Model checking for malicious family detection and phylogenetic analysis in mobile environment

Mario G.C.A. Cimino[a], Nicoletta De Francesco[a], Francesco Mercaldo[b,c,*], Antonella Santone[c], Gigliola Vaglini[a]

[a] *Department of Information Engineering, University of Pisa, Pisa, Italy*
[b] *Institute for Informatics and Telematics, National Research Council of Italy, Pisa, Italy*
[c] *Department of Biosciences and Territory, University of Molise, Pesche (IS), Italy*

ARTICLE INFO

Keywords:
Malware evolution
Malware phylogeny
Formal methods
Model checking
Process mining
Security
Android

ABSTRACT

Malware targeting mobile devices is widespread, in fact considering the great amount of sensitive and private information stored in tablets and smartphones they represent an interesting surface attack for malware developers. From the defensive side, the well-known weaknesses of the current anti-malware technologies do not permit only the detection of new obfuscated malicious payloads, but also of obfuscated malware (even with trivial obfuscation techniques applied with automatic morphing engines). In fact, a threat is recognized only if its signature is present in the anti-malware repository and typically the signature extraction consists in a time consuming task performed by security analysts. In this paper we propose a two-fold method aimed to (i) detect the belonging family of a mobile malicious application and (ii) collocate the application in the right position in the phylogenetic tree. We represent application system call traces in terms of automaton and, through the adoption of process mining, we extract temporal logic property verified with the adoption of a formal verification environment. The evaluation on a data-set composed by more than 12,000 Android applications (4552 malicious ranging from 2010 to 2018, 4552 obfuscated with three different obfuscation engines and 3500 legitimate) confirms the effectiveness of the proposed formal methods-based approach, obtaining an accuracy ranging from 0.882 to 0.987 in the analysis of 12 real-world widespread malicious families implementing different behaviours.

1. Introduction

Mobile device use is on the rise: mobility firm Ericsson predicts that there will be over six billion smartphone users worldwide by 2020, overtaking landlines (Kaspersky, 2019).

Over the last years, the growth in the number of applications for mobile devices has radically changed the way to communicate and to access our information. In fact, given their increasing capabilities, mobile devices are currently used to access sensitive data, such as personal information and email, and to perform a wide range of activities, like paying a bill and checking a bank account.

But along with increased use comes an explosion of mobile malware i.e., malicious code developed to target smartphones and tablets. Mobile malware, as its name suggests, is a malicious software that specifically targets mobile devices operating systems (Arp et al., 2014). There are many types of mobile malware variants and several methods of distribution and infection (Zhou and Jiang, 2012).

The risk to be infected from mobile malware is more than real: researchers from the University of Cambridge found that 87% of all smartphones are exposed to at least one critical vulnerability, while Zimperium Labs discovered that 95% of mobile devices could be hacked with a simple text message (Kaspersky, 2019).

In fact, in the last years mobile devices have become target of continuous attacks obtained through an increasing number of malicious software that becomes everyday more aggressive and sophisticated (Li et al., 2017). In response to this phenomenon, a wide range of anti-malware software have been developed, mostly signature-based. This approach is accomplished essentially through manual processes: malicious signatures are typically generated through expert knowledge by manual inspection of code and comparison to known signatures. This approach can be trivially evaded by malware codes (the application of obfuscation techniques is enough to alter the signature and, to evade the detection), and it can be also prone to false classification and attribution, depending on the quality of the signatures.

* Corresponding author at: Institute for Informatics and Telematics, National Research Council of Italy, Pisa, Italy.
E-mail address: francesco.mercaldo@iit.cnr.it (F. Mercaldo).

The malicious code is usually generated from existing code (Li et al., 2019; Zhang et al., 2019) with freely available software to generate new malware using, for instance, code permutations from the original code. This phenomenon is boosting a proliferation of variants of the same malware sample, that are logically grouped in "malware families".

Malware writers modify existing code in ways that are typical in software industry and open source landscape (Walenstein and Lakhotia, 2012a): they add features to an existing malware, they can generate multiple configurations of the same malware to allow the execution on several platforms, they combine together components of different malicious software. Other techniques for malicious code automatic generation include: recompiling (Rosenblum et al., 2011), packing (Chen et al., 2008) and obfuscating (Nagra and Collberg, 2009). For these reasons, even when the signature for a malicious payload is generated, it is really trivial for malware writers to generate an obfuscated variant of the malicious behaviour with the aim to elude the current anti-malware mechanism.

In this scenario, it emerges the need of methodologies for malware family detection and for malware phylogeny analysis. Malware family detection is meant to identify whether a sample exhibits a malicious behaviour (Canfora et al., 2018), while malware phylogeny analysis is the study of similarities and differences in program structure with the aim to find relationships within groups of applications (Cimitile et al., 2017), providing insights about new malware variants not available within the databases of malware signatures (Jilcott, 2015). Phylogeny analysis is usually considered as a support for lineage reconstruction (Haq et al., 2018) i.e., the identification of the ancestor-descendant relationships among malware samples, identifying, whether possible, the direct samples from which a specific piece of malware may have been derived (Dumitras and Neamtiu, 2011).

Recognizing the relationships among malware programs is at the basis of a variety of security tasks, from malware characterization to threat detection and cyber-attack prevention (Suarez-Tangil et al., 2014).

Moreover, in malware triage (i.e., the assessment of a security event to determine if there is a security incident and its priority) (Bayer et al., 2009; Hu et al., 2009; Jang et al., 2011), malware phylogenesis can be used by malware analysts to understand trends over time and make informed decisions about the best strategies to dissect the malware samples. Moreover, identifying malware lineage through phylogenetic can help to face more promptly zero day malware programs, when they are or contain evolved versions of known malware.

Starting from the assumption that malicious behaviors are elicited by specific sequences of system calls (Canfora et al., 2015b; Xiao et al., 2019), we propose an approach for malware detection and malware phylogeny based on the generation of a formal model for each application starting from system call traces. Furthermore, from a set of traces belonging to the same malicious family, we adopt process mining to automatically infer the properties related to the family under analysis, assuming that similarities and derivations between system calls can be discovered and modeled in system call traces, similarly to what applies for business process activities in business process logs. According to this, we consider process mining to infer a characterization of the behavior of malicious mobile applications from a set of system call traces gathered from it in response to a set of operating system events. From the process gathered by the process mining, we automatically generate a temporal logic property and thus we exploit a model checker as formal verification environment. With the verification environment, we are able to establish (i) to which family an app belongs and (ii) its antecedent-descendant relationship by placing the samples in the phylogenetic tree.

Even if the proposed approach can be applied to all the existing platforms, in this work the focus is on the Android platform as, according to recent survey (Tecktalk, 2019), Android malware has increased approximately 40% in 2018 with respect to the previous year. Security analysts showed over 3 million malware variants had been identified by the end of the third quarter of 2018 (Tecktalk, 2019). Moreover, current solutions to protect Android users are still inadequate. In fact, in addition to the weaknesses of the signature-based approach, there are new techniques that evade signature-based detection approaches by including various types of source code transformation and simple forms of polymorphic attacks (Rastogi et al., 2014). The Android security is also affected by another problem: differently from anti-malware software on desktop operating systems, Android does not permit to monitor file system operations. An Android application, indeed, can only access its own disk space; as such, an Android anti-malware cannot access and verify the malicious code eventually downloaded and run at run-time by another application installed in the device. This problem has been mitigated, but not solved, by Google with the introduction of Bouncer (Oberheide and Mille, 2012). When a new application is submitted to the Google Play Store, Bouncer executes it in a sandbox for a fixed-time window before making it available to users on the official store. Consequently, Bouncer can detect malware actions that happen in this time interval but cannot detect the other malware actions that happen after this observation period. For this reason, new techniques that go beyond to detect malicious software targeting Android devices are required.

With respect to existing approaches to malware family detection and phylogeny model extraction, the adoption of process mining to automatically infer the temporal logic properties is the main contribution of the paper. The contributions in the current state-of-the-art in this context are mainly related to the adoption of machine learning technique for detecting Android families, while the approaches to build the phylogenetic tree basically consider a signature composed of several instruction of the samples under analysis. From the other side the works exploiting formal methods require the manual generation of the property, for both the malware family detection task and the phylogenetic tree building. Considering that formal methods exhibited robustness to obfuscation techniques, the rationale behind this paper is to propose a method exploiting formal methods with the automatic property generation, in this way the adoption of formal method in this context can be fully automatised. The proposed approach is particularly suitable to be used as an automatic verification step in the approval process performed by application stores to ensure the security of the published applications.

This paper is an extension of our earlier work presented in the 5th International FME Workshop on Formal Methods in Software Engineering (Cimitile et al., 2017). With respect to this work, this paper presents following novelties:

- an integrated approach for both family identification and phylogeny analysis (the work Cimitile et al., 2017 presents a preliminary analysis focused on malware phylogeny);
- the automatic rule inferring through process mining (the properties in the work in (Cimitile et al., 2017) are manually gathered by looking at technical reports);
- an extended set of operating systems events to trigger the malicious payload are considered (in the work Cimitile et al., 2017 only the *BOOT_COMPLETED* event is considered);
- a wider experiment involving a larger set of applications belonging to twelve widespread real-world Android malware families (the work Cimitile et al., 2017 explores evolution relationships of only five Android families, while in this one we experiment twelve families) ranging from 2010 to 2018;

- we generated a morphed version of the malicious samples to demonstrate the resilience of the proposed method against a set of common obfuscation techniques (provided by three different morphing engines, two freely available and the third one developed by the authors) currently widespread by malware writers to evade signature-based detection.

The rest of the paper is organized as follows. Section 2 describes background notions related to formal methods. Section 3 presents the proposed method for malware detection and phylogeny analysis. Section 4 evaluates the effectiveness of the approach by testing it on a data-set 12,604 Android samples (ranging from 2010 to 2018) between legitimate and malware applications. Related work are discussed in Section 5, finally Section 6 provides conclusive remarks.

2. Formal methods preliminaries

In this section we recall preliminary notions on formal methods.

2.1. Calculus of communicating systems

The Calculus of Communicating Systems of Milner (CCS) is one of the most well known process algebras. Readers unfamiliar with CCS are referred to Milner (1989) for further details. The basic idea in the definition of process algebras is the algebraic structure of the concurrent processes. This uses a state-based approach with labeled transitions, where states and transitions correspond to processes and actions, respectively. Briefly, we assume that the system behaviour is represented as a labeled transition system, i.e, an automaton. It basically consists of a set of nodes together with a set of labelled edges between these nodes. A node represents a system state, while a labelled edge represents a transition from one system state to the next. That is, if a the automaton contains an edge $s \xrightarrow{\alpha} s'$, then the system can evolve from state s into state s' by the execution of action α. One state is selected to be the root state, i.e., the initial state of the automaton.

2.2. Model checking and mu-Calculus logic

Considering 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 system state of $\mathcal{T}$ reachable from every state?

Temporal logics are logical formalisms designed for expressing the above properties. In detail, in the model checking framework, systems are modeled 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 (Kozen, 1983) in the usual extended form Stirling (1989) as a branching temporal logic to express behavioural properties. The syntax of the extended mu-calculus (from now on, mu-calculus for short) is described in the following equation, where K ranges over sets of actions (i.e., $K \subseteq \mathcal{A}$) and Z ranges over variables:

$$\phi ::= \texttt{tt} \mid \texttt{ff} \mid Z \mid \phi \wedge \phi \mid \phi \vee \phi \mid [K]\phi \mid \langle K\rangle\phi \mid \nu Z.\phi \mid \mu Z.\phi \quad (1)$$

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.

Table 1
Satisfaction of a closed formula by a state.

$p \not\models \texttt{ff}$		
$p \models \texttt{tt}$		
$p \models \varphi \wedge \psi$	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:		
$\nu Z^0.\varphi = \texttt{tt}$		$\mu Z^0.\varphi = \texttt{ff}$
$\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z]$		$\mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]$
where the notation $\varphi[\psi/Z]$ indicates the substitution of ψ for every free occurrence of the variable Z in φ.		

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 $\texttt{tt}$ and no state satisfies $\texttt{ff}$; 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 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 no free variables. $\mu Z.\phi$ is the least fix-point 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 $\mathcal{T}$.

In the sequel we will use the following abbreviations:

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

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

In this work we resort to the Concurrency Workbench of New Century (CWB-NC) (Cleaveland and Sims, 1996) formal verification environment, which supports several different specification languages, among which CCS. In the CWB-NC the verification of temporal logic formulae is based on model checking (Clarke et al., 2001). 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.

2.3. Motivation of the use of the CCS and of the mu-calculus logic

The adoption of CCS is a viable solution, due to the availability of efficient formal verification methodologies. Moreover, being a

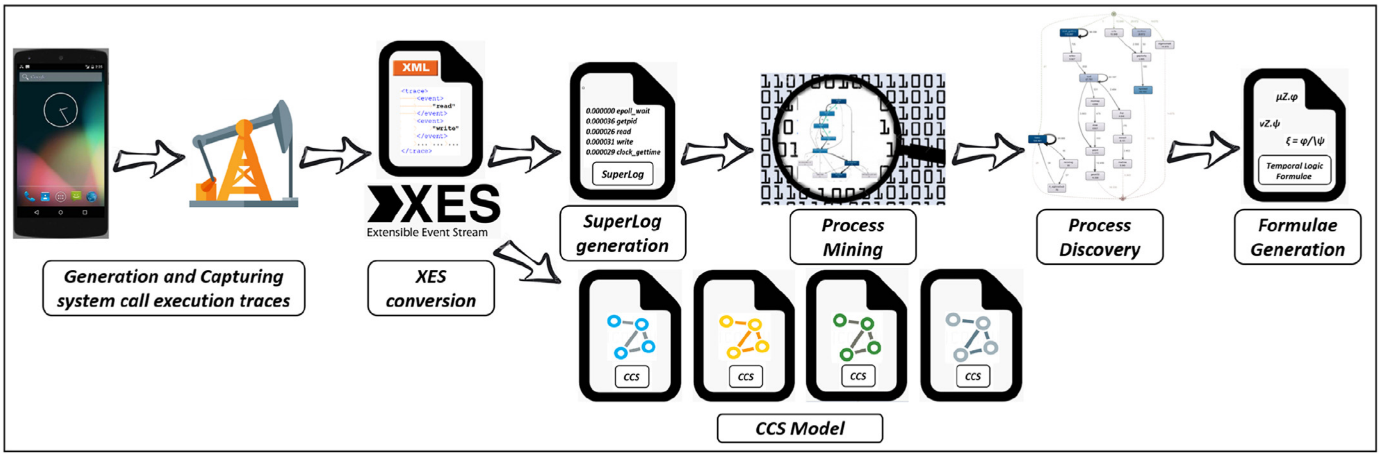

Fig. 1. Formal model and temporal logic formulae generation.

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 for example the pattern-matching approach, but also checking a branching temporal logic formula, that permits to define a wide range of properties of 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.

3. The method

In this section the proposed approach is described.

As mentioned in Section 1, the proposed method is aimed to (i) detect malicious samples belonging to the same malicious family by automatically generating a formula and (ii) track the philogenetic tree.

The main contribution of the proposed work is the automatic generation of the formula characterizing a family malicious behaviour. To do this formal methods and process mining are used. However, we have to analyze applications that we know to be certainly belonging to specific malware families. As stated in the experimental evaluation section, this is possible since the research community has released in last years several collections of malicious samples with the label related to the belonging family. Thankful to this effort, in the experimental evaluation, a real-world malicious data-set of Android applications is considered.

The overall approach is shown in Figs. 1 and 3. In detail, in Fig. 1 the formal model and temporal logic formulae generation step is shown.

The formal model and temporal logic formula generation is composed by following steps:

- Generation and Capturing system call execution traces;
- XES conversion;
- Superlog generation;
- Process Mining;
- Process Discovery;
- Formulae generation;
- CCS model.

3.1. Generation and capturing 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 application archive (i.e., the APK) is installed and started on an Android device emulator. Successively, a set of 25 different operating system events (Jiang and Zhou, 2013; Zhou and Jiang, 2012) is generated (at regular time intervals) and sent to the emulator and the correspondent sequence of system calls is gathered. Table 2 shows the considered events.

Looking at the table, the first row represents the BOOT event, the most used within existing Android malware. This is not surprising since this event will be triggered and sent to all the applications installed in an Android device as the system finishes its booting process, a perfect time for a malware to kick off its background malicious services (Mercaldo et al., 2016a). By listening to this event, a malware can start itself without any intervention or interaction of the user with the system.

Other events frequently used by malware writers are the ACTION_ANSWER and NEW_OUTGOING_CALL events (second and third row in Table 2): these events will be sent in broadcast to the whole system (and all the running applications) when a call is received or started.

The system call collection from the application under analysis is handled by a set of shell scripts developed by authors able to perform the following actions:

- start the target Android device emulator;
- install the APK of the application on the device emulator;
- wait 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);
- start the capture of syscall traces;
- send one event from the activation system events in Table 2;
- send the selected event to the application;
- capture syscalls made by the application until a stable state is reached;
- select a new system event and repeat the steps above to capture syscall traces for this event;

Table 2
System events considered for the malicious payload activation.

#	System Event	Description
1	*BOOT_COMPLETED*	Able to catch the boot completed
2	*ACTION_ANSWER*	Incoming call
3	*NEW_OUTGOING_CALL*	Outgoing call
4	*ACTION_POWER_CONNECTED*	Battery status in charging
5	*ACTION_POWER_DISCONNECTED*	Battery status discharging
6	*BATTERY_OKAY*	Battery full charged
7	*BATTERY_LOW*	Battery status at 50%
8	*BATTERY_EMPTY*	Battery status at 0%
9	*SMS_RECEIVED*	Reception of SMS
10	*AIRPLANE_MODE*	The user has switched the phone into or out of Airplane Mode
11	*BATTERY_CHANGED*	Battery status changed
12	*CONFIGURATION_CHANGED*	The current device Configuration (orientation, locale, etc) has changed
13	*DATA_SMS_RECEIVED*	A new data based SMS message has been received by the device
14	*DATE_CHANGED*	Receives data changed events
15	*DEVICE_STORAGE_LOW*	Free storage on device is less than 10% of total space
16	*DEVICE_STORAGE_OK*	Free storage on device is adequate
17	*INPUT_METHOD_CHANGED*	An input method has been changed
18	*PROVIDER_CHANGED*	Providers publish new events or items that the user may be especially interested in
19	*PROXY_CHANGE*	Variation of proxy configuration
20	*SCAN_RESULTS*	An access point scan has completed, and results are available from the supplicant
21	*SENDTO*	Send a message to someone specified by the data
22	*SIM_FULL*	The SIM storage for SMS messages is full
23	*SMS_SERVICE*	CDMA SMS has been received containing Service Category Program Data
24	*STATE_CHANGED*	The state of Bluetooth adapter has been changed.
25	*WAP_PUSH_RECEIVED*	A new WAP PUSH message has been received by the device

- repeat the step above until all 25 system events in Table 2 have been considered (i.e., the application under analysis was stimulated with all the system events shown in Table 2);
- stop the syscall capture and save the captured syscall trace;
- stop the Android device and revert its disk to a clean baseline snapshot.

For each application we collect 10 execution traces by executing 10 times each mobile application with the developed scripts. We collected 10 execution traces for each application, in order to mitigate the occurrence of rare conditions and to have more chances to extract the malicious payload and discern the system call related to a legitimate code.

3.2. XES Conversion

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[1]). 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, are filtered out.

3.3. Superlog generation

The *Superlog* is obtained by combining in a single log file several execution traces belonging to the same malware family: for each family a Superlog is obtained. This is possible since we consider a data-set of malicious apps, each one labelled with the name of its family. The underlying idea is that different malicious samples belonging to the same family exhibit the same malicious behaviour: in this context the process mining can be useful to extract the common malicious behaviour exhibited by different samples belonging to the same family. For this reason, we create one Superlog for family. The Superlog represents the basis for using Process Mining to perform process discovery. Through the Superlog from the process discovery it is expected to obtain a dependency graph between system calls representative of the several traces included in the Superlog.

3.4. Process mining

Process mining represents a series of techniques in the process management field supporting the analysis of business processes based on event logs. During process mining data mining algorithms are applied to event log data to identify trends, patterns and details contained in event logs recorded by an information system. Process mining aims to improve process efficiency and understanding of processes. In detail we consider process mining for automatically extract a common behavior (represents in terms of dependency graph) between several traces belonging to the same family (i.e., the Superlog). The rationale behind this idea is that samples belonging to the same family exhibits the (or a variant of) same malicious behaviour that, regardless to the obfuscation techniques considered by malware writers and the code-level implementation, at system call level must be exhibited by the malware samples.

3.5. Process discovery

The output of the *Process Mining* step is a process extracted from the *Superlog*. There are different algorithms for process discovery, in this work we use a heuristic algorithm based on the Fuzzy Miner (Günther and Van Der Aalst, 2007) designed by Gunther and Van der Aalst in 2007. The Fuzzy Miner is the first Process Mining algorithm introducing the concept of *map* to show the dependencies between several activities using two parameters: *activities* and *paths*. The activities parameter is used to select the most frequent percentage of the activities (i.e., the system calls in the considered context) belonging to the Superlog. The other parameter concerns the percentage of connections (and therefore of dependencies) between the activities themselves. It is important to tune these two parameter to obtain processes easy to understand but also representative of the malware family.

[1] http://www.xes-standard.org/.

3.6. Formulae generation

Once obtained the process representing the malicious behaviour (we recall that the process is extracted from the Superlog i.e., a log containing several traces belonging to the same malware family), the temporal logic formula characterizing the family behaviour is automatically inferred.

3.7. CCS Model

From the XES 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 $\mathcal{T}$ which transforms system call execution traces into a CCS model. The reader can refer to Santone and Vaglini (2016) for more details about the function $\mathcal{T}$. Roughly speaking, the function $\mathcal{T}$ 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.

Below we show an example of model obtained from the following traces:

- *Execution trace #1: read, ioctl, getpid, write*;
- *Execution trace #2: read, ioctl, getpid, close*;
- *Execution trace #3: epoll_wait, rcvfrom*.

The obtained automaton is shown in Fig. 2.

Considering that the #1 and #2 traces share the *read, ioctl* and *getpid* path, this path is represented one time in the Fig. 2 automaton.

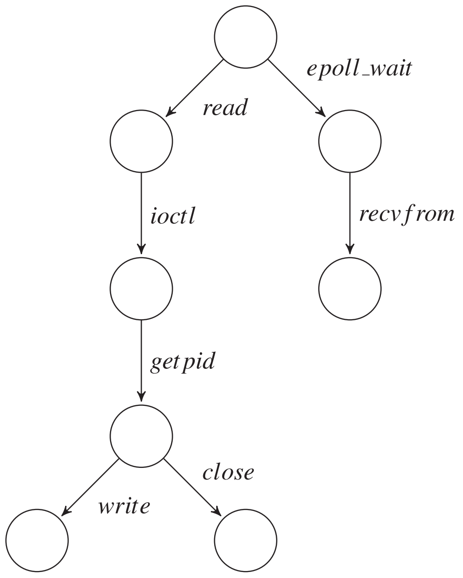

Fig. 2. An example of automaton modeling of the three example traces.

3.8. Formal model verification

Once generated the formal models and the temporal logic formulae, the formal model verification is shown in Fig. 3.

The formal model verification is composed by following steps:

- Model Checker;
- Analysis.

Model Checker. Using as input the *Formulae* and the *Models* we invoke the Concurrency Workbench of New Century (CWB-NC) (Cleaveland and Sims, 1996) as formal verification environment. Whether the formula it verified on the model, the model checker will output *true*, otherwise false.

Analysis The aim of this step is (i) to identify the belonging family and (ii) to place the sample in the right antecedent descending collocation in the phylogenetic tree. We recall that from each Superlog a process is gathered and from this process a temporal logic formula is generated: each formula is able to catch a different malicious behaviour.

With regard to the *Family Identification* task, we verify the φ formula characterizing the specific family F individually: the malicious behavior shown by the sample will be the one described by the formula that will be verified on the model.

As shown in the next section, each φ_F formula characterizing the specific family F is composed by several subformulae φ_i with $1 \leq i \leq n$ where n is the number of the paths related to the specified malicious behaviour. More precisely, $\varphi_F = \varphi_1 \wedge \ldots \wedge \varphi_n$. With regard to the *Philogeny tracking*, we consider one subformula φ_i, with $1 \leq i \leq n$, that mostly characterize the specific family F on each app of all the families of a fixed data-set. In the following, $Sel(\varphi_F)$ denotes the set containing the above subformula. In details, to the $Sel(\varphi_F)$ set belong the most representative formula of the malicious behaviour of the family, i.e., those codifying the greater

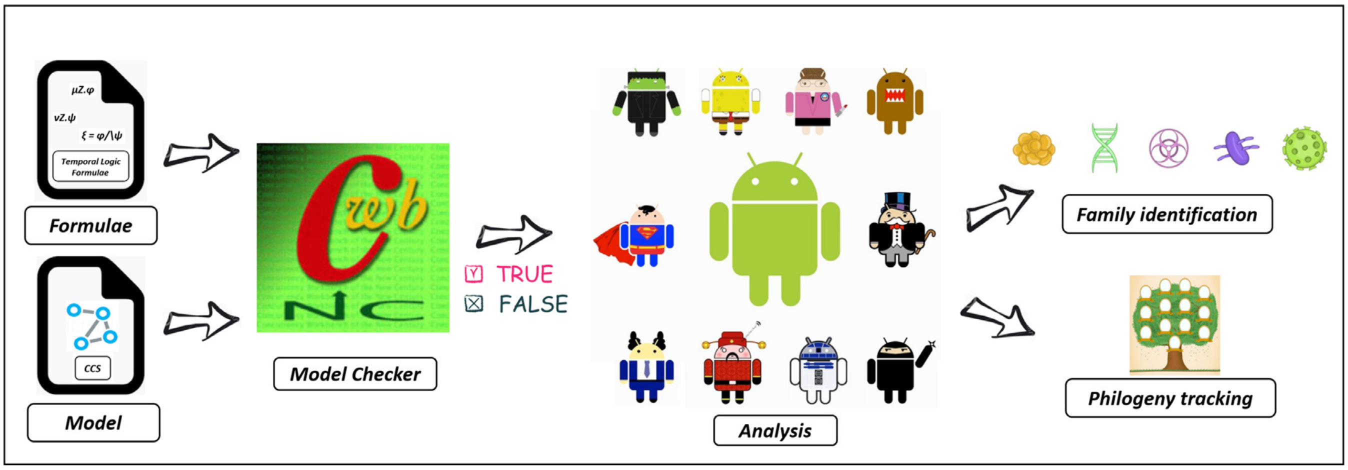

Fig. 3. Formal model verification.

system calls number (in fact, a temporal logic property including a greater number of system calls is more characterizing of the malicious behaviour than the temporal logic property with a less system call number). Given a malware family *F*, we report the percentage of the apps that satisfy the formulae in $Sel(\varphi_F)$. From this analysis we consider the family *X* as "related" with the family *Y* if the in $Sel(\varphi_X)$, characterizing the family *X*, is true on more than the 50% of the apps belonging to *Y*. This threshold is computed on the basis of experimental results.

4. The experimental analysis

In this section we respectively describe the data-set involved in the experiment, we provide an example of temporal logic formula generated from the related discovered process and, finally, we present the obtained results for the malware family detection and the philogeny tracking task.

The aim of the experimental analysis is to demonstrate that process mining technique can be adopted to extract significant relationships between Android samples belonging to the same malicious family. In this way, a set of logical temporal properties built from the gathered process can be considered to detect the belonging family of unseen samples. Moreover we consider the most characterizing temporal property gathered to establish the antecedent-descendant relationship of the family in the phylogenetic tree. The machine used to run the experiments was an Intel Core i7 8th gen, equipped with 2GPU and 16Gb of RAM.

4.1. Data-set

The data-set examined in the experiment was gathered from several repositories: with regard to the malicious samples we obtained mobile malware from the Drebin data-set (Arp et al., 2014; Spreitzenbarth et al., 2013), a very well known collection of malware widely used by the scientific community, which includes the most widespread Android families, Contagio Mobile[2] and by crawling the VirusTotal[3] API.

The considered malware data-set consists of families characterized by several installation methods: (i) *standalone*, apps that intentionally include malicious functionalities; (ii) *repackaging*, known and common (legitimate) apps that are first disassembled, then the malicious payload is added, and finally are re-assembled and distributed as a new version (of the original app); and (iii) *update attack*, apps that initially do not show harmful behaviors and download an update containing the malicious payload, at runtime.

The malware data-set is also partitioned according to the *malware family*; each family contains malicious samples sharing several characteristics: the payload installation, the kind of attack and the events triggering the malicious payload (Zhou and Jiang, 2012). Table 3 shows the 12 malware families involved in the experiment with the details of the installation types, the kinds of attack, the events which activate the payload, the discovery date and the number of malicious samples belonging to each family.

To obtain legitimate applications, we crawled the official app store of Google[4], by using an open-source crawler[5]. The obtained collection includes samples belonging to all the different categories available on the market. The legitimate applications were collected between January 2016 and April 2016.

We analyzed the data-set with the VirusTotal service[6], a service running 61 different free and commercial anti-malware software (i.e., ESET NOD32, Kaspersky Lab, Norton Antivirus, F-Prot, Avast!, and others): the analysis confirmed that the legitimate applications did not contain malicious payload while the malware ones were actually recognized as malicious.

Moreover, to demonstrate the effectiveness of our method in malware detection, we applied a set of well-known code transformations techniques (Canfora et al., 2015a; Rastogi et al., 2013; Zheng et al., 2012) to the malicious applications. These techniques are used by malware writers to evade the signature-based detection approaches adopted by current anti-malware (Rastogi et al., 2014; You and Yim, 2010).

Clearly, the process mining task is performed on original samples. Subsequently, through the formal verification environment, we evaluate whether the properties (automatically obtained from the Superlog related to the families) is able to detect also obfuscated variant.

In particular, we applied following transformation techniques:

1. **Disassembling & 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 represented in another manner. In this way, signatures relying on the order of different items in the *.dex* file are likely to be ineffective with this transformation.
2. **Repacking.** Every Android application has a developer signature key that will be lost after disassembling and reassembling the application. Using the *signapk*[7] tool, it is possible to embed a new default signature key in the reassembled application in order to avoid detection signatures that match the developer keys.
3. **Changing package name.** Each application is identified by a unique package name. This transformation renames the application package name in both the Android Manifest file and all the application classes.
4. **Identifier renaming.** This transformation renames each package name and class name by using a random string generator, in both the Android Manifest file 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 is restored during application execution with a call to a *smali* method that knows the *Caesar key*.
6. **Call indirections.** This transformation mutates the original call graph of the application by modifying every method invocation in the code with a call to a new method which simply invokes the original method.
7. **Code Reordering.** This transformation is aimed at modifying the instructions order in the application methods. A random reordering of instructions has been accomplished by inserting *goto* instructions with the aim of preserving the original runtime execution trace.
8. **Defunct Methods.** This transformation adds new methods that perform defunct functions, clearly the logic of the original source code remains unchanged.
9. **Junk Code Insertion.** These transformations introduce code sequences that have no effect on the function of the code. Detection algorithms relying on instructions sequences may be defeated by this transformation. This transformations provides insertion of *nop* instructions into each method, unconditional jumps into each method, and allocation of three additional registers performing garbage operations.
10. **Encrypting Payloads and Native Exploits.** In Android, native code is usually made available as libraries accessed via Java

[2] http://contagiominidump.blogspot.com/.
[3] https://www.virustotal.com.
[4] https://play.google.com/store.
[5] https://github.com/liato/android-market-api-py.
[6] https://www.virustotal.com/.

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

Table 3
The malware families.

Family	Description	Inst.	Events	Date	#
Geinimi	it has the potential to receive commands from a remote server that allows the owner of that server to control the phone	r	MAIN	12–2010	77
Plankton	advance the update attack by stealthily upgrading certain components in the host apps, it does not require user approval.	u	MAIN	06–2011	479
BaseBridge	it sends information to a remote server running one or more malicious services in background	r,u	BOOT, SMS, NET, BATT	09–2011	314
Kmin	it is similar to BaseBridge, but does not kill anti-malware processes	s	BOOT	10–2011	87
GinMaster	it contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and install applications	r	BOOT	06–2012	328
Opfake	it demands payment for the application content through premium text messages	r	MAIN	2013	597
FakeInstaller	SMS trojan adding server-side polymorphism, obfuscation, antireversing techniques and frequent recompilation	r	BOOT, SMS	2014	831
HummingBad	it shows ads and installs other apps, for the monetary gain of their creators	s	MAIN	2016	590
RedDrop	malware with the ability to record audio files from the infected devices and sends them to attackers	s	MAIN	2018	266
Overlay	it overlays the targeted application with a phishing screen to steal bank credentials	s	MAIN	2017	75
Judy	auto-clicking adware, generating large amounts of fraudulent clicks on advertisements, generating revenues for the perpetrators behind it	s	MAIN	2017	85
Xbot	it intercepts and parse SMS messages from banks. Moreover it is able to perform an "activity hijacking"	s	MAIN	2016	36

Native Interface (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. These are easily implemented and have been observed in the wild (e.g., DroidKungFu malware uses encrypted exploit Zhou and Jiang, 2012).

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 also be used for call graph obfuscation.
12. **Reflection.** This transformation converts any method call into a call to that method via reflection. This makes it difficult to statically analyze which method is being called. A subsequent encryption of the method name can make it impossible for any static analysis to recover the call.

We apply the full transformation set to the malicious samples with the Droidchameleon (Rastogi et al., 2013), the ADAM (Zheng et al., 2012) and the Carnival[8] tools. Table 4 shows the obfuscation techniques implemented by the three tools.

We combined together all the transformations provided by the three morphing engines: the transformations are applied in sequence to generate from a malicious sample its obfuscated version. Moreover, the transformations are applied to each class of the application, in this way all the classes of the application (including the ones implementing the malicious payload) are afflicted by the morphing techniques.

With the evaluation of obfuscated malicious samples, we demonstrate that the properties automatically generated from each families, differently from the signature provided by current anti-malware, are able also to detect the obfuscated versions of the sample of analysed families. In fact, new (or obfuscated) versions

Table 4
The transformation techniques provided by considered obfuscators.

Transformation	Carinival	DroidChamelon	ADAM
Dissassembling	X	X	X
Repacking	X	X	X
Changing package name	X	X	
Identifier renaming	X	X	
Data Encoding	X	X	
Call indirections	X	X	
Code Reordering	X	X	
Defunct Methods			X
Junk Code Insertion	X	X	
Encrypting Payloads		X	
Function Outlining		X	
Reflection		X	

of malicious code appear that are not recognized by signature-based anti-malware (Rastogi et al., 2013). This is the reason why we consider also obfuscated malware, to demonstrate that the proposed method is able to detect morphed samples.

To summarize, the analysed data-set is composed by 12,604 samples: 3765 malicious samples belonging to the 12 families shown in Table 3, 3500 crawled by the Google Play and 3765 morphed samples generated by applying the three obfuscation engines on the malicious data-set. Moreover, we considered other 787 malicious samples (and their related obfuscated version) to verify that no errors of misclassification occur, in fact these samples are belonging to other families (i.e., *Adrd, DroidDream* and *DroidKungFu*).

4.2. Temporal logic formulae discovering

Generally, to apply process mining the following three requirements are needed:

Case ID: a case identifier, also called process instance, is necessary to distinguish different executions of the same process. In the considered context, the process instance is represented by the execution trace, while the process is the application: clearly from one application we can gather several system call execution traces: in this work we obtain 10 different traces for each application.

Activity: in this context the activity is represented by the system call name.

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

Table 5
Maps readability with different activity and path settings with 20 execution traces/2 apps for family.

#	Activity (%)	Path (%)	Comment
1	30	0	The maps contain a very high number of activities, therefore it is not possible to distinguish the processes discovered and the relative dependencies between system calls. Even if the paths parameter is changed, the map is illegible, for this reason it is necessary to reduce the activity parameter
2	0	0	Some maps are reduced to few activities, while others maintain a higher number. In particular, for some families the map does not change at all with respect to the configuration with 30 and 0. Very simple maps are generated, while others are extremely complicated. For the simpler ones we note that there are some similar patterns between them, this could certainly be a threat to distinguishing.
3	10	0	With this pair of parameters the maps generated are still very complex for different families, but they can be distinguished one from another. Some maps remain identical to the 0 and 0 configuration, while others are unchanged compared to the 30 and 0. Therefore, it is not possible to simplify all the maps sufficiently.
4	20	0	With this configuration, the families with a complex map maintain it unchanged with respect to the 10 and 0 configuration. The map of the others becomes more complex with respect to the previous configurations.

Timestamp: each system call is accompanied with the timestamp related to its generation, this is important to extract useful temporal information related to the system call sequences.

To discovery a process related to a single malware family we firstly consider 20 execution traces belonging to the same family.

We consider for the process discovery the Disco software (Günther and Rozinat, 2012), able to manage large event logs and complex models. This software implements the Fuzzy Miner algorithm (Günther and Van Der Aalst, 2007). Moreover it includes through its graphical interface the highlighting of recurrent activities and paths.

The process discovery algorithm provided by Disco represents the first Process Mining algorithm introducing the concept of “map metaphor” to show the dependencies between the several activities. Moreover it allows to simplify the discovered process by using two parameters: *activities* and *paths*. The first parameter, the activities one, serves to select the percentage of the most frequent activities (in this case the system calls) within the log (to avoid to display activities not really recurrent limiting the map readability). The second parameter i.e., the paths, concerns the percentage of connections (and therefore of the dependencies) between the activities themselves. This parameter changes the map depending on the parameter of the activities, so as not to leave any activity disconnected from the rest of the map.

The several Superlogs contain a few million events and the maps for each family, without an adequate tuning of the activity and path parameters, resulted incomprehensible and unnecessarily.

For this reason we are looking for a trade-off solution between comprehensibility and distinctiveness of the generated maps. In particular, we are looking, for a map that is as simple as possible and that in any case maintains the family distinctive characteristics. To choose the best activity and path setting, we tried several configurations. We report this analysis in Table 5.

Considering that we are not able to obtain a compromise between comprehensibility and distinctiveness of the generated maps, and considering that the Disco algorithm shows the activities basing to their frequency, we considered to add more system calls traces in each Superlog. It can be possible that there is not a preponderant difference between the activities in terms of frequency, therefore the software is forced to show them almost in their entirety, not knowing how to perform the choice in another way. It would be necessary that this difference existed, because it is expected that the system calls representing the malicious behaviour have been run several times, if compared with the system call invocations of the legitimate behaviour. For this reason we considered a number of execution traces in each Superlog ranging from 70 to 100 (depending from the syscall trace length), Table 6 reports the obtained results with 70–100 traces (i.e., 7–10 app) for each family.

From this analysis, it appears that the optimal configuration is the one with 20% of the activities and 0% of the paths: we confirm that the generated processes exhibit unique and distinguishable behaviours.

Table 7 shows the extracted processes for each superlog with the total number of flows extracted for family (**% flows** column), the occurrence percentage in the traces considered in the superlog (**% family** column) and the occurrence percentage in the traces belonging to other family superlog (**% other**).

For instance, in the Plankton family, there is a particular flow: the one connecting the *epoll_wait* system calls with the *recvfrom* one. The first system call, as previously mentioned, is the way in which the application succeeds in putting itself in the expectation of an event. The second one is invoked when the application is waiting to receive a message coming from a socket. According with the behaviour described in Table 3, it is possible that the malware, triggered by one or more of the events that are launched from the emulator, is waiting for a message containing a command coming from a socket connected to the attacker server. It is possible to observe that in no other family there are direct edges connecting these two system calls as in the case of the Plankton family. These two system calls are also parts of the discovered process of other families, but there are always system calls placed in the middle of the path from *epoll_wait* to *recvfrom*. For this reason we state that the Plankton family is distinguishable from the others. Also the Geinimi family uses the *recvfrom* system call, probably for the same purpose of the Plankton family, but is considered into a different sequence. The use of this syscall is justified also in this case by the description in the Table 3. In the BaseBridge family instead, we note the uniqueness of the *epoll_wait, read, sigprocmask, writev* and *ioctl* sequence. The peculiarity lies in the use of the *sigprocmask* function: this function can be used to change the signal mask of a process. The signal mask is the set of signals whose delivery is currently blocked for the caller. It could therefore be used to modify the signal mask of an anti-malware process (once obtained root privileges), to ensure that it responds to a certain signal and terminate it (behaviour described in Table 3, or to no longer respond to certain signals. In the FakeInstaller family we note the use of the *mprotect* function (able to change the memory access protections), rarely used in other discovered processes. Being a family of polymorphic malware, it may be that it comes used this function to change the access rights in memory of a process, which then it can modify his code for example by inserting useless operations, in order to evade anti-malware.

To understand how the temporal logic formulae are generated from the discovered processes, in Fig. 4 the process discovered by the Kmin family superlog is shown.

In Fig. 4, the nodes are labelled with the system calls and the average times they are invoked, while the thickness and number on the edges represent the average frequency in the specific

Table 6
Maps readability with different activity and path setting with 70–100 (i.e., 7–10 app) execution traces for family.

#	Activity (%)	Path (%)	Comment
1	30	0	The complexity is reduced if compared to the same configuration with 20 traces in each superlog, moreover the maps are very distinguishable.
2	0	0	Increasing the paths parameter makes the maps more complex, making them again unreadable. In any case, the maps, although not legible, are not complex as in the previous configuration with 20 traces for each superlog.
3	10	0	With this configuration, 8 families out of 10 maintain the map with respect to the 0 and 0 configuration. The remaining two families contain patterns that are not sufficiently distinguishable.
4	20	0	Most of the maps remain unchanged with respect to configuration 10 and 0. The two that turned out to be undetectable instead show new patterns that make them distinguishable. This configuration is therefore the best compromise between comprehensibility and distinctiveness.

Table 7
Examples of extracted system call sequences for each family.

Family	System call process	flows	% family	% other
Geinimi	*getuid32, epoll_wait, rcvfrom, mmap2, close, getpid + sigprocmask + cacheflush*	9	0.990	0.059
Plankton	*mmap2, munmap + read, close, writev + getpid + sigprocmask*	8	0.961	0.187
BaseBridge	*write, getpriority, getpid, getuid32, epoll_wait, read, sigprocmask, writev, ioctl + write, getpriority, getpid, getuid32, epoll_wait, read, mprotect*	4	0.752	0
Kmin	*sigprocmask, madvise, mprotect + ioctl, madvise, mprotect + writev+ write, getpriority + write, getpid, getuid32, epoll_wait*	5	0.999	0.081
GinMaster	*munmap, madvisegetpriority*	6	0.989	0.018
Opfake	*close, getpid, getuid32, epoll_wait, read, recvfrom, mmap2 + madvise + close, clock_gettime + close, getpid, ioctl, dup*	6	0.861	0.023
FakeInstaller	*clock_gettime, writev, ioctl, dup, mmap2, madvise + sigprocmask*	7	0.984	0.164
HummingBad	*close, getpriority, mprotect, dup, write, cacheflush*	6	0.952	0.193
RedDrop	*getuid32, write, mmap2, ioctl, read, epoll_wait*	4	0.843	0.024
Overlay	*read, getpid, dup, sigprocmask*	6	0.972	0.156
Judy	*getpriority, getuid32, epoll_wait, read, sigprocmask*	4	0.863	0.035
Xbot	*write, clock_gettime, getpid, mprotect, dup, sigprocmask getuid32*	5	0.872	0.039

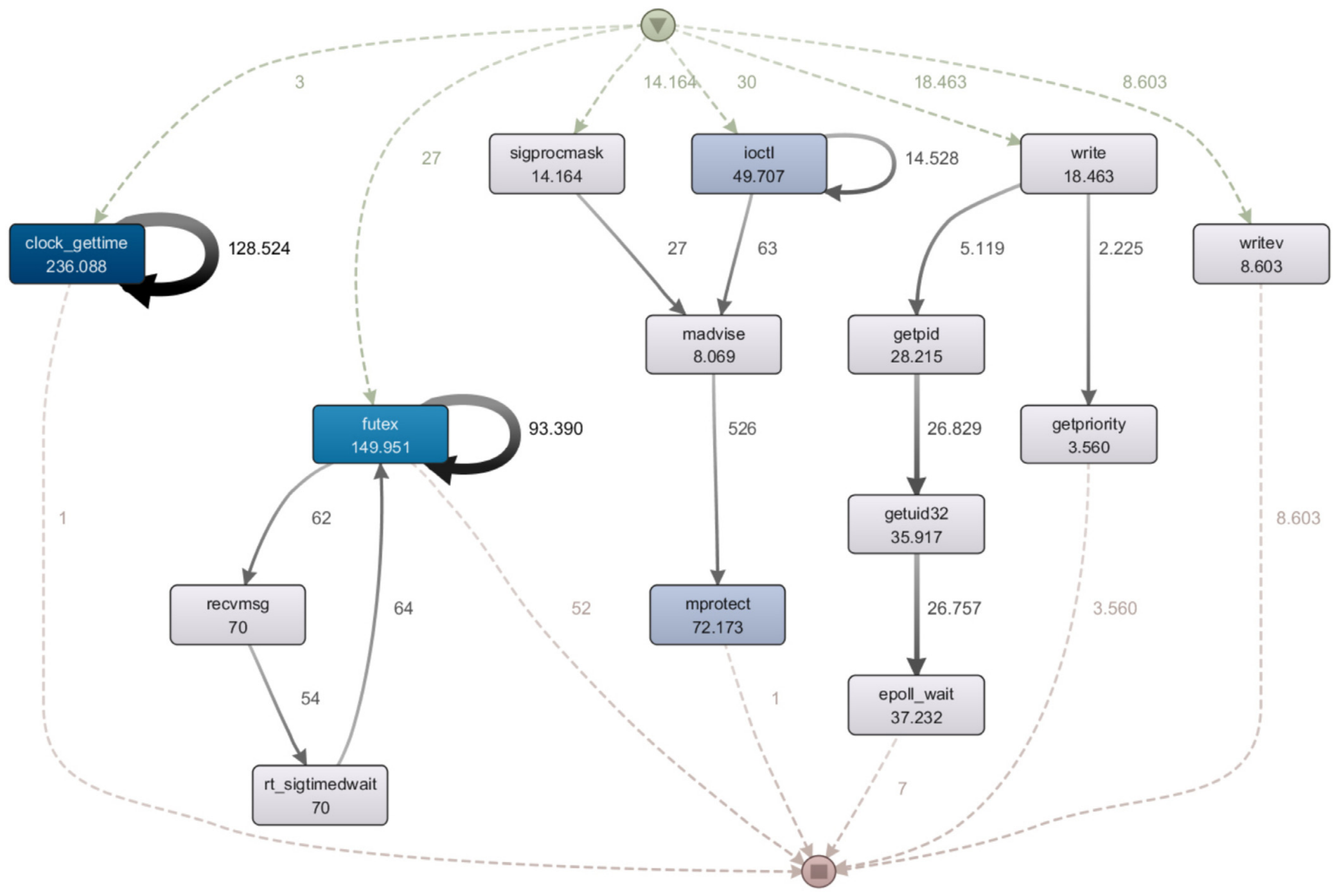

Fig. 4. The process discovered from the analysis of the *Kmin* family superlog.

Table 8
Temporal logic property for the Kmin malicious behaviour detection.

φ_1	$=$	$\mu X.\langle clock\_gettime\rangle\ \texttt{tt}\ \langle -clock\_gettime\rangle X$
φ_2	$=$	$\mu X.\langle futex\rangle\ \varphi_{21} \vee \langle -futex\rangle X$
φ_{21}	$=$	$\mu X.\langle recvmsg\rangle\ \varphi_{22} \vee \langle -recvmsg\rangle X$
φ_{22}	$=$	$\mu X.\langle rt\_sigtimedwait\rangle\ \texttt{tt} \vee \langle -rt\_sigtimedwait\rangle X$
φ_3	$=$	$\varphi_{3a} \wedge \varphi_{3b}$
φ_{3a}	$=$	$\mu X.\langle sigprocmask\rangle\ \varphi_{32} \vee \langle -sigprocmask\rangle X$
φ_{3b}	$=$	$\mu X.\langle ioctl\rangle\ \varphi_{32} \vee \langle -ioctl\rangle X$
φ_{32}	$=$	$\mu X.\langle madvise\rangle\ \varphi_{33} \vee \langle -madvise\rangle X$
φ_{33}	$=$	$\mu X.\langle mprotect\rangle\ \texttt{tt} \vee \langle -mprotect\rangle X$
φ_4	$=$	$\mu X.\langle write\rangle\ \varphi_{41} \vee \langle -write\rangle X$
φ_{41}	$=$	$\varphi_{4a} \wedge \varphi_{4b}$
φ_{4a}	$=$	$\mu X.\langle getpid\rangle\ \varphi_{4a1} \vee \langle -getpid\rangle X$
φ_{4a1}	$=$	$\mu X.\langle getuid32\rangle\ \varphi_{4a2} \vee \langle -getuid32\rangle X$
φ_{4a2}	$=$	$\mu X.\langle epoll\_wait\rangle\ \texttt{tt} \vee \langle -epoll\_wait\rangle X$
φ_{4b}	$=$	$\mu X.\langle getpriority\rangle\ \texttt{tt} \vee \langle -getpriority\rangle X$
φ_5	$=$	$\mu X.\langle writev\rangle\ \texttt{tt} \vee \langle -writev\rangle X$
φ	$=$	$\varphi_1 \wedge \varphi_2 \wedge \varphi_{3b} \wedge \varphi_4 \wedge \varphi_5$

path. The larger this number, the more thick the edge that connects them. The dotted paths are used only to connect the several paths discovered from the superlog. The graph representation of the discovered process is automatically generated from the Disco software. As shown by Fig. 4 there are five different paths distinctive of the family superlog.

Table 8 shows the temporal logic property aimed to identify the Kmin malicious payload generated from the process in Fig. 4.

Each path of the discovered process in Fig. 4 is represented as a single temporal logic formula (i.e., φ_1, φ_2, φ_3, φ_4 and φ_5). The formula is verified whether all these paths are simultaneously verified on the model under analysis (i.e., φ).

We are aware that some constraints of the discovered process are codified in simplified form using temporal logic formulae, considering a relaxed version of the constraint. This is a design choice guided for performance benefits and, as demonstrated by the experimental results, the temporal logic formulae even if representing relaxed constraints are able to discriminate between malicious payloads belonging to different families.

4.3. Baseline

To provide a baseline, we evaluate two different Android malware detection methods (Canfora et al., 2013; 2015c; Mercaldo et al., 2016b), both of them relying on static analysis and related to the malware detection task (i.e., the discrimination between legitimate and malicious application) exploiting supervised machine learning techniques. We refer to the first method is *permission* based, while the second one is *op-codes* based.

The first method considers two features, the first one is a weighted sum of a subset of permissions that the application required and a set of combinations of permissions. The first metric computes a weighted sum of all the permissions requested by the application under analysis, where the weight is assigned according to the potential threat that the permission could cause if the application has evil goals. For instance, “RECEIVE_BOOT_COMPLETED” permission could be more dangerous than “ACCESS_WIFI_STATE” permission.

The second metric is a weighted sum of selected combinations of permissions. The underlying idea is that specific combinations of permissions can be more effective to detect malware applications rather than a weighted sum of all the permissions. Relevant permission combinations were obtained from a literature analysis about smartphone malware. For instance, the $(RECEIVE\_SMS \wedge SEND\_SMS) \vee CALL\_PHONE$ combination obtains a higher risk level if compared to the $ACCESS\_NETWORK\_STATE \vee ACCESS\_WIFI\_STATE$ one. Once computed the metric values for each application involved in the experiment, we build a machine learning model, exploiting the RandomForest classification algorithm (the one reaching the best performances in (Canfora et al., 2013)).

In the second method, opcodes occurrence are considered as features. In a nutshell, six features obtained by counting some Dalvik op-codes of the instructions which form the smali code of the application. The occurrences of the following opcodes are considered as feature vector:

- Move: which moves the content of one register into another one.
- Jump: which deviates the control flow to a new instruction based on the value in a specific register.
- Packed-Switch: which represents a switch statement. The instruction uses an index table.
- Sparse-Switch: which implements a switch statement with sparse case table, the difference with the previous switch is that it uses a lookup table.
- Invoke: which is used to invoke a method, it may accept one or more parameters.
- If: which is a Jump conditioned by the verification of a truth predicate.

From the experimental results in (Canfora et al., 2015c; Mercaldo et al., 2016b) emerges that the model obtaining the best performances is the one built with the J48 classification algorithm, for this reason the same algorithm is considered to evaluate the performance on the second method on the considered experimental dataset.

Table 9
Evaluation results for family identification task.

Family	Precision	Recall	F-Measure	Accuracy
Geinimi	0.994	0.941	0.966	0.987
Plankton	0.947	0.872	0.878	0.922
BaseBridge	0.865	0.846	0.853	0.882
Kmin	0.989	0.919	0.952	0.968
GinMaster	0.968	0.962	0.964	0.966
Opfake	0.921	0.977	0.948	0.941
FakeInstaller	0.969	0.963	0.965	0.968
HummingBad	0.967	0.923	0.934	0.978
RedDrop	0.916	0.917	0.919	0.924
Overlay	0.956	0.950	0.954	0.961
Judy	0.918	0.922	0.923	0.934
Xbot	0.925	0.927	0.928	0.933

4.4. Experimental evaluation

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

$$PR = \frac{TP}{TP+FP}; \quad RC = \frac{TP}{TP+FN};$$

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

where *TP* is the number of malware that are correctly associated to the right family (True Positives), *TN* is the number of malware correctly identified as not belonging to the family (True Negatives), *FP* is the number of malware that are incorrectly associated to a family (False Positives), and *FN* is the number of malware that are not recognized as belonging to their family (False Negatives).

Table 9 presents the results obtained for the family identification task.

As shown in Table 9, with regard to the family detection, we obtain an accuracy ranging from 0.882 (obtained with BaseBridge family) to 0.987 (reached with the Geinimi one).

Table 10
Baseline performances in terms of Accuracy. The **Permissions** column is related to the accuracy obtained with the permission based method, while the **Op-codes** column is referring to the accuracy reached with the op-code based method.

Family	Permissions	Op-codes
Geinimi	0.768	0.838
Plankton	0.722	0.814
BaseBridge	0.701	0.788
Kmin	0.759	0.817
GinMaster	0.748	0.811
Opfake	0.721	0.809
FakeInstaller	0.777	0.831
HummingBad	0.785	0.842
RedDrop	0.726	0.814
Overlay	0.761	0.823
Judy	0.758	0.821
Xbot	0.761	0.832

In Table 10 we present the baseline accuracy: we highlight that the methods used as baseline (Canfora et al., 2013; 2015c; Mercaldo et al., 2016b) are referred to the discrimination between legitimate and malicious samples and, for this reason, they consider a binary classification problem. To consider these methods as baseline, we considered a multi-class classification, where each class is represented by a malware family.

From the baseline results in Table 10 emerges that the *op-codes* methods obtain better accuracy if compared to the *permissions* one. In particular, the accuracy for the *permissions* is ranging between 0.838 (Geinimi family) and 0.788 (BaseBridge family), while the accuracy for the *permissions* method is ranging between 0.701 and 0.768 for the same families. From these results, we can state that the proposed method outperforms the baseline in terms of accuracy.

Below we discuss in details the results we obtained in terms of FP. In the FP falls the samples incorrectly associated to a family under analysis (both legitimate samples than malware belonging to other families). From the experimental analysis it emerges that FP value contribution in terms of legitimate applications wrongly detected as belonging to a malicious family is really minimal, in fact only 8 legitimate applications (on the total of 3500 legitimate application) are labelled as belonging to a malicious family. The remaining FP are related to malicious samples which belonging family is wrongly identified.

The time to check t_{total} whether an application is belonging to a certain family is computed by adding following contributions:

- $t_{syscall}$: this value represents the time required to extract and store the system calls when the 25 events are sent. In average this value is equal to 15 min (i.e., 1.5 min to obtain each trace);
- t_{model}: this value represents the time required to build the model from the 10 syscall traces. In average this value is equal to 40 s;
- $t_{verification}$: this value represents the time required by the formal verification environment to verify the properties on the model. This value in average is equal to 2 min.

Considering the following contributions, the average t_{total} computed on all the evaluated applications is equal to 15 + 0.40 + 2 = 17.4 min.

With regard to the phylogeny tracking, Table 11 shows the percentages at which the various samples are verified for the subformula related to each family.

As shown in Table 11, the formal verification environment verifies the longest path of the $\varphi_{Geinimi}$ formula on the 68% of the Plankton and on the 51% of the GinMaster samples: this is symptomatic that both the families embed a part of the malicious payload of the Geinimi family, for this reason there is a descent relationship between the Geinimi family and Plankton and GinMaster. The relationship can be justified from the extensive use of network resources to steal sensitive information using a botnet without using SMSs to fraud credit to victims (Xiao et al., 2019), a common behaviour of the malicious payload belonging to Geinimi, Plankton and GinMaster. We highlight that for the family detection task the property generated from the superlog is considered (where each different system call path is verified with the $\wedge$ operator), while for the philogeny tracking task we consider a property related to the longest path.

The $\varphi_{Plankton}$ formula (second row in Table 11) is verified on the 72% of the samples belonging to the BaseBridge family: we recall that both the malicious payloads of these families are delivered using the update attack technique i.e., the package with the malicious behaviour is downloaded at runtime.

With the 74% of the Kmin samples verified, the proposed method labels the BaseBridge family as related to the Kmin one. In fact, the Kmin family exhibit a payload closed to the one of the BaseBridge family: the only difference is that the Kmin malicious payload is not able to kill the anti-malware processes.

The Kmin property is not verified in a percentage higher than 50%, for this reason there is no relation for this family.

The Ginmaster, as shown from the sixth row of Table 11 exhibits a relation with the Opfake family. In fact, both the malicious payloads are able to install external applications.

The Opfake property is verified on the 59% of the samples belonging to the FakeInstaller family and on the 59% of the samples exhibiting the HummingBad payload: this is symptomatic of a relationship, due to the fact that these malicious payload are able to send SMS to premium rate numbers (Sahay and Sharma, 2019).

The FakeInstaller property is verified in a percentage higher than 50% on the samples belonging to the Overlay (i.e., 54%) and Xbot (i.e., 62%) families, for this reason these two families exhibit a relation with the samples of the FakeInstaller family.

The HummingBad family has a relation with the Judy family, in fact the HummingBad property is verified on the 67% of the samples belonging to the Judy family.

The RedDrop, Judy, Xbox and Overlay formulae are not verified in a percentage higher than 50%, for this reason there is no relation for these families.

Fig. 5 shows the obtained phylogenetic tree.

The resultant phylogenetic tree is shown in Fig. 5.

As shown by the phylogenetic tree in Fig. 5, for all the families involved in the experiment, the draw the relationship coherently with the time of discovery of the malicious family, confirming the effectiveness of the proposed approach to correctly place malware families in the right position within the phylogenetic tree.

As shows by the experimental results, the process mining technique can be successfully adopted to extract significant relationship in terms of system call. In detail we demonstrated that the gathered processes are representative of malicious families: in fact the temporal logic properties built starting from the process are able to discriminate between malicious samples belonging to different families. With regard to the phylogenetic tree task, we confirmed the correct antecedent-descendant positioning of the families in the phylogenetic tree with the family discovery date. The proposed method can be useful for the detection of malicious behaviours end user side but also to security analyst, in fact the positioning in the phylogenetic tree can help analysts to understand the Android malware evolution.

4.5. Limitations

In the follow we describe the weaknesses related to the proposed approach:

Table 11
Formal analysis of malware evolution (column G stands for *Geinimi* family, P for *Plankton*, BB for *BaseBridge*, K for *Kmin*, GM for *GinMaster*, OF for *OpFake*, FI for *FakeInstaller*, HB for *HummingBad*, RD for *RedDrop*, OL for *Overlay*, J for *Judy* and X for *Xbot*).

Formulae	Family											
	G	P	BB	K	GM	OF	FI	HB	RD	OL	J	X
$\varphi_{Geinimi}$	-	**68%**	38%	34%	**51%**	14%	28%	21%	31%	19%	38%	23%
$\varphi_{Plankton}$	37%	-	**72%**	33%	18%	42%	39%	9%	26%	17%	28%	22%
$\varphi_{BaseBridge}$	27%	42%	-	**74%**	36%	26%	15%	8%	**55%**	11%	14%	37%
φ_{Kmin}	12%	16%	33%	-	27%	29%	9%	7%	45	6%	22%	14%
$\varphi_{GinMaster}$	16%	24%	13%	15%	-	**73%**	48%	35%	31%	11%	28%	32%
φ_{OpFake}	8%	44%	38%	19%	48%	-	**59%**	**76%**	42%	4%	37%	21%
$\varphi_{FakeInstaller}$	13%	39%	25%	33%	23%	27%	-	26%	38%	**54%**	5%	**62%**
$\varphi_{HummingBad}$	41%	27%	11%	29%	34%	31%	42%	-	37%	12%	**67%**	13%
$\varphi_{RedDrop}$	21%	14%	46%	43%	22%	26%	32%	18%	-	7%	23%	36%
$\varphi_{Overlay}$	27%	38%	25%	45%	34%	45%	28%	33%	14%	-	28%	12%
φ_{Judy}	23%	26%	47%	36%	23%	16%	39%	29%	11%	15%	-	46%
φ_{Xbot}	9%	29%	24%	41%	32%	46%	49%	19%	34%	44%	37%	-

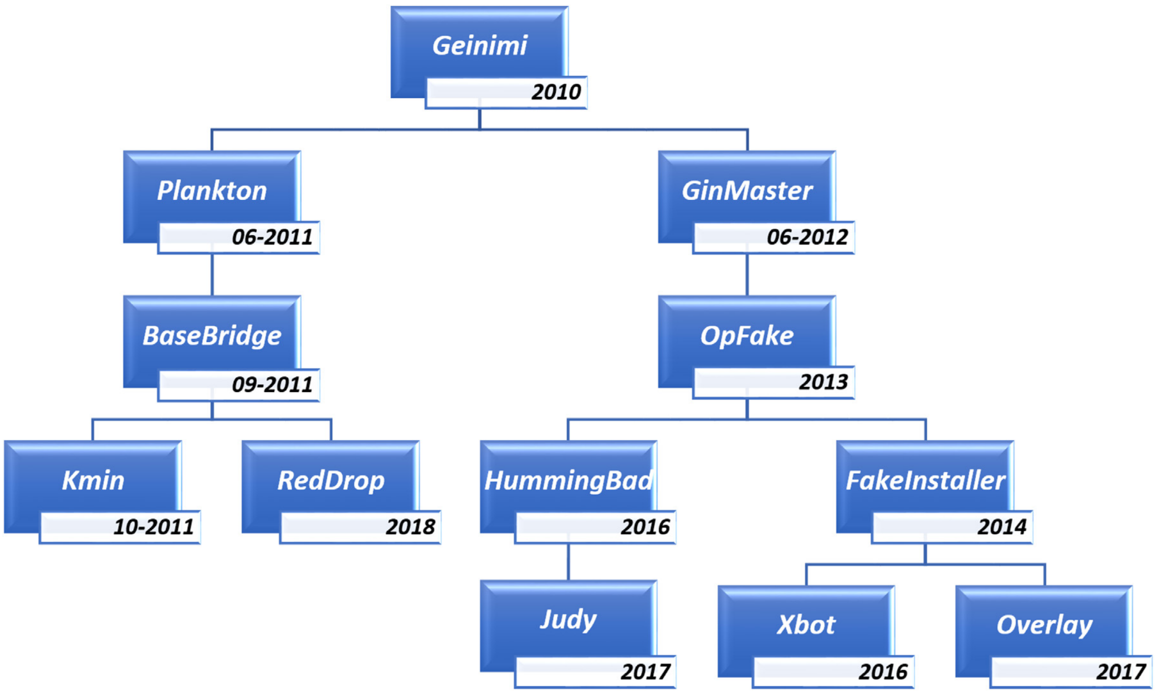

Fig. 5. The phylogenetic tree.

- to have more chances to trigger the malicious payload we sent to the application under analysis a set of 25 events: we are aware that the malicious payload can be triggered at specific time or when a complex combination of conditions is verified, for instance if the user is present. With the aim to make the proposed detection method fully automatised we do not considered user interactions;
- a malicious payload may not necessarily be implemented in the single piece of code that will be activated by some system events and only a part of the payload may be reflected in the traces. For instance, in the case of malware belonging to the "update attack" category, the malicious payload is delivered at run-time. We configure the emulator with the possibility to connect to the network, with the external storage, the geolocalization and the camera with the aim to offer to malware the best environment for carrying out their malicious behavior;
- the system call traces are generated from the Android emulator: an Android is able to know if is running on a real device or on an emulated one for instance, by simply checking if the *Build.MODEL*[9] variable contains the "Emulator" value. We anyway considered the adoption of the Android emulator to perform a wide experiment on several families to confirm the effectiveness of the proposed method.

5. Related work

In this section the current state of the art literature is discussed, considering that we propose a method for mobile malware family detection and philogeny tracking we firstly discuss the state in the art in the malware family detection and secondly in the malware philogeny analysis.

5.1. Malware family identification

Several works in literature explore the effectiveness system calls (Canfora et al., 2014; 2013; Jeong et al., 2014; Reina et al., 2013) for the detection of malicious Android families. For instance

[9] https://developer.android.com/reference/android/os/Build.

in (Canfora et al., 2013), a method for detecting mobile malware is proposed. This method considers three metrics respectively measuring the occurrences of a reduced subset of system calls, the weighted sum of a subset of permissions which the application requires and a set of combinations of permissions. Their experiment considers a sample of 400 real world legitimate and malicious applications, obtaining a precision of 74%. Another approach to the malware detection and malicious behaviour identification based on system calls analysis is implemented in the CopperDroid (Reina et al., 2013) tool. This tool requires the customization of the Android emulator to track syscalls. The usage of the emulator to gather system call is proposed in (Wang et al., 2009). This method is validated on a data-set of 1600 malicious apps, and was able to find the 60% of the malicious apps belonging to one data-set (the Genoma Project) and the 73% of the malicious apps included in the second data-set (the Contagio data-set). Also in (Jeong et al., 2014) system calls used to read/write operations on files are checked in order to detect malicious behavior. The authors used a customized kernel on a real device, and the sample included 2 malicious applications developed by the authors.

Suarez-Tangil and Stringhini (2018) focus their efforts to discern malicious component from the legitimate one in repackaged Android malware. Basically they consider control flow graphs generated from code fragments of the application under analysis. They highlight that most research papers on Android malware detection are focused on outdated repositories, as the MalGenome project (Zhou and Jiang, 2012) and the Drebin (Arp et al., 2014) data-sets. The output of this work is a malware fingerprint basically composed by the list of methods characterizing the malicious family. Basically authors mine methods that are common to Android applications belonging to the same family, assuming that samples belonging to the same family exhibit the same purpose and are developed by the same authors, and therefore there will be code in common with all the malware samples in the family. For instance, with regard to the KMin family, Suarez-Tangil and Stringhini extract a fingerprint composed by 49 different methods[10]: in the follow we discuss how the fingerprint gathered from the Kmin family by Suarez-Tangil and Stringhini can be compared to our Kmin family temporal logic formula shown in Table 8. Clearly it is not possible to perform a direct comparison because Suarez-Tangi and Stringhini consider a static analysis where a list of method is gathered, while our analysis is related to system call paths. For instance, the *ioctl* in the logic formula shown in Table 8, related for device-specific input/output operations, can be considered for the invocation of following methods aimed to gather information from the device: *getActiveNetworkInfo, getApplicationInfo, getRunningServices*. The *writev* system call writes data from a buffer declared by the user to a given device: can be related to the invocation of the *sendTextMessage* method of the *SmsManager* class. Moreover Suarez-Tangi and Stringhini highlight that the *connect* method of the *HttpURLConnection* is distinctive of the Kmin family, this method can request the functionalities of the *rcvmsg* syscall, aimed to receive a message from a socket.

DroidScope (Yan and Yin, 2012) uses a customized Android kernel to reconstruct semantic views with the aim to collect detailed applications execution traces. The obtained detection rate is of 100%, but it is evaluated only on two Android malicious applications.

Researchers in (Canfora et al., 2015b) present an approach aimed for detecting Android malware families. The method is based on the analysis of system calls sequences and is tested obtaining an accuracy of 97% in mobile malware identification using 3-gram syscall as feature.

There are several limitation to the adoption of the dynamic analysis: the first one is that dynamic analysis is time intensive and resource consuming, thus elevating the scalability issues. The reason why we resort to dynamic analysis is its ability to activate the malicious behaviour, triggered only whether certain conditions happen (Zhou and Jiang, 2012) (for instance, by system events with regard to the Android environment). Furthermore, another weakness of the current state of the art approaches for detecting malware by dynamic analysis is that they require to build the kernel (with the exception of the method proposed in reference (Canfora et al., 2015b)), making the adoption of the proposed techniques difficult in the real-world.

Researchers in (Su et al., 2018) discuss a method for Android malware detection exploiting a set of static features. They consider unsupervised machine learning techniques to build models with the considered feature set, statically obtained from permission invocations, strings and code patterns. They obtain an accuracy of 79.53%.

The Alde (Liu et al., 2019) framework employs static analysis and dynamic analysis to detect the actions of users collected by analytics libraries. Moreover, Alde analyses what private information can be leaked by the apps that use the same analytics library. The main outcome of the study is some apps indeed leak users' personal information through analytics libraries even though their genuine purposes of using analytics services are legal.

Supservised machine learning techniques are considered by authors in (Wang et al., 2018), where they design an ensemble model, composed by several classifier, to detect whether an Android application is malicious or not. An accuracy equal to 96.91% is obtained with an ensemble model built with four different classification algorithms: Support Vector Machine, RanfomForest, k-nearest neighbors and Classification And Regression Trees.

5.2. Malware phylogeny

Below we discuss the current literature focused on malware phylogeny. Authors in (Karim et al., 2005) discuss a method to build phylogeny models. They consider code permutation to obtain the malware tree. Authors state that the proposed models support the detection of new malware variants. Compared with the proposed approach, this method is static and hence does not require malware execution. Moreover, with respect to the proposed approach, this method is less robust to code modifications that cannot be represented as permutations.

Researchers in (Walenstein and Lakhotia, 2012b) define a framework to gather malware evolution relations in terms of path patterns. The limitation of this approach is that the model's definition of source code excludes machine generated code. This is very restrictive considering that usually malicious code is automatically generated from the existing malicious applications. Differently, the proposed method considers the system calls directly generated from the running machine code.

6. Conclusion and future work

Considering the current weaknesses of free and commercial signature based mobile anti-malware, in this paper we propose a method aimed to detect the belonging family of a mobile malicious application. Moreover, the proposed method is also able to track the antecedent-descendant relationship, with the aim to support the malware analyst in new malicious behaviour recognition. The main novelty of the proposed work is represented by the adoption of the process mining technique to automatically infer temporal logic properties aimed to detect Android malware families and

[10] https://github.com/gsuareztangil/adrmw-measurement/blob/master/results/explanatory_Malgenome.txt.

for phylogenetic tree tracking. The proposed method models the system call sequences generated by mobile applications in terms of automaton, thus applying the model checking technique it verifies a set of property automatically inferred exploiting the process mining technique. We obtain for the family identification task an accuracy ranging between 0.882 and 0.987, by analyzing a data-set composed of 12,604 Android samples. Moreover, we demonstrate how the proposed method is able to reconstruct the malicious behaviour phylogenetic tree.

As future work we plan to extend the proposed method considering the data dependency graph, with the aim to grasp the passage of variables between different methods (to detect the exchange of sensitive information). Moreover, we are investigating whether the proposed method is able to detect the so-called colluding applications, i.e., groups of apps that collaborate to run a small and undetectable role in a larger malicious operation.

Declaration of Competing Interest

The authors whose names are listed immediately below certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

References

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

Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E., 2009. Scalable, behavior-based malware clustering. In: NDSS, vol. 9. Citeseer, pp. 8–11.

Canfora, G., Di Sorbo, A., Mercaldo, F., Visaggio, C.A., 2015. Obfuscation techniques against signature-based detection: a case study. In: 2015 Mobile Systems Technologies Workshop (MST). IEEE, pp. 21–26.

Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2018. LEILA: formal tool for identifying mobile malicious behaviour. IEEE Trans. Softw. Eng.

Canfora, G., Medvet, E., Mercaldo, F., Visaggio, C.A., 2014. Availability, Reliability, and Security in Information Systems: IFIP WG 8.4, 8.9, TC 5 International Cross–Domain Conference, CD-ARES 2014 and 4th International Workshop on Security and Cognitive Informatics for Homeland Defense, SeCIHD 2014, Fribourg, Switzerland, September 8–12, 2014. Proceedings. Springer International Publishing, Cham, pp. 226–238.

Canfora, G., Medvet, E., Mercaldo, F., Visaggio, C.A., 2015. 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.

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

Canfora, G., Mercaldo, F., Visaggio, C.A., 2015. Mobile malware detection using op–code frequency histograms. In: Proceedings of International Conference on Security and Cryptography (SECRYPT).

Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J., 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on. IEEE, pp. 177–186.

Cimitile, A., Mercaldo, F., Martinelli, F., Nardone, V., Santone, A., Vaglini, G., 2017. Model checking for mobile android malware evolution. In: Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 24–30. doi:10.1109/FormaliSE.2017.4.

Clarke, E.M., Grumberg, O., Peled, D., 2001. Model Checking. MIT Press.

Cleaveland, R., Sims, S., 1996. The NCSU concurrency workbench. In: Alur, R., Henzinger, T.A. (Eds.), CAV. Springer, pp. 394–397.

Dumitras, T., Neamtiu, I., 2011. Experimental challenges in cyber security: a story of provenance and lineage for malware. CSET 11, 2011–2019.

Günther, C.W., Rozinat, A., 2012. Disco: discover your processes. BPM (Demos) 940, 40–44.

Günther, C.W., Van Der Aalst, W.M., 2007. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In: International Conference on Business Process Management. Springer, pp. 328–343.

Haq, I., Chica, S., Caballero, J., Jha, S., 2018. Malware lineage in the wild. Comput. Secur. 78, 347–363.

Hu, X., Chiueh, T.-c., Shin, K.G., 2009. Large-scale malware indexing using function–call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. ACM, pp. 611–620.

Jang, J., Brumley, D., Venkataraman, S., 2011. BitShred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. ACM, pp. 309–320.

Jeong, Y.-s., Lee, H.-t., Cho, S.-j., Han, S., Park, M., 2014. A kernel-based monitoring approach for analyzing malicious behavior on android. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1737–1738.

Jiang, X., Zhou, Y., 2013. Android Malware. Springer Publishing Company, Incorporated.

Jilcott, S., 2015. Scalable malware forensics using phylogenetic analysis. In: Technologies for Homeland Security (HST), 2015 IEEE International Symposium on. IEEE, pp. 1–6.

Kaspersky. last visit 8 June, 2019. https://www.kaspersky.com/resource-center/threats/mobile.

Karim, M.E., Walenstein, A., Lakhotia, A., Parida, L., 2005. Malware phylogeny generation using permutations of code. J. Comput. Virol. 1 (1–2), 13–23.

Kozen, D., 1983. Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354. doi:10.1016/0304-3975(82)90125-6.

Li, H., Zhou, S., Yuan, W., Li, J., Leung, H., 2019. Adversarial-example attacks toward android malware detection system. IEEE Syst. J.

Li, L., Li, D., Bissyandé, T.F., Klein, J., Cai, H., Lo, D., Le Traon, Y., 2017. On locating malicious code in piggybacked android apps. J. Comput. Sci. Technol. 32 (6), 1108–1124.

Liu, X., Liu, J., Zhu, S., Wang, W., Zhang, X., 2019. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. IEEE Trans. Mob. Comput.

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

Mercaldo, F., Visaggio, C.A., Canfora, G., Cimitile, A., 2016. Mobile malware detection in the real world. In: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, pp. 744–746.

Milner, R., 1989. Communication and Concurrency. Prentice Hall.

Nagra, J., Collberg, C., 2009. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education.

Oberheide, J., Mille, C., 2012. Dissecting the android bouncer. SummerCon.

Rastogi, V., Chen, Y., Jiang, X., 2013. 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.

Rastogi, V., Chen, Y., Jiang, X., 2014. Catch me if you can: evaluating android anti-malware against transformation attacks. IEEE Trans. Inf. ForensicsSecur. 9 (1), 99–108.

Reina, A., Fattori, A., Cavallaro, L., 2013. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of EuroSec.

Rosenblum, N., Miller, B.P., Zhu, X., 2011. Recovering the toolchain provenance of binary code. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp. 100–110.

Sahay, S.K., Sharma, A., 2019. A survey on the detection of android malicious apps. In: Advances in Computer Communication and Computational Sciences. Springer, pp. 437–446.

Santone, A., Vaglini, G., 2016. Conformance checking using formal methods. In: Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24, - 26, 2016.. SciTePress, pp. 258–263.

Spreitzenbarth, M., Echtler, F., Schreck, T., Freling, F.C., Hoffmann, J., 2013. Mobilesandbox: looking deeper into android applications. 28th International ACM Symposium on Applied Computing (SAC).

Stirling, C., 1989. An introduction to modal and temporal logics for CCS. In: Yonezawa, A., Ito, T. (Eds.), Concurrency: Theory, Language, And Architecture. Springer, pp. 2–20.

Su, D., Liu, J., Wang, W., Wang, X., Du, X., Guizani, M., 2018. Discovering communities of malapps on android-based mobile cyber-physical systems. Ad Hoc Netw. 80, 104–115.

Suarez-Tangil, G., Stringhini, G., 2018. Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned. arXiv:1801.08115, Technical Report, 2018.

Suarez-Tangil, G., Tapiador, J., Peris-Lopez, P., Ribagorda, A., 2014. Evolution, detection and analysis of malware for smart devices. Commun. Surv. Tutorials 961–987.

Tecktalk, 2019. https://techtalk.pcpitstop.com/2018/11/08/android-malware/.

Walenstein, A., Lakhotia, A., 2012. A transformation-based model of malware derivation. In: Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on. IEEE, pp. 17–25.

Walenstein, A., Lakhotia, A., 2012. A transformation-based model of malware derivation. In: Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on, pp. 17–25.

Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X., 2018. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Future Gener. Comput. Syst. 78, 987–994.

Wang, X., Jhi, Y.-C., Zhu, S., Liu, P., 2009. Detecting software theft via system call based birthmarks. In: Proceedings of the 2009 Annual Computer Security Applications Conference. IEEE Computer Society, Washington, DC, USA, pp. 149–158.
Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K., 2019. Android malware detection based on system call sequences and LSTM. Multimed. Tools Appl. 78 (4), 3979–3999.
Yan, L.K., Yin, H., 2012. DroidScope: Seamlessly reconstructing the os and Dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium. USENIX Association, Berkeley, CA, USA. 29–29.
You, I., Yim, K., 2010. Malware obfuscation techniques: a brief survey. In: Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on. IEEE, pp. 297–300.
Zhang, Y., Ren, W., Zhu, T., Ren, Y., 2019. SaaS: a situational awareness and analysis system for massive android malware detection. Future Gener. Comput. Syst. 95, 548–559.
Zheng, M., Lee, P.P., Lui, J.C., 2012. 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.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: characterization and evolution. In: Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012).

Mario G.C.A. Cimino is an Associate Professor at the Department of Information Engineering of the University of Pisa (Italy). He received the Ph.D. degree in Information Engineering from the University of Pisa in 2007. From 2003 to 2006, as a Ph.D. student he joined the Department of Information Engineering of the University of Pisa, working on computational intelligence and information systems. Since April 2006, he spent six months as a visiting scholar at the Electrical & Computer Engineering Department of the University of Alberta, Edmonton (Canada), under the supervision of Prof. W. Pedrycz, for a research activity on neurocomputing and granular computing.

Nicoletta De Francesco graduated with honors in Information Science from the Faculty of Mathematical, Physical and Natural Sciences of the University of Pisa in 1974. From 1981 to 1988 she was a researcher at the Computer Science Department of the University of Pisa. From 1989 she was an associate professor, for a year in Salerno and then at the Faculty of Engineering of the University of Pisa. Since October 2000 she is full professor of Information Processing Systems. He carried out a wide and diversified didactic activity, first in the degree course in Computer Science and then in that of Computer Engineering. She is a member of the PhD course in Information Engineering.

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

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

Gigliola Vaglini is full professor of Computer Science at the Department of Engineering of the University of Pisa and is a member of Department of Information Engineering of the University of Pisa. Born in Pisa in 1952, she graduated in Information Science at the University of Pisa in 1974. She was a two-year research fellow at the Computer Science Department of the University of Pisa and then an associate professor from 1989 to September 2002, first at the Faculty of SMFN of the Federico II University of Naples and then to the Faculty of Engineering of the University of Pisa. Since October 2002 she is full professor at the Department of Information Engineering of the University of Pisa.