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., Rieck, K.: Drebin: efficient and explainable detection of android malware in your pocket. In: Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS), IEEE (2014)
5. Attaluri, S., McGhee, S., Stamp, M.: Profile hidden markov models and metamorphic virus detection. J. Comput. Virol. **5**(2), 151–169 (2009)
6. Aurangzeb, S., Aleem, M., Iqbal, M.A., Islam, M.A.: Ransomware: a survey and trends. J. Inf. Assur. Secur. **6**(2), 48–58 (2017)
7. Battista, P., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.: Identification of android malware families with model checking. In: International Conference on Information Systems Security and Privacy, SCITEPRESS (2016)
8. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. J. Comput. Virol. **2**(1), 67–77 (2006)
9. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, ACM, pp. 15–26 (2011)
10. Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., Visaggio, C.A.: Effectiveness of opcode ngrams for detection of multi family android malware. In: 2015 10th International Conference on Availability, Reliability and Security (ARES), IEEE, pp. 333–340 (2015)
11. Canfora, G., Di Sorbo, A., Mercaldo, F., Visaggio, C. A.: Obfuscation techniques against signature-based detection: a case study. In: 2015 Mobile Systems Technologies Workshop (MST), IEEE, pp. 21–26 (2015)
12. Canfora, G., Medvet, E., Mercaldo, F., Visaggio, C.A.: Detecting android malware using sequences of system calls. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, ACM, pp. 13–20 (2015)
13. Canfora, G., Mercaldo, F., Moriano, G., Visaggio, C.A.: Composition-malware: building android malware at run time. In: 2015 10th International Conference on Availability, Reliability and Security (ARES), IEEE, pp. 318–326 (2015)
14. Canfora, G., Mercaldo, F., Visaggio, C.A.: A classifier of malicious android applications. In: 2013 Eighth International Conference on Availability, Reliability and Security (ARES), IEEE, pp. 607–614 (2013)
15. Canfora, G., Mercaldo, F., Visaggio, C.A.: Evaluating op-code frequency histograms in malware and third-party mobile applications. In: E-Business and Telecommunications, Springer, pp. 201–222 (2015)
16. Canfora, G., Mercaldo, F., Visaggio, C.A.: Mobile malware detection using op-code frequency histograms. In: Proceedings of International Conference on Security and Cryptography (SECRYPT) (2015)
17. Canfora, G., Mercaldo, F., Visaggio, C.A.: An hmm and structural entropy based detector for android malware: an empirical study. Comput. Secur. **61**, 1–18 (2016)
18. Carter, H., Mood, B., Traynor, P., Butler, K.R.B.: Secure outsourced garbled circuit evaluation for mobile devices. J. Comput. Secur. **24**(2), 137–180 (2015)
19. Chenette, S.: The ultimate deobfuscator. In: Proceedings of the ToorConX Conference (2008)
20. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Technical Report, DTIC Document (2006)
21. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (S&P'05), IEEE, pp. 32–46 (2005)
22. Cimitile, A., Mercaldo, F., Martinelli, F., Nardone, V., Santone, A., Vaglini, G.: Model checking for mobile android malware evolution. In: Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering, FormaliSE '17, Piscataway, NJ, USA, IEEE Press, pp. 24–30 (2017)
23. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2001)
24. Cleaveland, R., Sims, S.: The NCSU concurrency workbench. In: Alur, R., Henzinger, T.A. (eds.) CAV. Lecture Notes in Computer Science, vol. 1102. Springer, Berlin (1996)
25. di Vimercati, S.D.C., Foresti, S., Livraga, G., Samarati, P.: Data privacy: definitions and techniques. Int. J. Uncertain. Fuzziness Knowl. Based Syst. **20**(6), 793–818 (2012)
26. Dworkin, M.: Recommendation for block cipher modes of operation. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf (2001)
27. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M.: Android security: a survey of issues, malware penetration, and defenses. Commun. Surv. Tutor. IEEE **17**(2), 998–1022 (2015)
28. Feinstein, B., Peck, D., SecureWorks, I.: Caffeine monkey: automated collection, detection and analysis of malicious javascript. In: Black Hat, USA (2007)
29. FIPS. Advanced encryption standard. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (2001)
30. Ford, S., Cova, M., Kruegel, C., Vigna, G.: Analyzing and detecting malicious flash advertisements. In: Proceedings of the Computer Security Applications Conference, 2009. ACSAC'09. Annual. pp. 363–372. IEEE (2009)
31. Francesco, N.D., Santone, A., Vaglini, G.: A user-friendly interface to specify temporal properties of concurrent systems. Inf. Sci. **177**(1), 299–311 (2007)
32. Gharacheh, M., Derhami, V., Hashemi, S., Fard, S.M.H.: Detection of metamorphic malware based on hmm: a hierarchical approach. Int. J. Intell. Syst. Appl. **8**(4), 18 (2016)
33. Hallaraker, O., Vigna, G.: Detecting malicious javascript code in mozilla. In: 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), IEEE, pp. 85–94 (2005)

34. Hampton, N., Baig, Z.A.: Ransomware: emergence of the cyber-extortion menace In: Proceedings of the 13th Australian Information Security Management Conference, 2015. pp. 47–56. SRI Security Research Institute, Edith Cowan University (2015)
35. Hartstein, B.: Jsunpack: an automatic javascript unpacker. In: ShmooCon Convention (2009)
36. Jackson, W.: An introduction to the android application development platform. In: Android Apps for Absolute Beginners, Springer, pp. 61–99 (2014)
37. Jacob, G., Filiol, E., Debar, H.: Formalization of viruses and malware through process algebras. In: International Conference on Availability, Reliability and Security (ARES 2010), IEEE (2010)
38. Jang, J., Woo, M., Brumley, D.: Towards automatic software lineage inference. In: USENIX Security, pp. 81–96 (2013)
39. Kaspersky. Mobile malware evolution 2016. https://securelist.com/files/2017/02/Mobile_report_2016.pdf
40. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting Malicious Code by Model Checking. Springer, Berlin (2005)
41. Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. **27**, 333–354 (1983)
42. Li, J., Xu, M., Zheng, N., Xu, J.: Malware obfuscation detection via maximal patterns. In: Third International Symposium on Intelligent Information Technology Application, IITA 2009, vol 2, IEEE, pp. 324–328 (2009)
43. Likarish, P., Jung, E., Jo, I.: Obfuscated malicious javascript detection using classification techniques. In: MALWARE, Citeseer, pp. 47–54 (2009)
44. Liu, X., Liu, J.: A two-layered permission-based android malware detection scheme. In: 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), IEEE, pp. 142–148 (2014)
45. Maier, D., Müller, T., Protsenko, M.: Divide-and-conquer: why android malware cannot be stopped. In: 2014 Ninth International Conference on Availability, Reliability and Security (ARES), IEEE, pp. 30–39 (2014)
46. Mercaldo, F., Nardone, V., Santone, A.: Ransomware inside out. In: 2016 11th International Conference on Availability, Reliability and Security (ARES), IEEE, pp. 628–637 (2016)
47. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Download malware? No, thanks. How formal methods can block update attacks. In: 2016 IEEE/ACM 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE), IEEE (2016)
48. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Hey malware, I can find you! In: 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE Workshops 2016, Paris, June 13–15 (2016)
49. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Ransomware steals your phone formal methods rescue it. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Springer, pp. 212–221 (2016)
50. Mercaldo, F., Visaggio, C.A., Canfora, G., Cimitile, A.: Mobile malware detection in the real world. In: Proceedings of the 38th International Conference on Software Engineering Companion, ACM, pp. 744–746 (2016)
51. MGREffitas: In-the-wild ransomware protection comparative analysis 2016 q3. https://www.mrg-effitas.com/wp-content/uploads/2016/07/Zemana_ransomware_detection.pdf
52. Milner, R.: Communication and Concurrency. PHI Series in Computer Science. Prentice Hall, Upper Saddle River (1989)
53. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, IEEE, pp. 421–430 (2007)
54. Muttoo, S.K., Badhani, S.: Android malware detection: state of the art. Int. J. Inf. Technol. **9**(1), 111–117 (2017)
55. Oh, H.-S., Yeo, J.H., Moon, S.-M.: Bytecode-to-c ahead-of-time compilation for android Dalvik virtual machine. In: Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition, EDA Consortium, pp. 1048–1053 (2015)
56. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. ACM Trans. Progr. Lang. Syst. (TOPLAS) **30**(5), 25 (2008)
57. Preda, M.D., Giacobazzi, R.: Semantics-based code obfuscation by abstract interpretation. J. Comput. Secur. **17**(6), 855–908 (2009)
58. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ACM, pp. 329–334 (2013)
59. Rastogi, V., Chen, Y., Jiang, X.: Catch me if you can: evaluating android anti-malware against transformation attacks. IEEE Trans. Inf. Forensics Secur. **9**(1), 99–108 (2014)
60. Ren, K., Samarati, P., Gruteser, M., Ning, P., Liu, Y.: Guest editorial special issue on security for iot: the state of the art. IEEE Internet Things J. **1**(5), 369–371 (2014)
61. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, pp. 108–125 (2008)
62. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**(4), 639–668 (2011)
63. RSA. Pkcs #1 v2.2: Rsa cryptography standard. https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf (2012)
64. Saracino, A., Sgandurra, D., Dini, G., Martinelli, F.: MADAM: effective and efficient behavior-based android malware detection and prevention. IEEE Trans. Dependable Secure Comput. **PP**(99), 1–1 (2017). https://doi.org/10.1109/TDSC.2016.2536605
65. Song, F., Touili, T.: Efficient Malware Detection Using Model-Checking. Springer, Berlin (2001)
66. Song, F., Touili, T.: Pommade: pushdown model-checking for malware detection. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM (2013)
67. Song, F., Touili, T.: Model-Checking for Android Malware Detection. Springer, Berlin (2014)
68. Song, J., Han, C., Wang, K., Zhao, J., Ranjan, R., Wang, L.: An integrated static detection and analysis framework for android. Pervasive Mob. Comput. **32**, 1–11 (2016)
69. Song, S., Kim, B., Lee, S.: The effective ransomware prevention technique using process monitoring on android platform. Mob. Inf. Syst. **2016**, 1–9 (2016)
70. Sophos: The current state of ransomware. https://www.sophos.com/en-us/medialibrary/PDFs/technical
71. Spreitzenbarth, M., Echtler, F., Schreck, T., Freling, F.C., Hoffmann, J.: Mobilesandbox: looking deeper into android applications. In: 28th International ACM Symposium on Applied Computing (SAC), ACM (2013)
72. Stirling, C.: An introduction to modal and temporal logics for CCS. In: Yonezawa, A., Ito, T. (eds.) Concurrency: Theory, Language, And Architecture (LNCS), pp. 2–20. Springer, Berlin (1989)
73. Sung, A.H., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: 20th Annual Computer Security Applications Conference, IEEE, pp. 326–334 (2004)
74. Tan, D.J., Chua, T.-W., Thing, V.L., et al.: Securing android: a survey, taxonomy, and challenges. ACM Comput. Surv. (CSUR) **47**(4), 58 (2015)
75. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Secur. Priv. **5**(2), 32–39 (2007)

76. Yang, T., Yang, Y., Qian, K., Lo, D.C.-T., Qian, Y., Tao, L.: Automated detection and analysis for android ransomware. In: 7th International Symposium on Cyberspace Safety and Security (CSS), IEEE, pp. 1338–1343 (2015)

77. Zheng, M., Lee, P.P., Lui, J.C.: Adam: an automatic and extensible platform to stress test android anti-virus systems. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, pp. 82–101 (2012)