Model Checking for Mobile Android Malware Evolution

Aniello Cimitile*, Francesco Mercaldo†, Fabio Martinelli†, Vittoria Nardone*, Antonella Santone*, Gigliola Vaglini†

*Department of Engineering, University of Sannio, Benevento, Italy
†Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy

{cimitile, vnardone, santone}@unisannio.it
{fabio.martinelli, francesco.mercaldo}@iit.cnr.it
†Department of Information Engineering, University of Pisa, Pisa, Italy
gigliola.vaglini@unipi.it

Abstract—Software engineering researchers have largely demonstrated that newer versions of software make use of previous versions. No exception to this rule for the so-called malicious software, that frequently evolves in order to evade the detection by antimalware. As matter of fact, mobile malicious programs, such as trojans, are frequently related to previous malware through evolutionary relationships. Discovering those relationships and constructing a phylogenetic model is expected to be helpful for analyzing new malware and for establishing a principled naming scheme. In this paper we propose a model checking based approach to infer mobile malware phylogenetic trees. We demonstrate, implementing our approach in the droid-Sapiens tool, that mobile malware families come from an ancestor and they influence own descendant, basing on the payload that they exhibit.

Keywords—security; malware; evolution; phylogenesys; model checking

I. INTRODUCTION

Systematic code reuse has been an elusive goal for software engineering practice since the term was first coined: it represents the use of existing software, or software knowledge, to build new software, following the reusability principles [10].

Malware, as any software, evolves. However new malware is seldom created from scratch: more frequently they are generated, with the help of automatic tools [23], using third-party libraries and code borrowed from existing malicious software [14] that is accessible by a robust network for malware code exchange. In last years, we are witnessing the growing phenomenon of mobile apps, i.e., applications developed to run on devices such as smartphones or tablets. At June 2016, there were 2.2 million available apps at Google Play store and two billion apps available in the Apple’s App Store1.

As matter of fact, with particular regards to mobile environment, malware frequently evolves due to rapid modify-and-release cycles, creating numerous strains of a common form [9], [22]. The result is a tangled network of derivation relationships between malicious programs. In biology such a relationship network is called a “phylogeny”; significant recent efforts in bioinformatics involve automatically constructing meaningful phylogeny models based on information in nucleotide, protein, or gene sequences [18]. Reconstructing malware phylogenies using automatic techniques is expected to help in forensic malware analysis [13]. It could provide clues for the analyst, particularly in terms of understanding how new samples relate to previously seen samples. Useful phylogenies could also serve as a principled basis for naming malware. Many efforts have been performed by antimalware vendors to protect users from the mobile threats. The main problem related to signature-based malware detection is represented by the fact that a signature must be contained in the antimalware repository in order to recognize the threats. However, in the frenetic world of mobile malware due to the rapid development of new releases of infections every day, it is impossible to avoid the spread of infections considering that the signature extraction process is laborious and the new sample must be widespread in order to be analyzed and added to repository. This is a very appealing scenario for malware writers, that develop code more and more aggressive to infect users, modifying the malicious action (i.e., the so-called payload) to continuously evade the detection provided by current antimalware technologies making useless the extracted signature. Starting from these considerations, in order to help the mobile malware analyst to efficiently identify the relationship between mobile malware samples to reduce the signature extraction time, in this paper we propose a model checking based approach to build evolutionary relationship between Android malware families. A malware family groups together a set of malware programs with common behaviors and properties that are common to all its member [27]. We infer a phylogenetic tree by analyzing system call traces extracted using dynamic analysis, because malware authors usually try to hide the derivation relationships adopting several techniques, including garbage insertion, code reordering, and instruction substitution that typically invalidate static analysis [20], [19] not dynamic one [5]. From the system call traces we generate a formal model used to verify properties characterizing the malware behavior. In this paper we use the Calculus of Communicating Systems (CCS) [21] to specify the model and the selective mu-calculus logic [3] to express

1https://www.statista.com/topics/1002/mobile-app-usage/
properties. The results of the verification help to identify the common malware behavior shared by the malware samples and to infer the ancestor-descendant relationships between them. The premise of our work is that an expanded perspective on mobile malware behavior and in particular the relationships between malware families will lead the development of more effective countermeasures.

The rest of the paper is organized as follows: the next section illustrates the proposed approach to track phylogenetic tree related to mobile malware; the third section discusses the evaluation; the fourth section discusses the current literature about malware phylogenesis and, finally, conclusions are given in the last section.

II. THE METHOD

In this section we present our methodology for the analysis and detection of Android malware evolution using model checking. We suppose the reader is familiar with the Calculus of Communicating Systems (CCS) [21] and model checking. We briefly recall only the selective mu-calculus logic.

A. Selective Mu-Calculus Logic

In this paper we use the selective mu-calculus, introduced in [3], which is a branching temporal logic to express behavioral properties of systems. The syntax of the selective mu-calculus is the following, where $K$ and $R$ range over sets of actions, while $Z$ ranges over a set of variables:

$$\phi ::= \top | \bot | Z \vee \phi | \phi \wedge \phi | \langle K \rangle_R \phi | \langle K \rangle_R \phi | \nu Z.\phi | \mu Z.\phi$$

The satisfaction of a formula $\phi$ by a state $s$ of a transition system, written $s \models \phi$, is defined as follows: each state satisfies $\top$ and no state satisfies $\bot$; a state satisfies $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) if it satisfies $\phi_1$ (and) $\phi_2$. $\langle K \rangle_R \phi$ and $\langle K \rangle_R \phi$ are the selective modal operators: they require that the formula $\phi$ is verified after the execution of an action of $K$, provided that it is not preceded by any action in $K \cup R$. More precisely: (i) $\langle K \rangle_R \phi$ is verified by a behavior expression which, for every performance of a sequence of actions not belonging to $R \cup K$, followed by an action in $K$, evolves in a behavior expression obeying $\phi$. (ii) $\langle K \rangle_R \phi$ is verified by a behavior expression which can evolve to a behavior expression obeying $\phi$ by performing a sequence of actions not belonging to $R \cup K$ followed by an action in $K$.

As in standard mu-calculus, $\mu Z.\phi$ is the least fixpoint of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one.

B. Model Checking for Malware Evolution

We propose a formal based methodology to infer mobile malware evolution and to find ancestor-descendant relationships between malware samples implemented in the droid-Sapiens tool. The tool is freely available by contacting one of the authors. The workflow of the method that we are going to propose is summarized in Fig. 1. We now analyze each process of our methodology in details.

Generation of system call execution traces. The aim of this process is to capture and store, in a textual format, the system call traces generated by Android applications. To this aim, the APK is installed and started on an Android device emulator. Successively, the $BOOT\_COMPLETED$ event is generated and sent to the emulator more that once and the correspondent sequence of system calls is gathered. We select this event because is able to activate the majority of widespread malware payloads [27]. The process is handled by a set of shell scripts developed by authors that perform the following actions for each application: (i) starting the target Android device emulator; (ii) installing the application on the device emulator; (iii) waiting until a stable state of the device is reached (i.e., when $epoll\_wait$ is executed and the application waits for user input or a system event to occur); (iv) starting the capture of system call traces; (v) sending the $BOOT\_COMPLETED$ event to the application; (vi) capturing system calls made by the application until a stable state is reached; (vii) stopping the system call capture and save the captured system call trace; (viii) stopping the Android device and revert its disk to a clean baseline snapshot.

XES-based Event Stream Generation. The process aims to clean, filter and convert the system call traces collected in textual format in the previous process into eXtensible Event Stream (XES)-compliant log format (IEEE XML-based standard for event logs$^2$). The extracted traces are in a textual format that needs to be cleaned, filtered and converted to an XES event stream. During this conversion, all the unnecessary information, as the system call arguments, is filtered out.

Property based reduction. In this process we exploit the powerful of the selective mu-calculus logic to reduce the traces expressed in XES format. The behavior of a system expressed with the execution traces is observed in an “abstract” way, disregarding all the non-interesting actions occurring in the traces. This abstraction can be used, in the next process, as a method for efficiently verifying properties of formal models. In fact, while properties are usually checked on transition systems describing the complete behavior of a process, many properties depend only on the behavior of the system with respect to a small subset of actions: thus, they can be more efficiently checked on a transition system which contains only the actions in this subset and which behaves, with respect to them, as the original one. The abstraction is easily defined since, given a selective mu-calculus logic formula $\varphi$, expressing the property we want to verify, the set of interesting actions, called $\rho$, is composed of all the actions syntactically occurring in $\varphi$. The actions in $\rho$ are the only actions relevant for checking the formula $\varphi$ and different transition systems that behave in the same way w.r.t. $\rho$ satisfy all the formulae with occurring actions in $\rho$. Thus, given $\rho$ and a set of traces, the abstraction function deletes the non-interesting actions from the traces; the reduced transition system we produce maintains, w.r.t. $\rho$, the behavior of the original system.

Model Discovery. From the abstracted traces we generate a formal model, i.e., a CCS process, that can be used to analyze and infer the evolution of mobile malware. We use a

general syntactic transformation function $T$ which transforms system call execution traces into a CCS model. The reader can refer to [25] for more details about the function $T$. Roughly speaking, the function $T$ consists of an iterative procedure that starts from an initial raw main process, which includes all the system traces as alternative branches. At each step, a more compact process is obtained, through (sub)processes merge and reduction. The model described by the CCS processes is simpler and more compact than one directly given as a transition system; moreover, all model checking environment can easily obtain the corresponding transition system.

**Formal analysis of malware evolution.** This last process aims at discovering Android malware evolution through model checking of selective mu-calculus formulae. In our approach, we invoke the Concurrency Workbench of New Century (CWB-NC) [7] as formal verification environment. Since CWB-NC is no longer in active development, as future work we want to substitute CWB-NC with CAAL (standing for Concurrency workbench developed at AALborg university) [1], which supports CCS as input specification language (as CWB-NC), but uses a more efficient algorithm to perform model checking. Actually, there exist mature tools with modern designs like CADP [12] with expressive input languages and efficient analysis methods. However, the aim of this paper is to develop an initial rapid research prototype to evaluate our methodology. The idea of our method is to formally infer derivation relationships or other aspects of evolution. More precisely, the core of our method is as follows:

**First step.** Initially, we recognize the distinctive features of the malware behaviour with respect to a specific malware family. This specific behaviour is written as a set of properties expressed in selective mu-calculus logic. To specify the properties, we manually inspected a few samples in order to find the malware malicious behavior implementation at system call execution traces level. We want to find relationships between malware. Thus, we check each formula $\varphi_F$ characterizing the specific family $F$ on each app of all the families of a fixed dataset. We report, for each formula, the number and the percentage of the apps that satisfy $\varphi_F$. From this analysis we consider the family $X$ as “ancestor” of the family $Y$ if the formula $\varphi_X$, characterizing the family $X$, is true on more than the 35% of the apps belonging to $Y$. The choice of this value comes from experimental evaluations. The reason is that if “many” apps, belonging to the family $Y$, satisfy the formula characterizing the family $X$ it means that the family $Y$ share a common malware behavior and that $X$ can be considered as an ancestor of $Y$.

**Second step.** To validate the previous result, the logic formulae which identify specific malware behaviors of a family are combined to confirm the right evolutionary relationships among families. For example, if $X$ is the “ancestor” of the family $Y$, we also check the formula $\varphi = \varphi_X \lor \varphi_Y$ on all the apps and we compare the results of this verification with those of the verification of $\varphi_X$. It must hold that the percentage of the apps of the family $X$ that satisfy $\varphi$ is very close to the percentage of the apps of the family $X$ that satisfy $\varphi_X$, meaning that the family $Y$ introduces new malware behavior that family $X$ did not have, thus no many other apps are kept by the introduction of the formula $\varphi_Y$ in the formula $\varphi$. The same holds for the other families different from $Y$. On the other hand, the percentage of the apps of the family $Y$ that satisfy $\varphi$ must be much bigger than the percentage of the apps of the family $Y$ that satisfy $\varphi_X$, indicating that a new malware behaviour has been introduced in the family $Y$ captured by the formula $\varphi_Y$.

As shown in the next section, experimental results suggest that the approach is remarkably accurate and deals efficiently with consistent databases of Android malware samples. The novelty of the approach is the use of temporal logic formulae to infer malware evolution, differently from [15] where vector similarity is used or from [6], [8], where graph matching is adopted. To the Authors’ knowledge, model checking has never used before for the malware evolution. The main distinctive features of the approach proposed in this paper are: (i) the use of formal methods to infer malware evolution;
Following the methodology explained in the previous section, we construct the Table II which shows, for each formula $\varphi_F$ (where $F$ is a family of our dataset), the number (resp. the percentage) of apps satisfying $\varphi_F$. Note that when verifying the formula $\varphi_F$, the higher percentage (in bold in the table) is obtained on the family $F$, confirming that the formula really characterizes the family.

From the analysis of Table II, we infer the malware families phylogenetic tree, as described in Fig. 2. In our experiment, it holds that Geinimi is the ancestor of both Plankton and DroidKungFu, since $\varphi_G$ is true on more that the 35% of the apps of these families. Moreover, for the same reason, we can infer that DroidKungFu is the ancestor of Opfake and Opfake is the ancestor of FakeInstaller, while Plankton has no descendant. This is also confirmed by the discovery date of Table I.

The phylogenetic tree is also validated by the analysis of malicious behaviours characterizing the families. As matter of fact, Plankton has no descendant because it represents the first example of update-attack malware, i.e., malware that do not embed the malicious payload at installation time but it will be downloaded at run-time. This behaviour is not represented in other families and for this reason Plankton does not exhibit descendants. His ancestor is Geinimi because, in according to [27] both Plankton and Geinimi make an extensive use of network resources to steal sensitive information using a botnet without using SMSs to fraud credit to victims. Also DroidKungFu uses network resource without send fraud SMSs, but differently to Geinimi (and Plankton), it employs the exploit and the RATC/Zimperich techniques to escalate the privileges of infected devices. In addiction to the information sent by Geinimi family, DroidKungFu sends the OS/operator/network type and the information stored in the SD/phone memory to multiple locations. For these reasons, we consider the Plankton and DroidKungFu at the same tree level while Geinimi is their ancestor because the Geinimi family was the first in our dataset to introduce the exhaustive usage of network resources (the SMS fraud will appear later) to collect location coordinates, the IMEI and the IMSI. Differently, Plankton introduces a similar kind of infection using the upload-attack and the DroidKungFu family adds to this attack three different privilege escalation techniques but using the Geinimi technique to embed the payload.

Opfake family samples are descendants of DroidKungFu family because they introduce the server-side metamorphism and the ability to send out premium-rated SMSs from a user’s mobile phone without consent but using network resource in an exhaustive way to send information and to download a fake version of Opera browser passing itself off as the installer for a legitimate application.

FakeInstaller family persists to use the network with the usual aim to send sensitive information, but in addition it sends SMS to fraud victims as Opfake samples. In addition to the Opera browser (as Opfake family), it has spoofed the Olympic Games Results App, Skype, Flash Player, and other applications.

### III. The Evaluation

In this section we present the preliminary results of the experiments we performed in order to demonstrate the effectiveness of our method to track mobile malware phylogenies. The results have been obtained through the prototype tool droidSapiens. We apply our method on a malware dataset composed by 858 samples grouped in 5 malware families as Table I shown. The table reports the families involved in the evaluation, with details about the number of the samples belonging to each family (#samples column) and the discovery date with the month (when available) and the year (date column).

We retrieved the Android malware applications from both Genoma [27] and Drebin [2] dataset.

The dataset quality was verified using the VirusTotal service\(^3\) consisting of 57 antimalware running on the dataset applications. This service allows to filter the malware applications that were not recognized as malicious from at least five antimalware. The common characteristics of the malware applications belonging to a specific family are captured by a logic formula. In particular, we have 5 logic formulae for the 5 families of our dataset: $\varphi_G$ for the Geinimi family, $\varphi_F$ for the Plankton family, $\varphi_{DKF}$ for the DroidKungFu family, $\varphi_{OF}$ for the Opfake family, $\varphi_{FI}$ for the FakeInstaller family.

The above formulae have been specified after a manual inspection of a few samples in order to find the malware malicious behavior at system call execution traces level. For example, property $\varphi_F$ is the following: $\langle \text{writev} \rangle \varphi \langle \text{fstat64} \rangle \varphi \langle \text{tt} \rangle$, meaning that it is possible that a writev action occurs followed by the action fstat64, where writev system call writes buffers of data to the file associated with the file descriptor and fstat64 gets information about a file with the request of the permissions on all of the directories in paths that lead to that file. For lack of space we omit the definition of all the other formulae, but all the formulae describe the malicious behavior of the respective family.

### TABLE I

THE DATASET.

<table>
<thead>
<tr>
<th>Family</th>
<th>#samples</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geinimi</td>
<td>73</td>
<td>12-2010</td>
</tr>
<tr>
<td>Plankton</td>
<td>81</td>
<td>06-2011</td>
</tr>
<tr>
<td>DroidKungFu</td>
<td>183</td>
<td>08-2011</td>
</tr>
<tr>
<td>Opfake</td>
<td>423</td>
<td>2013</td>
</tr>
<tr>
<td>FakeInstaller</td>
<td>98</td>
<td>2014</td>
</tr>
</tbody>
</table>

\(^3\)http://virustotal.com
Let us consider, for example, Geimini and Plankton. We have two families belonging to the relations ancestor-descendant. The first block of rows concerns the formulae relating results are reported in Table III. The table is divided in three phylogenetic tree. The combined formulae and the verification combine the formulae of Table II to validate the inferred ancestor family behavior).

According to the second step of our methodology, we combine the formulae of Table II to validate the inferred phylogenetic tree. The combined formulae and the verification results are reported in Table III. The table is divided in three blocks. The first block of rows concerns the formulae relating two families belonging to the relations ancestor-descendant. Let us consider, for example, Geimini and Plankton. We have inferred that Geimini is the ancestor of Plankton. Thus, we verify the formula \( \varphi_G \lor \varphi_P \) and we compare the results of its verification with those of the verification of the formula \( \varphi_G \) (the ancestor), see the upper graphic in Fig. 3. The comparison shows that Plankton is really an evolution of Geimini. In fact, on the apps belonging to the Geimini family, we have an increase of only 7% between the apps that verify \( \varphi_G \) and those that verify \( \varphi_G \lor \varphi_P \) (82% for \( \varphi_G \) and 89% for \( \varphi_G \lor \varphi_P \)), as expected. The reason is that Plankton introduces new malware behavior that Geimini did not have, thus no many other apps are kept by the formula \( \varphi_G \lor \varphi_P \). Also a little increment can be found when verifying and comparing the above two formulae for the other families different from Plankton, i.e., DroidKungFu, OpFake and FakeInstaller. In this case the reason is that no relationship has been found among them and Plankton, see Fig. 2. On the other hand, on the apps belonging to Plankton family the increase is significant (over than 25%), stating that a new malware behaviour has been introduced in Plankton family captured by the formula \( \varphi_P \).

The other three formulae of the first block of Table III of the form \( \varphi_X \lor \varphi_Y \), with a ancestor-descendant relationship, have a similar trend, since Geimini is the ancestor of DroidKungFu, DroidKongFu is the ancestor of OpFake and OpFake is the ancestor of FakeInstaller, as shown in Fig. 2.

Let us consider the second block of rows of Table III. In this case we analyze formulae like \( \varphi_X \lor \varphi_Y \), where no relationship of ancestor-descendant has been found between \( X \) and \( Y \), like for example Plankton/DroidKongFu, Plankton/OpFake, or Plankton/FakeInstaller. As an example, let us consider the lower graphic in Fig. 3 which shows this situation for \( X = \text{Plankton} \) and \( Y = \text{DroidKongFu} \). Now, we verify \( \varphi_X \lor \varphi_Y \) and we compare the results with \( \varphi_X \). The verification results highlight that: (i) a significant increment for the apps belonging to the DroidKongFu family; (ii) an increment for the apps belonging to the OpFake family, which is the descendant of DroidKongFu; (iii) a little increment for the other families.

Finally, in the last row of Table III the formula \( \varphi_G \lor \varphi_{DKF} \lor \varphi_{OF} \lor \varphi_{FI} \) is satisfied by more than 70% by all the families confirming that: (i) “Geimini-DroidKungFu-OpFake-Fakeinstaller” belongs to the ancestor-descendant line of the tree in Fig. 2; (ii) Plankton is the descendant of the Geimini family.

Table IV shows the time verification for all samples of our dataset. With \( T_{chk} \), we refer to the time employed to retrieve system calls (i.e., 60 seconds for each application), with \( T_{mod} \) to the time required to build the model and with \( T_{chk} \) to the time to verify the properties. The \( T_{TOT} \) value is the sum of all these contributes. 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).

**Table II**

**Formal analysis of malware evolution - First step.**

<table>
<thead>
<tr>
<th>Formulae</th>
<th>Geimini</th>
<th>Plankton</th>
<th>DroidKungFu</th>
<th>OpFake</th>
<th>FakeInstaller</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi_G )</td>
<td>60 (82%)</td>
<td>38 (46%)</td>
<td>72 (39%)</td>
<td>135 (34%)</td>
<td>14 (14%)</td>
</tr>
<tr>
<td>( \varphi_P )</td>
<td>5 (6%)</td>
<td>12 (6%)</td>
<td>20 (4%)</td>
<td>20 (4%)</td>
<td>13 (13%)</td>
</tr>
<tr>
<td>( \varphi_{DKF} )</td>
<td>54 (66%)</td>
<td>13 (16%)</td>
<td>155 (36%)</td>
<td>16 (16%)</td>
<td></td>
</tr>
<tr>
<td>( \varphi_{OF} )</td>
<td>20 (27%)</td>
<td>23 (28%)</td>
<td>55 (30%)</td>
<td>229 (54%)</td>
<td>45 (45%)</td>
</tr>
<tr>
<td>( \varphi_{FI} )</td>
<td>18 (24%)</td>
<td>15 (18%)</td>
<td>51 (27%)</td>
<td>140(33%)</td>
<td>51 (52%)</td>
</tr>
</tbody>
</table>
Several studies are focused on the malware phylogeny topic and, generally speaking, on malware evolution one: in this section we discuss both of them. In [27], authors manually analyze more than 1,200 malware samples to cover the majority of existing Android malware families, ranging from their debut in August 2010 to recent ones in October 2011. Their characterization and a subsequent evolution-based study of representative families reveal that they are evolving rapidly to circumvent the detection from existing mobile antivirus software. Ramu [22] provides an overview of evolution for mobile malware, attack vectors, detection methodologies and defense mechanisms, concluding that are still in its infancy stage. He highlights that mobile malware classes have some similarity with PC malware, mobile devices have unique characteristics that can be targeted by attackers. Relating to research phylogeny-oriented, in [15] a method to realize phylogeny models is proposed. Considering a set of features, called n-perms, authors match possibly permuted code in order to obtain a malware tree. Basing on the observation of the obtained malware tree authors suppose that phylogeny models based on n-perms are able to support the identification of new malware variants and the reconciling of inconsistencies in malware naming. The described method is static and hence it does not require malware execution. Indeed, with respect to our method, this approach is less robust to code obfuscations that cannot be represented as permutations, while our method, performing a dynamic analysis, is resilient to code obfuscations. Authors in [26] discuss a framework defining malware evolution relations in terms of path patterns on derivation graphs. The limitation of this approach is that the model’s definition of source code excludes machine generated code. This represents a restriction, considering that usually malicious code is automatically generated from the existing malicious code. In [17], authors propose a data-centric approach based on packets inspections aiming to automatically identify shellcode similarity. They basically generate a shellcode phylogeny for a given vulnerability. The main difference with respect to our method is represented by the fact that we execute the samples in order to perform a more deep analysis. The Process Mining-based approach proposed in [4] consists in analyzing system call traces gathered from Android malware and trusted applications to identify a set of relationships and recurring execution patterns that characterize their respective behavior. Researchers in [16] collect logs derived from instructions executed, memory and register modifications in order to build the phylogenetic tree at runtime. Furthermore, they show that network resources were useful for visualizing short non-equivalent code metamorphism than trees. They consider

### TABLE III
**Formal analysis of malware evolution - Second Step.**

<table>
<thead>
<tr>
<th>Formulae</th>
<th>Geinimi (73)</th>
<th>Plankton (81)</th>
<th>DroidKungFu (183)</th>
<th>OpFake (423)</th>
<th>FakeInstaller (98)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varphi_G \lor \varphi_P )</td>
<td>65 (89%)</td>
<td>59 (72%)</td>
<td>81 (44%)</td>
<td>136 (32%)</td>
<td>27 (27%)</td>
</tr>
<tr>
<td>( \varphi_G \lor \varphi_{DKF} )</td>
<td>65 (89%)</td>
<td>50 (61%)</td>
<td>152 (83%)</td>
<td>286 (67%)</td>
<td>30 (30%)</td>
</tr>
<tr>
<td>( \varphi_{DKF} \lor \varphi_{OF} )</td>
<td>28 (38%)</td>
<td>30 (37%)</td>
<td>151 (82%)</td>
<td>250 (59%)</td>
<td>60 (61%)</td>
</tr>
<tr>
<td>( \varphi_{OF} \lor \varphi_{FI} )</td>
<td>21 (28%)</td>
<td>23 (28%)</td>
<td>61 (33%)</td>
<td>248 (58%)</td>
<td>51 (52%)</td>
</tr>
<tr>
<td>( \varphi_P \lor \varphi_{DKF} )</td>
<td>13 (17%)</td>
<td>65 (80%)</td>
<td>150 (81%)</td>
<td>173 (40%)</td>
<td>16 (16%)</td>
</tr>
<tr>
<td>( \varphi_P \lor \varphi_{OF} )</td>
<td>21 (28%)</td>
<td>62 (76%)</td>
<td>65 (35%)</td>
<td>230 (54%)</td>
<td>58 (59%)</td>
</tr>
<tr>
<td>( \varphi_P \lor \varphi_{FI} )</td>
<td>19 (26%)</td>
<td>60 (74%)</td>
<td>61 (33%)</td>
<td>141 (33%)</td>
<td>64 (65%)</td>
</tr>
<tr>
<td>( \varphi_G \lor \varphi_{DKF} \lor \varphi_{OF} \lor \varphi_{FI} )</td>
<td>70 (95%)</td>
<td>57 (70%)</td>
<td>160 (87%)</td>
<td>335 (79%)</td>
<td>76 (77%)</td>
</tr>
</tbody>
</table>

### IV. RELATED WORK

Several studies are focused on the malware phylogeny topic and, generally speaking, on malware evolution one: in this section we discuss both of them. In [27], authors manually analyze more than 1,200 malware samples to cover the majority of existing Android malware families, ranging from their debut in August 2010 to recent ones in October 2011. Their characterization and a subsequent evolution-based study of representative families reveal that they are evolving rapidly to circumvent the detection from existing mobile antivirus software. Ramu [22] provides an overview of evolution for mobile malware, attack vectors, detection methodologies and defense mechanisms, concluding that are still in its infancy stage. He highlights that mobile malware classes have some similarity with PC malware, mobile devices have unique characteristics that can be targeted by attackers. Relating to research phylogeny-oriented, in [15] a method to realize phylogeny models is proposed. Considering a set of features, called n-perms, authors match possibly permuted code in order to obtain a malware tree. Basing on the observation of the obtained malware tree authors suppose that phylogeny models based on n-perms are able to support the identification of new malware variants and the reconciling of inconsistencies in malware naming. The described method is static and hence it does not require malware execution. Indeed, with respect to our method, this approach is less robust to code obfuscations that cannot be represented as permutations, while our method, performing a dynamic analysis, is resilient to code obfuscations. Authors in [26] discuss a framework defining malware evolution relations in terms of path patterns on derivation graphs. The limitation of this approach is that the model’s definition of source code excludes machine generated code. This represents a restriction, considering that usually malicious code is automatically generated from the existing malicious code. In [17], authors propose a data-centric approach based on packets inspections aiming to automatically identify shellcode similarity. They basically generate a shellcode phylogeny for a given vulnerability. The main difference with respect to our method is represented by the fact that we execute the samples in order to perform a more deep analysis. The Process Mining-based approach proposed in [4] consists in analyzing system call traces gathered from Android malware and trusted applications to identify a set of relationships and recurring execution patterns that characterize their respective behavior. Researchers in [16] collect logs derived from instructions executed, memory and register modifications in order to build the phylogenetic tree at runtime. Furthermore, they show that network resources were useful for visualizing short non-equivalent code metamorphism than trees. They consider
execution trace of alternating APIs and user procedures: in this case the difference with respect to our method is represented by the fact that we consider system calls trace, that are not affected by API version.

V. CONCLUSION

In this paper we propose an approach based on model checking able to track mobile malware phylogenetic implemented in the droidSapiens tool. We extract syscall and from the retrieved traces we build the phylogenetic tree identifying the ancestor and the descendant between mobile malware families. As future work we intend to investigate the use of the \(k\)-bisimulation \([11], [24]\) to measure the similarity among malware families. Furthermore, we intend to investigate the multiple ancestors. Comparisons of droidSapiens with other tools will be performed.

ACKNOWLEDGMENTS

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII. The authors thank Rebecca Viglione for her support during the experiment.

REFERENCES