International Journal on Recent Researches in Science, Engineering & Technology (IJRRSET)
A Journal Established in early 2000 as National journal and upgraded to International journal in 2013 and is in existence for the last 10 years. It is run by Retired Professors from NIT, Trichy. Journal Indexed in JIR, DIIF and SJIF.
Research Paper
Available online at: www.jrrset.com
UGC Approved Journal No: 45483
Volume 7, Issue3 Pages 76-85

ISSN (Print) : 2347-6729
ISSN (Online) : 2348-3105
JIR IF : 2.54
SJIF IF : 4.334
Cosmos: 5.395

DETECTING SPYWARE BY IMITATING USER ACTIVITIES

Dr.G.Arul Dalton1,Dr.Muntha Raju, [2] Ms. Vishalakshmi , [3]Mr.K.Rajeev Reddy
[1]Professor, [2]Associate Professor , [3]Assist professor
[1,2,3]Dept of CSE , Shadan College of Engg & Tech

Abstract The success of any spyware is determined by its ability to evade detection. Although traditional detection methodologies employing signature and anomaly based systems have had reasonable success, new class of spyware programs emerge which blend in with user activities to avoid detection. One of the latest anti-spyware technologies consists of a local agent that generates honey tokens of known parameters (e.g., network access requests) and tricks spyware into assuming it to be legitimate activity. In this paper, as a first step, we address the deficiencies of static honey token generation and present an attack that circumvents such detection techniques. We synthesize the attack by means of data mining algorithms like associative rule mining. Next, we present a randomized honey token generation mechanism to address this new class of spyware. Experimental results show that (i) static honey tokens are detected with near 100% accuracy, thereby defeating the state-of the-art anti- spyware technique, (ii) randomized honey token generation mechanism is an effective anti-spyware solution

Keywords: Malware, Android, Security, Formal Methods, Temporal Logic.

1 INTRODUCTION

Mobile device currently permeate our every day acti- vity. From back transaction, to update the status on social networks, mobile devices allow us to perform a variety of activities. As a matter of fact, smartphone sales exceeded the current X86 PC platform in 2016, and this trend is expected to grow up in 20181.

Mobile devices quickly attracted the interest of the attackers, and it is easy to understand the reason why: if compared with PC platforms, in our smartphones are stored more and more sensitive and private infor- mation. Furthermore, smartphones manage the SIM card in which there is our credit, also for this reason this is an appealing attack surface for malicious soft- ware writers (Cimitile et al., 2018), (Mercaldo et al., 2016a).

Mobile operating systems producers tried to re- medy to this rampant spread of malicious software targeting mobile platform.

For instance, Google with the aim to consent the publication of a new app on Play Store (the official market for Android users) requires a deep scan of the app aimed to find possible malicious activities. Indeed the new app must be submitted to Bouncer (Oberheide and Miller, 2012), an automatic application scanning system introduces in 2012 with follo- wing distinctive features, including:

- static analysis in search of known threats;

it runs the software in a virtual emulator (QEMU) and identifies its behavior;
it starts and tracks the behavior of the app for 5 minutes;

- it explores the app in every button.

Bouncer performs a static analysis using the an- timalware provided by VirusTotal (a service able to evaluate the application simultaneously with 60 dif- ferent antimalware) but, considering the signature- based detection approach offered by current antimal- ware technologies, it is possible to mark a malicious sample as malware only whether their signature is sto- red into the antimalware repository (and consequently it is not possible to detect zero-day threat).

With regard to the dynamic analysis, the app is ran for a limited time window (5 minutes): in case the app does not exhibit the malicious behaviour in this period it passes this test. Furthermore, usually mal- ware is able to understand whether it is executed on a virtual environment (in this case it will not perform the malicious action, to avoid the sandbox detection).

For these reasons, it is easy from malicious writers to elude the current detection (Canfora et al., 2018;690
Cimitile et al., 2017; Mercaldo et al., 2016b; Canfora et al., 2015b).

The preferred target of mobile malicious software is represented by ourselves: this is the reason why usually mobile malware is able to secretly record phone calls, collect images, videos, text messages and even the GPS coordinates of the victims and send them to the attackers and, generally speaking, to spy the infected users (this is the reason why this kind of malicious software is called spyware).

This is the reason why in this paper we present a framework able to detect Android spyware. In parti- cular, we develop a model checking based framework identifying this kind of threat. Our solution is behavi- oural based since it is able to detect the malicious spy- ware using temporal logic formulae. The considered logic rules are the formal specification of the malici- ous behaviour performed by a spyware sample. The framework models an android application as a labeled transition system starting from its bytecode. Then, using a model checker tool, it verifies the specified malicious behaviour against the model of the appli- cation. The output of the model checker, and thus of our framework, is binary: it is equal to true when the formula is verified on the model and false otherwise. Our method considers an application under analysis as spyware if the output of model checker is equal to true.

The paper proceeds as follows: next section

introduces background concepts related to Model Checking and Mu-Calculus Logic exploited by the proposed framework, Section 3 describes our method aimed to detect Android spyware, Section 4 presents the performance evaluation of the proposed frame- work and, finally, conclusion and future work are dis- cussed in Section 6.

2 MODEL CHECKING AND Mu-Calculus LOGIC

Verification of a software or hardware system invol- ves checking whether the system in question beha- ves as it was designed to behave. Formal methods have been successfully applied to safety-critical sys- tems (Santone et al., 2013) and in other domains such as biology (Ruvo et al., 2015; Ceccarelli et al., 2014). One reason is the overwhelming evidence that for- mal methods do result in safer systems. In this pa- per we show that formal methods are extremely well- suited to spyware detection. First of all, in this section

we recall some basic concepts.

Model checking is an formal method for determi- ning if a model of a system satisfies a correctness specification (Clarke et al., 2001). A model of a system consists of a labelled transition system (LTS). A spe- cification or property is a logical formula. A model checker then accepts two inputs, a LTS and a tempo- ral formula, and returns true if the system satisfies the formula and false otherwise.

A labelled transition system comprises some num- ber of states, with arcs between them labelled by acti- vities of the system. A LTS is specified by:

- a set S of states;
- a set L of labels or actions;
- a set of transitions $T \subseteq S \times L \times S$.

Transitions are given as triples (start, label, end).

In this paper, to express proprieties of the system

we use the modal mu-calculus (Stirling, 1989) which is one of the most important logics in model checking. The syntax of the mu-calculus is the following, where K ranges over sets of actions (i.e., K L) and

Z ranges over variables:

$$\varphi \quad ::= \quad \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [K]\varphi \mid$$

$$\langle K\rangle\varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

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 con- tains 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 va- riables, can be defined in the mu-calculus in analogywith 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 ttand no state satisfies ff;

a state satisfies ϕ_1 ϕ_2 (ϕ_1 ϕ_2) if it satisfies $\phi_1$or (and) ϕ_2. $[K]$ ϕ is satisfied by a state $\vee$ which, $\wedge$ for every performance of an action in *K*, evolves to astate obeying ϕ. *K* ϕ is satisfied by a state whichcan evolve to a state obeying ϕ by performing an action in *K*.

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

The precise definition of the satisfaction of a clo-sed formula ϕ by a state *s* (written $s = \phi$) is given in Table 1.

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

International Journal on Recent Researches in Science, Engineering & Technology (IJRRSET)
A Journal Established in early 2000 as National journal and upgraded to International journal in 2013 and is in existence for the last 10 years. It is run by Retired Professors from NIT, Trichy. Journal Indexed in JIR, DIIF and SJIF.
Research Paper
Available online at: www.jrrset.com
UGC Approved Journal No: 45483

ISSN (Print) : 2347-6729
ISSN (Online) : 2348-3105
JIR IF : 2.54
SJIF IF : 4.334
Cosmos: 5.395

Table 1: Satisfaction of a closed formula by a state.

$p \models \mathtt{ff}$		
$p \models \mathtt{tt}$		
$p \models \varphi \wedge \psi$	iff	$p \models \varphi$ and $p \models \psi$
$p \models \varphi \vee \psi$	iff	$p \models \varphi$ or $p \models \psi$
$p \models [K]_R\, \psi$	iff	$\forall p'.\forall\alpha \in K.p \xrightarrow{\alpha}_{K\cup R} p'$ implies $p' \models \psi$
$p \models \square K \square_R\, \varphi$	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:

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

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

no free variables. µZ.ϕ is the least fix-point of the recursive equation Z = ϕ, while νZ.ϕ is the greatest one.
A transition system T satisfies a formula φ, writ- ten T = φ, if and only if q = φ, where q is the initial state of T .
In the sequel we will use the following abbreviati- ons:

We can then add a predicate p, and obtain the for- mula:
νY.p ∧⟨a⟩ 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 formu- lae, such as:

$$\begin{aligned}
\square\alpha_1,\ldots,\alpha_n\square\,\phi &= \square\{\alpha_1,\ldots,\alpha_n\}\square\,\phi\\
\square-\square\,\phi &= \square L\square\,\phi\\
\square-K\square\,\phi &= \square L-K\square\,\phi\\
[\alpha_1,\ldots,\alpha_n]\,\phi &= [\{\alpha_1,\ldots,\alpha_n\}]\,\phi\\
[-]\,\phi &= [L]\,\phi\\
[-K]\,\phi &= [L-K]\,\phi
\end{aligned}$$

We provide some examples of logic properties. The simplest formulae are just those of modal logic:
⟨a⟩ 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 for- mula:
meaning that "on some a-path there are infinitely many states where p holds".
Changing the order of fixpoints we obtain:
saying "on some a-path almost always p holds." In this paper we use CAAL (Concurrency Work-
bench, Aalborg Edition) (Andersen et al., 2015) as formal verification environment. It is one of the most popular environments for verifying systems. In the CAAL the verification of temporal logic formulae is based on model checking (Clarke et al., 2001).
means that "all the sequences of a-transitions are finite".
The formula:
νY. ⟨a⟩ Y
means "there is an infinite sequence of a- transitions". newpage

3 A FORMAL FRAMEWORK FOR SPYWARE DETECTION

In this section we describe our approach aimed to detect spyware Android applications. The approach models the Android application under analysis as a labelled transition system capturing the behaviour of

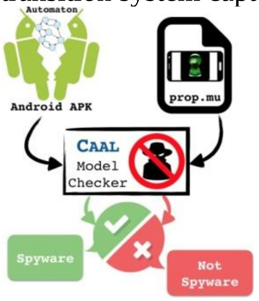


Figure 1: The proposed framework for mobile spyware de- tection and localization.

the app, and evaluates security temporal properties di- rectly on this LTS. Figure 1 shows the workflow of the proposed approach.

The proposed framework considers as inputs an Android application and a set of properties mobile spyware related. Through the model checker it is pos- sible to check whether one or more properties are ve- rified on the model representing the app under analy- sis: whether at least one property is verified, the pro- posed framework will mark the Android app as spy- ware, otherwise the app will be marked as not spy- ware (i.e., legitimate).

More specifically, the formal model of an Android application is a labeled transition system. It is built starting from the bytecode of the application and mi- mics the behaviour of the code. More precisely, every instruction is translated in a label and corresponds a transition between two states. Thus, the automaton si- mulates the normal execution of the instructions and a state transition is how to execute an instruction of the code. The if statement is modeled as an unconditional choice. Using a labeled transition system is also sim- ple to model a cycle, in fact, it is modeled as a branch (a transition) directed to a previous state of the code.

The construction of the labeled transition system is completely automatic. We have developed a trans- formation function able to convert the bytecode of an application into an automaton. This function is writ- ten in Java an is completely integrated in the frame- work.

Furthermore, our framework is also able to auto- matically calls the model checker tool in order to ve- rify the specified logic formulae on the formal mo- del. Summarizing, the workflow of the proposed fra- mework shown in Figure 1 is completely automatic. Starting from an Android application the framework automatically labels it as spyware or not, depending on the truth of the formula on the model.

3.1 Spyware Characterization through Temporal Logic Formulae

Temporal logic allows us to reason about changes in the behavior of a system over time, without explicitly mentioning specific instances of time. In particular, a formula may specify that some property eventually turns true, or always holds, or never turns true. In this section we use the mu-calculus logic to specify the spyware behaviour occurring in Android applications. We consider the model checking technique to de- tect spyware application for the following main rea-
sons:

The checking process is automatic. There is no need to construct a correctness proof.

The possibility of using the diagnostic counte- rexamples. If the specification is not satisfied, the model checker will produce a counterexample execution trace that shows why the specification does not hold. The counterexamples are invalua- ble in analyzing an application, since they can be use to understand where the spyware behaviour is in the application under analysis.

Temporal logic can easily and correctly express the behaviour of a spyware application.

There is no problem with partial specifications. It is unnecessary to completely specify all the appli- cation before beginning to model check proper- ties. Thus, model checking can be used only to verify part (methods) of the application.

Formal verification allows evaluating all possible scenarios, the entire state space all at once. Mo- del checking allows checking if, in each state, the system obeys certain properties. In particular, it allows verifying if the system under analysis ex- poses a certain behaviour expressed using a tem- poral logic formula. Spyware is a malware able to perform harmful actions in order to steal sensitive information. Basically, it is a software exposing in its code some malicious behaviours. Roughly speaking, in its code, there are some instructi- ons performing these actions. We can imagine this like a software specification: the software is designed to do something malicious. Now, ap- plying formal verification we investigate whether the software exhibits this malicious behaviour.

Table 2 shows an example of temporal logic formula written in mu-calculus logic. It catches the reading phone contacts suspicious behaviour. In Android environment the Content Provider al- lows reading phone contacts. In order to access to all contact information a ContentResolver object must be used. In our logic formula this operation is specified by the action invokegetContentResol- ver. After that it is necessary to communicate with the contacts applications performing a query to the URL of the contacts table (URI: Contact- sContract.Contacts.CONTENT URI). This step is specified in our logic formula by the sequence of actions: getstaticandroidproviderContactsContractContacts and invokequery. Finally the action invokegetString returns the contacts information as contact name, contact number, etc.

In order to better understand the behaviour speci- fied in our logic formula, we report the corresponding Java code snippet in Figure 2. In particular, the line highlighted in yellow shows the query to Content Pro- vider and the lines corresponding to get the contact in- formation (i.e., invocation of the getString method in Figure 2). Our logic formula specifies in mu-calculus logic the instructions show in Figure 2.

It should be underlined that we have formulated also the formula able to catch read phone contact for Android application with an API level less than or equal to 5. We have specify also the formula consi- dering the URI: Contacts.Phones, deprecated in API

level 5. The formula verified on the applications is κ. It is the logical disjunction between the formula con- sidering the API levels greater than the API level 5 (ξ) and the formula considering the other ones less than or equal to API level 5 (γ). In the following manner the formula covers all the Android API levels.

4 EXPERIMENTAL EVALUATION AND ASSESSMENT

In the following section, we detail how we generated the experimental dataset and we discuss the perfor- mances obtained by the proposed framework. In order to evaluate the effectiveness of the proposed method, we generated a set of Android spyware exploiting a framework able to automatically generate malicious samples: the Android Framework for Exploitation.

Android Framework for Exploitation

The Android Framework for Exploitation (i.e., AFE)2 is an open-source python-based project aimed to eva- luate Android vulnerabilities. It is composed by se- veral modules, we exploit the Malware Creator and the Stealer (able to inject code with the ability to steal

information from the attacked device including con- tacts, call logs, text messages and files from SD card).

Basically the Malware Creator module in order to inject the malicious behaviour implemented in the Steal module, it considers a pre-defined template able to embed the malicious payload (provided by the Steal module) and call it from a Service (declared in the An- droid Manifest file): the Service will be call when the Main activity is called (i.e., when the application is launched on the infected mobile device).

Basically, AFE considers following steps to auto- matically inject the malicious code into a legitimate applications: (i) it decompiles it into the smali lan- guage, (ii) the malicious payload is added and (iii) the app with the spyware behaviour is rebuilt.

Figure 3 depicts the difference between an An- droid application before and after the AFE injection.

As shown in Figure 3 in the injected version there is the xybot package added by AFE containing the spyware malicious payload.

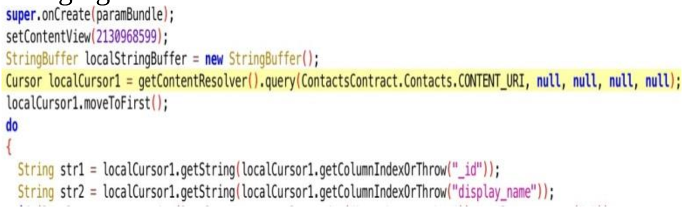
Figure 4 shows the classes included in the xybot package.

The main class responsible for the malicious be- haviours is com.xybox.infect.class (highlighted from a red circle in Figure 4): a java byte-code snippet be- longing to this class is shown in Figure 5.

From the snippet in Figure 5 it is possible to see the device contact gathering malicious action: as a matter of fact, basic contact information in Android are stored in Contacts table with detailed informa- tion stored in individual tables. The snippet shows a query to retrieve the records stored in Contact- sContract.Contacts.CONTENT URI3 (the instruction is highlighted by the red arrow).

Dataset Building

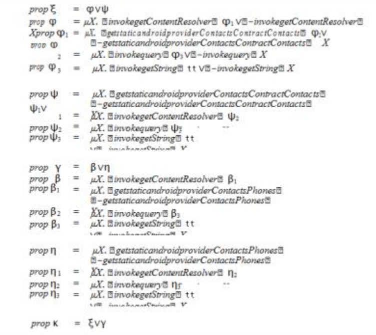
In order to evaluate the effectiveness of the propo- sed framework, a dataset composed by legitimate and spyware Android applications is considered. We col- lected 80 freely applications belonging to 26 diffe- rent categories from Google Play Store (i.e., Books and Reference, Lifestyle, Business, Live Wall- paper, Comics, Media and Video, Communication, Medi- cal, Education, Music and Audio, Finance and News, Magazines, Games, Personalization, Health and Fit- ness, Photography, Libraries and Demo, Productivity, Shopping, Social, Sport, Tools, Travel, Local and Transportation, Weather, Widgets). Their dimensions are ranging from 24 kB to 37 MB. We have selected an equal number of applications belonging to each



```
super.onCreate(paramBundle);
setContentView(2130968599);
StringBuffer localStringBuffer = new StringBuffer();
Cursor localCursor1 = getContentResolver().query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
localCursor1.moveToFirst();
do
{
  String str1 = localCursor1.getString(localCursor1.getColumnIndexOrThrow("_id"));
  String str2 = localCursor1.getString(localCursor1.getColumnIndexOrThrow("display_name"));
```

Figure 2: Code snippet able to access to contact information.

Table 2: Temporal logic formulae for Spyware detection.



prop ξ = $\varphi \vee \psi$
prop φ = $\mu X.$ □*invokegetContentResolver*□ $\varphi_1 \vee$□−*invokegetContentResolver*□
Xprop φ_1 = $\mu X.$ □*getstaticandroidproviderContactsContractContacts*□ $\varphi_2 \vee$
wov φ □−*getstaticandroidproviderContactsContractContacts*□ X
$_2$ = $\mu X.$ □*invokequery*□ $\varphi_3 \vee$□−*invokequery*□ X
prop φ_3 = $\mu X.$ □*invokegetString*□ tt $\vee$□−*invokegetString*□ X

prop ψ = $\mu X.$ □*getstaticandroidproviderContactsContractContacts*□
$\psi_1 \vee$ □−*getstaticandroidproviderContactsContractContacts*□
$_1$ = $\mu X.$ □*invokegetContentResolver*□ ψ_2
prop ψ_2 = $\mu X.$ □*invokequery*□ ψ_3
prop ψ_3 = $\mu X.$ □*invokegetString*□ tt
$\vee$□−*invokegetString*□ X

prop γ = $\beta \vee \eta$
prop β = $\mu X.$ □*invokegetContentResolver*□ β_1
prop β_1 = $\mu X.$ □*getstaticandroidproviderContactsPhones*□
□−*getstaticandroidproviderContactsPhones*□
prop β_2 = $\mu X.$ □*invokequery*□ β_3
prop β_3 = $\mu X.$ □*invokegetString*□ tt
$\vee$□−*invokegetString*□ X

prop η = $\mu X.$ □*getstaticandroidproviderContactsPhones*□
□−*getstaticandroidproviderContactsPhones*□
prop η_1 = $\mu X.$ □*invokegetContentResolver*□ η_2
prop η_2 = $\mu X.$ □*invokequery*□ η_3
prop η_3 = $\mu X.$ □*invokegetString*□ tt
$\vee$□−*invokegetString*□ X

prop κ = $\xi \vee \gamma$

category. The applications were downloaded in the time-window between March 2018 and April 2018.

We submitted the Play Store apps to the VirusTo- tal4 service: whether the 59 antimalware provided by VirusTotal marked as clean the application, we label the application as trusted.

To embed into the legitimate applications the spy- ware malicious behaviour we considered the AFE framework. For each applications downloaded from Play Store, through AFE a spyware version of the ap- plication was generated. We labeled the applications generated by AFE as spyware.4d

Furthermore, we generated an obfuscated version for each application submitted to the AFE framework using DroidChameleon tool (Rastogi et al., 2013). DroidChameleon applies code transformations to the smali code of the application under analysis. We con- sider obfuscated spyware to demonstrate that the pro- posed framework is resilient to the most widespread code obfuscation techniques implemented by mal- ware writers in order to elude the current signature ba- sed detection provided by antimalware technologies (usually ineffective against trivial code transformati- ons (Canfora et al., 2015a; Rastogi et al., 2014; Zheng et al., 2012)). As a matter of fact, antimalware soft-

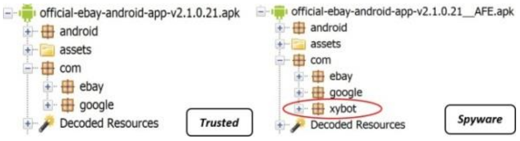


Figure 3: Android packages related to the trusted version of the official ebay application and the same application af- ter the AFE injection (with highlighted the xybot malicious package).

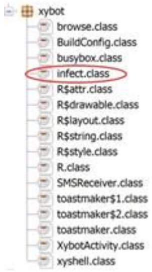


Figure 4: The classes belonging to the xybot package.

ware usually fail in the obfuscated malware recogni- tion since their detection mechanism is signature ba- sed and obfuscation techniques are considered to alter the code signature.

The samples generated with the AFE framework were injected with the following obfuscation techni- ques: (i) changing package name; (ii) identifier re- naming; (iii) data encoding; (iv) call indirection; (v) code reordering; (vi) junk code insertion.

At the end of this transformation process, we have collected 60 obfuscated applications which are a mor- phed version of spyware samples. It should be un- derlined that the number of morphed samples in less than the number of original once since in some cases DroidChameleon was not able to reassemble some of the selected samples, this is the reason why we had to discard them.

Summarizing 220 Android are included in the da- taset: 80 trusted apps, 80 spyware apps and 60 obfus- cated spyware apps.

Experimental Results

The dataset described above has been used to evaluate the proposed spyware detection framework. The re- sults achieved during the experimental evaluation are shown in Table 3.

As shown in Table 3, the proposed framework is able to correctly recognize the spyware samples and their morphed version. Regarding the trusted sam- ples our framework individuated 4 samples exposing suspicious spyware behaviour. We have manually in- spected the samples and we have found the suspicious behaviour to retrieve contacts. It should be underlined that only in one sample the read contacts suspicious behaviour is defined in the run method of a thread. In this case we can consider the sample under analysis as suspicious. In the other three samples the identi- fied behaviour is located in parts of code that seem harmless. Thus, in these cases we have to consider the identified samples as False Positive since our met- hod classified them as spyware but they seems to be trusted.

Table 3: Performance Evaluation.

Label	#Samples#	Identified Spyware	#Clean Samples
Trusted	80	4	76
Spyware	80	80	0
Morphed Spyware	60	60	0

Furthermore, the proposed method is able to lo- cate the code snippet where the logic formula results true. In particular, our framework provides as out- put both the label (spyware or not spyware) and, if the formula is true, the exact location in the code in terms of the method name, class name and packa- ges where the formula is resulted verified. In fact, from the localization results, it has emerged that all the spyware samples contain the malicious payload in the com.xybot.infect.class class (i.e., the class injected by the AFE framework).

It is worthy of note that for the 4 trusted sam- ples the logic formula turned out to be true in anot- her class belonging to another package different from com.xybot. In particular, during the analysis of spy- ware samples, the logic formula results verified in two different classes. Only in one application, it results verified on three classes.

With regard to the obfuscated versions of the spy- ware applications, the proposed framework was able to correctly identify as spyware all 60 morphed sam- ples.

In order to evaluate the obtained results we com- pute following metrics: Precision, Recall and F- Measure.

The precision has been computed as the propor- tion of the examples that truly belong to class X among all those which were assigned to the class. It is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved:

$$Precision = \frac{tp}{tp+fp}$$

where tp indicates the number of true positives and fp indicates the number of false positives.

The recall has been computed as the proportion

Figure 5: A java byte-code snippet related to the com.xybox.infect.class injected by the AFE framework.

of examples that were assigned to class X, among all the examples that truly belong to the class, i.e., how much part of the class was captured. It is the ratio of the number of relevant records retrieved to the total number of relevant records:

$$Recall = \frac{tp}{tp+fn}$$

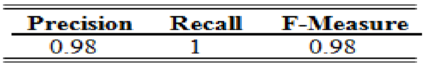
where tp indicates the number of true positives and fn indicates the number of false negatives.

The F-Measure is a measure of a test’s accuracy. This score can be interpreted as a weighted average of the precision and recall:

Precision Recall Precision+Recall

Table 4 shows the performances in terms of the
metrics we defined.

Table 4: Metrics Evaluation.



Precision	Recall	F-Measure
0.98	1	0.98

As shown in Table 4 the proposed framework is able to reach a precision value equal to 0.98, a recall value equal to 1 and an F-Measure of 0.98.

5 RELATED WORK

Several studies in current state of the art literature are mainly focused on generic mobile malware de- tection (Chen et al., 2016; Suarez-Tangil et al., 2017; Nix and Zhang, 2017; Duc and Giang, 2018). These works are mainly exploiting machine learning techni- ques by extracting distinctive features from samples under analysis to discriminate between malicious ap- plications and trusted ones. Contrarily, in this paper we investigate for a specific threat (i.e., the mobile spyware). Another difference with the these methods is that the proposed model checking based approach is behavioural: it models the code behaviour and then, it checks against it the temporal logic formulae by spe- cifying the malicious behaviour.

Shan et al. in (Shan et al., 2018) investigate about self-hiding behaviours (SHB), e.g. hiding the app, hi- ding app resources, blocking calls, deleting call re- cords, or blocking and deleting text messages. First of all the authors provide an in-deep characterization of SHB, then they present a suite of static analyses to detect such behaviour. They define a set of de- tection rules able to catch SHB. They test their ap- proach against more than 9,000 Android applications. Differently from the method we propose, authors are not mainly focused on spyware detection even if they define a set of rules able to detect specific behaviours. At the best of our knowledge the only work fo- cusing on Android spyware detection is the one pro- posed in (Chatterjee et al., 2018). Authors are focused in spyware used as intimate partner surveillance (IPS). The authors crawled apps from Google Play Store and using a combination of manual inspection and machine learning based approach discovered a large number of apps which are designed for legiti- mate use but also repurposed for IPS. Differently from this method we consider the model checking techni- que in order to identify spyware apps. Authors extract distinctive features from applications in order to apply machine learning based approach, instead, we define temporal logic formulae, which are behavioural ba- sed, to recognize Android spyware. Furthermore, we are focused about spyware with information gathering ability (i.e., the most widespread spyware in mobile environment (Wei et al., 2012)).

Zhang et al. in (Zhang et al., 2018) demonstrate that Google Assistant can be targeted since it suffers from some vulnerabilities. They develop an attacking framework able to record the voice of the user. This framework launches the attack using the recorded voice. This is a very dangerous vulnerability since the built-in voice assistant is able to access system re- sources and private information. Thus, hacking this assistant can lead to the leak of private and sensitive information. Differently, the proposed framework is able to recognize spyware applications in mobile en- vironment to stem these types of attacks.

6 CONCLUSION AND FUTURE WORK

Nowadays smartphones collect a large amount of per- sonal information. This is the reason why malware writers target these devices. More specifically, there is a kind of malicious software aiming to steal and collect these sensitive information and it is known as spyware.

Thus, in this paper we described a spyware de- tection framework. We exploit model checking technique and we use temporal logic formulae to de- tect Android spyware. We generated a synthetic data- set injected by spyware malicious payload in order to evaluate the effectiveness of the proposed method.

As future work, we plan to extend the experi- mental dataset including applications belonging from third-party marketplaces. We want also largely inves- tigate for many other applications belonging to the Android official market. Thus, we want to perform an in-deep analysis of the applications available in the stores. Furthermore, also secure information analysis will be investigated (Avvenuti et al., 2012).

Furthermore, we intend to compare our approach with other solutions proposed in literature, for exam- ple the approach proposed by (Chatterjee et al., 2018).

7 ACKNOWLEDGMENT

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

REFERENCES

1. Andersen, J. R., Andersen, N., Enevoldsen, S., Hansen,
2. M. M., Larsen, K. G., Olesen, S. R., Srba, J., and Wortmann, J. K. (2015). CAAL: concurrency work- bench, aalborg edition. In Theoretical Aspects of Computing - ICTAC 2015 - 12th International Col- loquium Cali, Colombia, October 29-31, 2015, Pro- ceedings, volume 9399 of Lecture Notes in Computer Science, pages 573–582. Springer.
3. Avvenuti, M., Bernardeschi, C., De Francesco, N., and Masci, P. (2012). JCSI: A tool for checking secure information flow in java card applications. Journal of Systems and Software, 85(11):2479–2493.
4. Canfora, G., Di Sorbo, A., Mercaldo, F., and Visag- gio, C. A. (2015a). Obfuscation techniques against signature-based detection: a case study. In 2015 Mobile Systems Technologies Workshop (MST), pages 21–26. IEEE.
5. Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., San- tone, A., and Visaggio, C. A. (2018). Leila: formal tool for identifying mobile malicious behaviour. IEEE Transactions on Software Engineering.
6. Canfora, G., Mercaldo, F., Moriano, G., and Visaggio, C. (2015b). Composition-malware: Building android malware at run time. pages 318–326. cited By 12.
7. Ceccarelli, M., Cerulo, L., and Santone, A. (2014). De novo reconstruction of gene regulatory networks from time series data, an approach based on formal met- hods. Methods, 69(3):298–305. cited By 10.

8. Chatterjee, R., Doerfler, P., Orgad, H., Havron, S., Palmer, J., Freed, D., Levy, K., Dell, N., McCoy, D., and Ris- tenpart, T. (2018). The spyware used in intimate part- ner violence. In 2018 IEEE Symposium on Security and Privacy (SP), pages 441–458. IEEE.

9. Chen, S., Xue, M., Tang, Z., Xu, L., and Zhu, H. (2016). Stormdroid: A streaminglized machine learning-based system for detecting android malware. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pages 377–388. ACM.

10. Cimitile, A., Mercaldo, F., Martinelli, F., Nardone, V., San- tone, A., and Vaglini, G. (2017). Model checking for

11. mobile android malware evolution. In Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering, pages 24–30. IEEE Press.

12. Cimitile, A., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2018). Talos: no more ransomware victims with formal methods. International Journal of Information Security, 17(6):719–738.

13. Clarke, E. M., Grumberg, O., and Peled, D. (2001). Model checking. MIT Press.

14. Duc, N. V. and Giang, P. T. (2018). Nadm: Neural network for android detection malware. In Proceedings of the Ninth International Symposium on Information and Communication Technology, pages 449–455. ACM.

15. Mercaldo, F., Nardone, V., and Santone, A. (2016a). Ran- somware inside out. In Availability, Reliability and Security (ARES), 2016 11th International Conference on, pages 628–637. IEEE.

16. Mercaldo, F., Visaggio, C., Canfora, G., and Cimitile, A. (2016b). Mobile malware detection in the real world. pages 744–746. cited By 13.

17. Nix, R. and Zhang, J. (2017). Classification of android apps and malware using deep neural networks. In 2017 International Joint Conference on Neural Networks (IJ- CNN), pages 1871–1878.

18. Oberheide, J. and Miller, C. (2012). Dissecting the android bouncer. SummerCon2012, New York.

19. Rastogi, V., Chen, Y., and Jiang, X. (2013). Droidchame- leon:evaluating android anti-malware against transfor- mation attacks. In ACM Symposium on Information, Computer and Communications Security, pages 329– 334.

20. Rastogi, V., Chen, Y., and Jiang, X. (2014). Catch me if you can: Evaluating android anti-malware against transformation attacks. IEEE Transactions on Information Forensics and Security, 9(1):99–108.

21. Ruvo, G., Nardone, V., Santone, A., Ceccarelli, M., and Ce- rulo, L. (2015). Infer gene regulatory networks from time series data with probabilistic model checking. pages 26–32. cited By 11.

22. Santone, A., Vaglini, G., and Villani, M. (2013). Incremen- tal construction of systems: An efficient characteriza- tion of the lacking sub-system. Science of Computer Programming, 78(9):1346–1367. cited By 14.

23. Shan, Z., Neamtiu, I., and Samuel, R. (2018). Self-hiding behavior in android apps: detection and characteriza- tion. In Proceedings of the 40th International Confe- rence on Software Engineering, ICSE 2018, Gothen- burg, Sweden, May 27 - June 03, 2018, pages 728–739.

24. Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In Yonezawa, A. and Ito, T., editors, Concurrency: Theory, Language, And Architecture, volume 491 of LNCS, pages 2–20. Springer.

25. Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Gi- acinto, G., and Cavallaro, L. (2017). Droidsieve: Fast and accurate classification of obfuscated android mal- ware. In Proceedings of the Seventh ACM on Confe- rence on Data and Application Security and Privacy, pages 309–320. ACM.

26. Wei, T.-E., Jeng, A. B., Lee, H.-M., Chen, C.-H., and Tien, C.-W. (2012). Android privacy. In Machine Learning and Cybernetics (ICMLC), 2012 International Confe- rence on, volume 5, pages 1830–1837. IEEE.

27. Zhang, R., Chen, X., Lu, J., Wen, S., Nepal, S., and Xiang,

28. Y. (2018). Using ai to hack ia: A new stealthy spy- ware against voice assistance functions in smart pho- nes. arXiv preprint arXiv:1805.06187.

29. Zheng, M., Lee, P. P., and Lui, J. C. (2012). Adam: an au- tomatic and extensible platform to stress test android anti-virus systems. In International Conference on Detection of Intrusions and Malware, and Vulnerabi- lity Assessment, pages 82–101. Springer.