International Journal of Information Security (2019) 18:257–284
https://doi.org/10.1007/s10207-018-0415-3

REGULAR CONTRIBUTION

Dynamic malware detection and phylogeny analysis using process mining

Mario Luca Bernardi[1] · Marta Cimitile[2] · Damiano Distante[2] · Fabio Martinelli[3] · Francesco Mercaldo[3]

Published online: 29 June 2018

Abstract
In the last years, mobile phones have become essential communication and productivity tools used daily to access business services and exchange sensitive data. Consequently, they also have become one of the biggest targets of malware attacks. New malware is created everyday, most of which is generated as variants of existing malware by reusing its malicious code. This paper proposes an approach for malware detection and phylogeny studying based on dynamic analysis using process mining. The approach exploits process mining techniques to identify relationships and recurring execution patterns in the system call traces gathered from a mobile application in order to characterize its behavior. The recovered characterization is expressed in terms of a set of declarative constraints between system calls and represents a sort of run-time fingerprint of the application. The comparison between the so defined fingerprint of a given application with those of known malware is used to verify: (1) if the application is malware or trusted, (2) in case of malware, which family it belongs to, and (3) how it differs from other known variants of the same malware family. An empirical study conducted on a dataset of 1200 trusted and malicious applications across ten malware families has shown that the approach exhibits a very good discrimination ability that can be exploited for malware detection and malware evolution studying. Moreover, the study has also shown that the approach is robust to code obfuscation techniques increasingly being used by nowadays malware.

Keywords Malware detection · Malware evolution · Malware phylogeny · Security · Process mining · Linear temporal logic · Declare

1 Introduction

Over the last years, the growth in the number of applications for mobile phones has changed the way to communicate and to access information. Given their increasing capabilities, mobile phones 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. As a consequence, they have become target of continuous attacks obtained through an increasing number of malicious software that becomes everyday more aggressive and sophisticated [23]. This malicious code is usually generated from existing malicious software [23] using some automatic tools that generate new malware from libraries and code borrowed from robust networks used for malware code exchange. In response to this phenomenon, a wide range of antimalware programs have been developed to perform malware detection and to study malware phylogeny. Malware detection aims to identify malware (i.e., malicious software) in software applications [8].

Malware phylogeny analysis is meant to extract a model, inspired by biological research, that highlights similarities and relationships between a set of malware [19]. This model can then be used to support the identification of malware evolution trends, the individuation of new strategies to dissect malware samples, and the discovery of software vulnerabil-

✉ Marta Cimitile
marta.cimitile@unitelmasapienza.it

Mario Luca Bernardi
m.bernardi@unifortunato.eu

Damiano Distante
damiano.distante@unitelmasapienza.it

Fabio Martinelli
fabio.martinelli@iit.cnr.it

Francesco Mercaldo
francesco.mercaldo@iit.cnr.it

1 Giustino Fortunato University, Benevento, Italy

2 University of Rome Unitelma Sapienza, Rome, Italy

3 Institute for Informatics and Telematics, CNR, Pisa, Italy

ities (e.g., the vulnerabilities present in an application may be inherited by applications derived from it) [23,30].

In this paper, starting from the assumption that any malicious behavior is implemented by specific sequences of system calls, we propose an approach for malware detection and malware phylogeny analysis that adopts process mining techniques for the analysis of the system call traces generated by an application. Process Mining (PM) is a process management technique for the analysis of business processes based on event logs [43]. In our approach, we use PM to analyze the system call traces of a mobile application, assuming that similarities and derivations between system calls can be discovered and modeled in the system call traces similarly to what applies for business process activities in business process logs. According to this, in our approach, we use PM to derive a characterization of the behavior of a (trusted or malware) mobile application from a set of system call traces gathered from it in response to a set of operating system events.

We use a tool called Declare Miner[1] that allows to discover a model from a set of traces collected from different runs of the trusted/malware applications. This model captures the behavior of the application characterizing and discriminating the malware and supporting the study of its phylogeny.

Such model is expressed as a set of declarative constraints between system calls using the Declare process modeling language [34] and is named *System Calls Execution Fingerprint* (SEF).

Even if the proposed approach can be applied to all the existing mobile platforms, in this work the focus is on the Android platform as, according to recent survey [29], it is the favorite target of mobile threats. This is not surprising considering that in the smartphone operating system (OS) market, Google's Android extended its lead by capturing more than 80% of the total market in the fourth quarter of 2016 and that in the same period the sales of smartphones worldwide totaled 431 million of units [18]. Moreover, current solutions to protect Android users are still inadequate. For example, several antimalware adopt a signature-based malware detection approach requiring antimalware vendors to be aware of new malware code in order to identify their signatures (in the form of fixed strings and regular expressions) and to send out updates regularly. Furthermore, there are new techniques that evade signature-based- detection approaches by including various types of source code transformation and simple forms of *polymorphic attacks* [37]. Malware detection in Android is also affected by another problem: differently from antimalware 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 antimalware 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 [33]. 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. However, since signature-based detection techniques are evaded by attackers with new malware which is increasingly aggressive, new techniques that go beyond to detect malware software in Android devices are required.

With respect to existing approaches to malware detection and phylogeny model extraction, our approach introduces the following novelties:

- the SEF model extracted to characterize a malware behavior is obtained using a declarative constraint-based language that allows exploiting a much wider range of properties and relationships between system calls in system call traces;
- the same model can be effectively used as a malware fingerprint for both malware detection and phylogeny analysis.

Finally, the 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 [8,30]. With respect to these works, this paper presents:

- an integrated approach for both phylogeny analysis and malware detection;
- a wider experimentation involving a larger set of applications belonging to an increased number of malware families;
- a robustness analysis aimed at assessing the impact of behavioral-preserving code transformations on the detection capability of our approach.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 presents the background of our study. Section 4 presents the proposed approach. Section 5 evaluates the effectiveness of the approach by testing it on a dataset of 10 malware families and 1200 malicious and trusted applications. Section 6 evaluates the robustness of the approach with regard to a set of well-known code transformation techniques. Section 7 discusses threats to validity. Finally, Sect. 8 provides some conclusive remarks for our work.

[1] http://www.win.tue.nl/declare/declare-miner/.

2 Related work

2.1 Malware detection

Several studies about malware detection have been presented in the last years. They differ from each other for the features (or characterization) used to discriminate between malware and trusted applications. These features can be obtained with static or dynamic analysis techniques.

2.1.1 Static methods

Static methods capture suspicious patterns from the code or related artifacts (e.g., metadata) and consist in analyzing malicious software without executing it. Some common patterns used in static analysis include string signature, byte-sequence *n*-grams, control flow graph, syntactic library call, operational code frequency distribution. An approach which considers control flow has been proposed in [39]. It is based on AndroGuard [1], a tool that starting from the extraction of mobile applications features allows training a one-class Support Vector Machine and to extract permissions and control flow graphs from packaged Android applications. This method has been tested on a collection of 2081 trusted applications and 91 malicious ones. The results show a very low false negative rate but also a high false positive one. This is due to the considered features i.e., permissions and control flow graph, that are susceptible to the trivial obfuscation techniques. As demonstrated in [2,37] attackers are able to write malicious code with the ability to evade the detection based on these kinds of features. This is the reason why we resort to a dynamic approach system call based, evaluating the proposed method on a real-world mobile malware dataset and using an obfuscated version of the samples generated with the common morphing techniques. In [16], the control flow to detect application communication vulnerabilities are also considered. In this study, ComDroid, a tool to examine inter-application communication in Android, is proposed. The tool is applied to analyze 20 applications founding 34 exploitable vulnerabilities (12 of the 20 analyzed applications have at least one vulnerability). The authors presented also several classes of potential attacks on mobile applications: outgoing communication can put an application at risk of Broadcast theft (including eavesdropping and denial of service), data theft, result modification, and Activity and Service hijacking. An incoming communication can put an application at risk of malicious Activity and Service launches and Broadcast injection. With respect to the proposed method, ComDroid is focused on the data-leakage identification, i.e. the data exfiltration process. Differently, we propose a mobile malware detector: as a matter of fact, mobile malware usually gather information from the infected devices, but it also able to perform a plethora of harmful actions (i.e., download at run-time malicious packages and/or send SMS to premium-rate numbers) that are not data leakage related. In addition, we evaluate a dataset composed of 1200 trusted and malicious applications, while the ComDrod tool is evaluated using 20 applications. Recently, the possibility of identifying the malicious payload in Android malware using a model checking-based approach has been explored in [5,31,32]. Starting from payload behavior definition, the authors formulate logic rules and then test them by using a real-world dataset composed of Ransomware, DroidKungFu, Opfake families, and update attack samples. Despite these approaches obtain high ratio in mobile malware detection, authors starting from the payload behavior definition define a set of logic rules that need the malware analyst experience in order to be formulated. This is the reason why these methods are not fully automated. Differently, the proposed method, combining process mining and machine learning, does not require the knowledge of the malware internals in order to build a classifier. Several alternative techniques, considering other features derived from static analysis, have been recently investigated. For example, in [52], a signature-based analytic system for Android malware automatic management, collection, analysis, and extraction, is proposed.

The approach is supported by a tool, called DroidAnalytics, that works at opcode level. In the testing phase, DroidAnalytics detects 2475 Android malware from 102 different families with 327 of them being zero-day malware samples belonging to six families found in a dataset of 150,368 Android applications. They also proposed a methodology to detect zero-day malware that allowed them to discover other three new malware families: AisRs (with 77 samples), AIProvider (with 51 samples) and G3app (with 81 samples). All families discovered are repackaged samples from legitimate applications. As demonstrated in [49], the opcode level based techniques are not able to reach a high-performance detection ratio, for this reason, we propose a method based on dynamic analysis able to be resistant to the common obfuscation techniques and to obtain a higher detection ratio. Starting from the consideration that mobile malware is usually employed to gather sensitive information from device, authors in [17] propose a compiler to uncover usage of phone identifiers and locations. Here, a large set of Android applications collected from the market are analyzed to identify a set of dataflow, structure, and semantic patterns. As previously explained, DroidMOSS [53] adopts a fuzzy hashing technique to effectively localize and detect possible changes from app repackaging. The app similarity measurement system developed by authors shows that a range from 5 to 13% of app hosted on several third-party marketplaces and the official Android Market are repackaged. In addition to this, 200 samples from each third-party marketplace have been analyzed. Moreover, authors detect whether the applications are repackaged from some official

Android Market apps. These approaches focus on the differences between a trusted application and the same one with the malicious payload embedded. As discussed in [4,54] several malware families are standalone, i.e., they do not derive from trusted applications with the malicious payload embedded, this is the reason why this approach cannot be able to detect these kinds of threats. Concerning the static approaches, even trivial obfuscation techniques (for example, junk code insertion or code reordering) can be used to evade the detection techniques of this type of antimalware. The limits of the static approaches are overcome by dynamic detection approaches, which are usually more robust with respect to the code obfuscation techniques currently employed by malware developers.

2.1.2 Dynamic methods

Dynamic methods observe the behavior of a malicious application, while it is being executed in a controlled environment (virtual machine, emulator, sandbox etc.) or using a real device. These methods are more effective with respect to the static ones and do not require the executable to be disassembled. There are several studies proposing malware detection techniques based on dynamic analysis of mobile applications and some of them are focused on the analysis of system calls [11,13,21,24,38,40,42]. In [11], a method for detecting mobile malware is proposed. The method is based on three metrics respectively evaluating the (i) occurrences of a reduced subset of system calls; (ii) a function of a subset of permissions which the application requires and (iii) the set of combinations of permissions. The experimentation considers a sample of 200 real-world malicious apps and 200 real-world trusted apps scoring a precision of 74%. Another approach to the malware detection based on the system calls analysis is proposed by CopperDroid [38]. Authors customize the Android emulator to track syscalls. With respect to these methods, our proposed approach does not require any customization of the Android kernel, this is the reason why this method is applicable immediately by the user. In addition, our method is able to track mobile malware phylogeny. Similarly, in [47], the authors use an emulator to analyze syscalls. The method was validated on a set of 1600 malicious apps and was able to find the 60% of the malicious apps belonging to one sample (the Genoma Project) and the 73% of the malicious apps included in the second sample (the Contagio dataset). Differently, we evaluate our method using the most recent Android malware dataset available for research purpose i.e., the Drebin one [4,41] obtaining a precision equal to 0.94 in the identification of the most recent threats in Android malware landscape. Also in [24] read/write operations system calls are checked in order to detect malicious behavior. The authors used a customized kernel on a real device but the evaluation is performed on a synthetic dataset composed of 2 malicious applications developed by the authors.

In [42], the response of an application is based on a subset of system calls generated from bad activities in background that are activated by user interfaces events. The experimentation is based on the use of an Android emulator and included 2 malicious samples of the DroidDream family.

An approach aiming at detecting anomalies in the Android system is also proposed in [40]. Here, the authors use the view from Linux-kernel such as network traffic, system calls, and file system logs. Basically, authors take a look on how Android-based smartphones can be secured. Starting from the assumption that the Android kernel is derived from the Linux one, they evaluate a set of Linux executable files (not Android samples), i.e., authors do not evaluate Android malware. In [21], a method based on the analysis of an application log and of a set of system calls related to management of file, I/O and processes is introduced. The evaluation phase considers 230 applications mostly downloaded from Google Play. From these, 37 applications steal personal sensitive data, 14 applications execute exploit code and 13 applications aim to damage the system. Differently, from our approach, the above discussed one depends on the Android version (an Android 2.1 based modified ROM image is used). This limits the applicability of the approach since requires a patch to running kernel.

DroidScope [50] 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. Also, in this case, an Android kernel customization is required limiting the applicability.

Authors in [48] propose an Android malware detection mechanism which is based on feature-network. The usage of these features (i.e., ingoing and outgoing information) is more related to the data-leakage information than to solve the malware detection issue.

Arora et al. [3] analyze the network traffic features building a rule-based classifier to detect Android malware. Their experimental results suggest that the approach is accurate and it detects more than 90% of the traffic samples. The work is focused only on Android malware which is remotely controlled or leaks information to some remote server, for a total of 27 different families.

In [14], a method for detecting Android malware is presented. The method is based on the analysis of system calls sequences and is tested obtaining an accuracy of 97% in mobile malware identification. They use machine learning to automatically learn 1, 2 and 3-gram from syscall traces. Conversely, the proposed approach considers, using process mining techniques, a richer behavioral model based on relationships and recurring execution patterns among system call within traces. Moreover, our method is also able to track the malware phylogeny and is robust with respect to code obfuscation techniques, increasingly being used by nowadays malware.

Looking at the discussed dynamic malware detection methods, we can generalize that dynamic analysis is time intensive and resource consuming, thus increasing the scalability issues.

In order to mitigate this limitation, we gather only the system calls generated by the application under analysis in response to system events.

Finally and differently from our approach, all the discussed methods (with the exception of the one in reference [14]) require to recompile the kernel.

2.2 Malware phylogeny

Several studies are focused on malware phylogeny. As an example, in [26] a method to realize phylogeny models is introduced. In this method, some features, called n-perms, are used to match possibly permuted code and allow to obtain a malware tree. Looking to the obtained malware tree authors suppose that phylogeny models based on n-perms may support the identification of new malware variants and the reconciliation of inconsistencies in malware naming. Compared with our 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.

In [46], a framework defining malware evolution relations in terms of path patterns on derivation graphs is proposed. 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. Our method considers the syscall gathered from the application under analysis keeping trace of the machine-generated code behavior.

Another approach allowing to find similarities and differences between malware variants and strains is proposed in [15]. It uses some clustering algorithms to obtain a taxonomy of the malware at hand. Unfortunately, a formal comparison to a reference phylogeny is missed. In [28], a data-centric approach based on packets inspections aiming to automatically identify and quantify the shellcode similarity, is proposed. This approach is used to create a shellcode phylogeny for a given vulnerability and differs from ours because it is mainly based on a behavior analysis. Researchers in [27] collect logs derived from instructions executed, memory and register modifications in order to build the phylogenetic tree. In addition, they show that network resources were more useful for visualizing short nop-equivalent code metamorphism than trees. They consider execution traces of alternating APIs and user procedures, whereas system calls traces considered in our method are not affected by API version. In [2], a dynamic and static analysis approach based on multiple kernels learning to define a new malware classification is proposed. This framework places a similarity metric in some different view adopting a kernel. Moreover, it employs multiple kernels learning to discover a weighted combination of data sources achieving the best classification accuracy of a support vector machine classifier. This approach cannot be directly compared to our one since it proposes a framework to improve classification accuracy rather than a detection approach. However, it can be effectively integrated with our approach in order to obtain better results.

3 Background

3.1 Mobile malware families

Malicious programs are frequently related to previous program versions through evolutionary relationships. The knowledge of these relationships can be useful for both constructing a phylogeny model and supporting malware detection through the analysis of new malware based on the similarities with the known ones [26]. In particular, malware applications can be grouped into families. Each family defines a set of behaviors and properties that, to a certain extent, are common to all its members. Starting from the analysis of security announcements, threat reports, articles in researchers' blogs, and data published by existing mobile antimalware companies, in [54] a list of 49 Android malware families, with their characteristics, is reported. The list of top 10 malware families (i.e., the list of known malware families having the highest number of known samples) is reported in Table 1, with their main properties. For each malware family, the table shows (i) the name of the family (first column), (ii) a brief description (second column), (iii) the installation type (third column), which refers to the way the malicious payload is installed ('r' for repackaging, 's' for standalone, and 'u' for the update attack), and (iv) the activation mechanism (fourth column), i.e., the system events that activate the malicious behavior.

Table 2 shows a list of the most relevant system events that an application can receive during its life cycle and that, according to several studies [25,54], are known to trigger a malware payload most frequently. 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 on an Android device as the system finishes its booting process, a perfect time for a malware to kick off its background services. By listening to this event, a malware

Table 1 The top 10 malware families

Family	Description	IT	AE
DroidKungFu 1–4	It installs a backdoor that allows attackers to access the smartphone when they want and use it as they please	r	BOOT, BATT, SYS
Opfake	It demands payment for the application content through premium text messages	r	MAIN
GinMaster	It contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and install applications	r	BOOT
AnserverBot	It repackages into the host app with two hidden apps	r,u	BOOT, NET, CALL
BaseBridge	It sends information to a remote server running one or more malicious services in background	r,u	BOOT, SMS, NET, BATT
Kmin	It is similar to BaseBridge, but does not kill antimalware processes	s	BOOT
Pjapps	It is a Trojan horse that has been embedded on the third-party applications and opens a back door on the compromised device	r	BOOT, SMS, BATT
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
Adrd	It is close to Geinimi but with less server side commands	r	BOOT, CALL
DroidDream	It gains root access to device to access unique identification information	r	MAIN

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 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.

Starting from existing malware, new variants are released by malware writers to get as much mileage as possible from the original code and to create new undetectable malware. Therefore, malware variants are new strains and slightly modified versions of a malware belonging to the same malware family. These malware variants include increasingly sophisticated techniques for obfuscating malicious behavior in order to elude detection strategies employed by current antimalware products [54]. In particular, polymorphism is one of the obfuscating techniques that are rapidly spreading among malware targeting mobile applications [36].

For instance, the first version of DroidKungFu malware was detected in June 2011. Successively, security researchers detected the second version DroidKungFu$_2$ and the third version DroidKungFu$_3$ in July and August 2011, respectively.

Finally, the fourth version DroidKungFu$_4$ was detected in October 2011. Similarly, to the previous generations of DroidKungFu, the latest version is able to install a backdoor that gives hackers full control of the mobile device. Therefore, while previous versions of DroidKungFu retrieved instructions from a remote "command and control" server and stored the URL for the server in plain text, DroidKungFu$_3$ and DroidKungFu$_4$ encrypt the URL, making it harder to identify and block them. Moreover, starting from this version, the vulnerable code is encrypted, making more difficult to identify the malware [54]. Finally, starting from DroidKungFu$_3$, after installing the embedded payload, it is masked as an official Google update, thus increasing its diffusion and reducing users' diffidence.

3.2 Declare

This work is based on the initial assumption that any malicious behavior is implemented by a specific sequence of system calls. For this reason, we analyze system calls traces produced by a mobile application in response to some system events (listed in Table 2) to abstract a process model of the malware behavior in which the process activities are system calls. In this work, the Declare language and its supporting tools (ProM 6.1 tool and the Declare Miner plug-in) are used respectively to mine and to represent the malware behavioral model (defined as a set of system calls and their relationships) describing the behavior of a malware family. Declare is a declarative constraint language proposed by Pesic and van der Aalst and largely diffused in the Process Mining domain [34] that allows representing a process as a set of rules constraining all the events to be executed

Table 2 System events used to activate the malicious payload

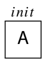
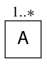
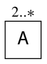
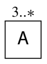
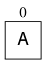
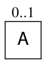
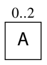
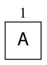
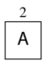
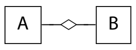
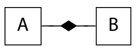
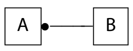
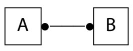
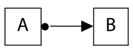
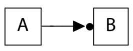
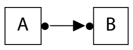
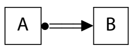
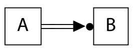
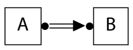
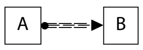
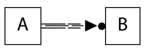
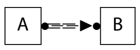
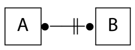
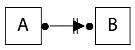
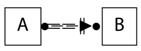
#	Event	Description
1	*BOOT_COMPLETED*	Able to catch the boot completed
2	*ACTION_ANSWER,NEW_OUTGOING_CALL*	Incoming and Outgoing call
3	*ACTION_POWER_CONNECTED*	Battery status in charging
4	*ACTION_POWER_DISCONNECTED*	Battery status discharging
5	*BATTERY_OKAY*	Battery full charged
6	*BATTERY_LOW*	Battery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

in a given order and implicitly describing all the possible workflows. Differently from procedural approaches which explicitly specify the interactions between process events (the produced models are “closed”, i.e., activities that are not explicitly specified in the model are forbidden) in declarative models all the workflows that do not violate the specified constraints are allowed (the produced models are “open”). For this reason, declarative approaches are suitable to represent complex processes with high flexibility [7]. Another advantage of using Declare is that it is a process modeling language more understandable for end-users and provides an executable and verifiable formal semantics. ProM 6.1 [44] is a tool supporting a wide variety of process mining techniques. In particular, the Declare Miner ProM 6.1 plug-in is able to extract from a set of log traces a model representing trends, patterns, and details related to an observed phenomenon. The obtained model is represented using the Declare language. In this work, the Declare Miner is used to extract, from a set of system call traces, the set of constraints holding between system calls in all the traces. According to this, the Declare Miner can extract the malware model and the application model from a set of traces collected during the application running in response to some system events.

Declare constraints can be seen as concrete instantiations of templates. A template is an abstract entity that defines parametrized classes of properties through a usable and simple graphical representation connected to a formal semantics based on the adoption of Linear Temporal Logic (LTL) formulas.

LTL formulas can be translated in non-deterministic Finite State Automatons (FSA) that represent all the traces satisfying the constraint. The temporal operators used to describe

Table 3 Graphical notation and LTL formalization of the declare templates

Template	Formalization	Notation
Init(A)	A	
Existence(A)	$\Diamond A$	
Existence2(A)	$\Diamond(A \wedge \bigcirc(\Diamond A))$	
Existence3(A)	$\Diamond(A \wedge \bigcirc(\Diamond(A \wedge \bigcirc(\Diamond A))))$	
Absence(A)	$\neg\Diamond A$	
Absence2(A)	$\neg\Diamond(A \wedge \bigcirc(\Diamond A))$	
Absence3(A)	$\neg\Diamond(A \wedge \bigcirc(\Diamond(A \wedge \bigcirc(\Diamond A))))$	
Exactly1(A)	$\Diamond A \wedge \neg\Diamond(A \wedge \bigcirc(\Diamond A))$	
Exactly2(A)	$\Diamond(A \wedge \bigcirc(\Diamond A)) \wedge$ $\neg\Diamond(A \wedge \bigcirc(\Diamond(A \wedge \bigcirc(\Diamond A))))$	
Choice(A,B)	$\Diamond A \vee \Diamond B$	
Exclusive Choice(A,B)	$(\Diamond A \vee \Diamond B) \wedge \neg(\Diamond A \wedge \Diamond B)$	
Responded Existence(A,B)	$\Diamond A \rightarrow \Diamond B$	
Co-Existence(A,B)	$\Diamond A \leftrightarrow \Diamond B$	
Response(A,B)	$\Box(A \rightarrow \Diamond B)$	
Precedence(A,B)	$\neg B \, \mathcal{W} \, A$	
Succession(A,B)	$\Box(A \rightarrow \Diamond B) \wedge (\neg B \, \mathcal{W} \, A)$	
Alternate Response(A,B)	$\Box(A \rightarrow \bigcirc(\neg A \mathcal{U} B))$	
Alternate Precedence(A,B)	$(\neg B \, \mathcal{W} \, A) \wedge \Box(B \rightarrow \bigcirc(\neg B \, \mathcal{W} \, A))$	
Alternate Succession(A,B)	$(\neg B \, \mathcal{W} \, A) \wedge \Box(B \rightarrow \bigcirc(\neg B \, \mathcal{W} \, A))$ $\wedge \Box(A \rightarrow \bigcirc(\neg A \mathcal{U} B))$	
Chain Response(A,B)	$\Box(A \rightarrow \bigcirc B)$	
Chain Precedence(A,B)	$\Box(\bigcirc B \rightarrow A)$	
Chain Succession(A,B)	$\Box(A \rightarrow \bigcirc B) \wedge \Box(\bigcirc B \rightarrow A)$	
Not Co-Existence(A,B)	$\Diamond A \rightarrow \neg\Diamond B$	
Not Succession(A,B)	$\Box(A \rightarrow \neg\Diamond B)$	
Not Chain Succession(A,B)	$\Box(A \rightarrow \neg\bigcirc B)$	

the semantics of the Declare templates are reported in Table 3. Let be, φ and ψ the LTL formulas, in the table, $\bigcirc\varphi$ is used to indicate that φ has to hold in the next position in a trace. $\Box\varphi$ means that φ is always in the subsequent positions in a trace. $\Diamond\varphi$ indicates that φ has to hold eventually in the subsequent positions in a trace. $\varphi\mathcal{U}\psi$ means that φ has to hold at least until ψ holds in a trace. Moreover, ψ must hold in a future or in the current position. Finally, $\varphi \, \mathcal{W} \, \psi$ means that φ has to hold in the subsequent positions at least until ψ holds. If ψ never holds, φ must hold everywhere.

Looking at the table, the *response* constraint $\Box(a \rightarrow \Diamond b)$ indicates that if *a occurs*, *b* must eventually *follow*. According to this, the *response* constraint is satisfied for traces such as $\mathbf{t}_1 = \langle a, a, b, c\rangle$, $\mathbf{t}_2 = \langle b, b, c, d\rangle$ and $\mathbf{t}_3 = \langle a, b, c, b\rangle$, but not for $\mathbf{t}_4 = \langle a, b, a, c\rangle$ because, in this case, the second instance of a is not followed by a b. Some consider-

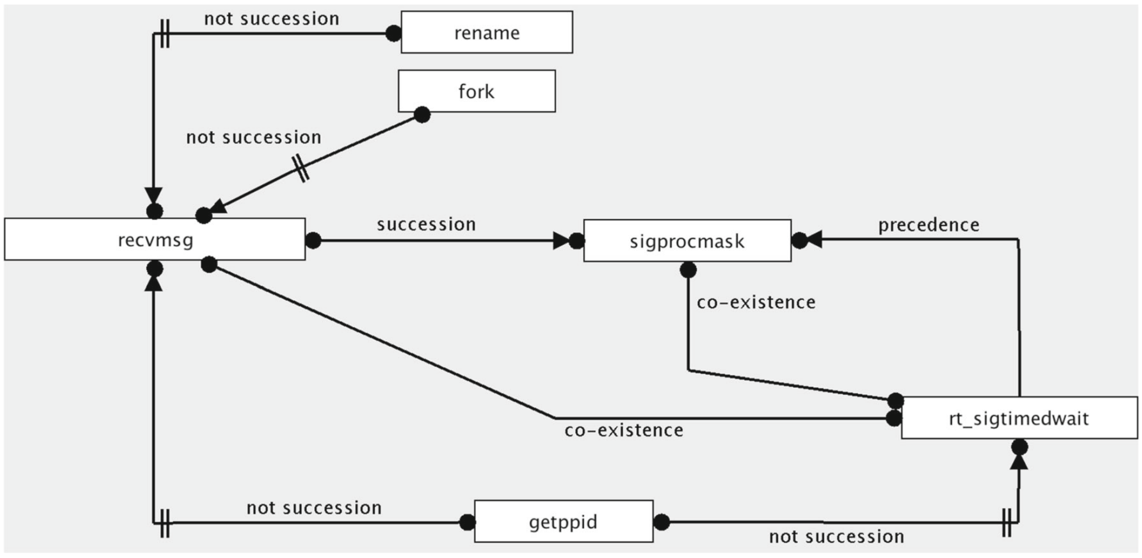


Fig. 1 Declare model derived from the traces of a malware application execution

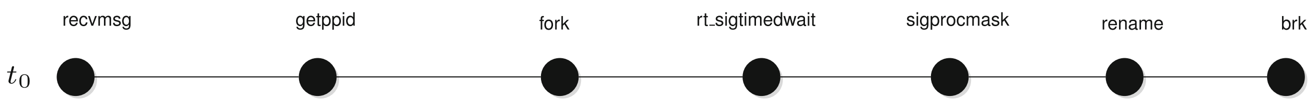


Fig. 2 An example trace log of the process in Fig. 1

ations apply to $\mathbf{t}_2$, where the response constraint is satisfied only because a never occurs. In this case, the constraint is called *vacuously satisfied* according to the notion introduced in [10]. Moreover, according to [10], a constraint is non-vacuously satisfied in a trace when it is activated in that trace. Moreover, a constraint is activated in a trace when its occurrence imposes some obligations. For example, for the *response* constraint $\Box(a \rightarrow \Diamond b)$, a is an activation because the execution of a forces b to be eventually executed. There are two kinds of constraint activation: the *fulfillment* and the *violation*. When every constraint activation in a trace leads to a fulfillment the trace is perfectly compliant with respect to a constraint. For example, the response constraint $\Box(a \rightarrow \Diamond b)$ in trace $\mathbf{t}_1$ is activated and fulfilled twice, whereas, in trace $\mathbf{t}_3$, it is activated and fulfilled only once. On the other hand, there are both a fulfillment and a violation of an activation of a constraint when a trace is not compliant with respect to this constraint (at least one activation leads to a violation). For example, looking at the trace $\mathbf{t}_4$, the response constraint $\Box(a \rightarrow \Diamond b)$ is activated twice: the first activation leads to a fulfillment (eventually b occurs) while the second activation leads to a violation (b does not occur subsequently). For clarity, we report in Fig. 1 an example of a declarative process obtained from a set traces of the execution of an application infected with the DroidKungFu4 malware. It consists of six activities (syscalls) and the constraints between them. The considered activities are listed and described in the following:

– *rename*: change the name or location of a file;
– *fork*: create a child process;
– *recvmsg*: receive a message from a socket;
– *sigprocmask*: examine and change blocked signals;
– *getppid*: get process identifier;
– *rt_sigtimedwait*: synchronously wait for queued signals.

For example, the *precedence* rule between *rt_sigtimewait* and *sigprocmask* means that whenever *sigprocmask* happens *rt_sigtimewait* happens before it. Figure 2 shows an excerpt from an execution trace for the process in Fig. 1. For example, according to the *precedence* constraint between *rt_sigtimewait* and *sigprocmask*, the *sigprocmask* event occurs one time and the *rt_sigtimewait* happens before it.

Finally, in Fig. 3, we report for completeness a finite state automaton (FSA) for the constraint *co-existence(rt_sigtimedwait, sigprocmask)* considering for simplicity a reference alphabet composed of three syscalls: *rt_sigtimedwait*, *sigprocmask*, and *fork*. The figure shows an automaton with the states *0*, *1*, *2*, *3*. The state *0* has an incoming arrow without a source state, which means that it is an initial state. Moreover, it is represented with a double border meaning that it is also an accepting state. The automaton remains in the initial state *0* until one of the syscalls *rt_sigtimedwait*, *sigprocmask* occurs. If the syscall *rt_sigtimedwait* occurs the automaton moves in the state *1* where it remains until the occurrence of syscall *sigprocmask*. Conversely, if in the initial state *0*, the syscall *sigprocmask*

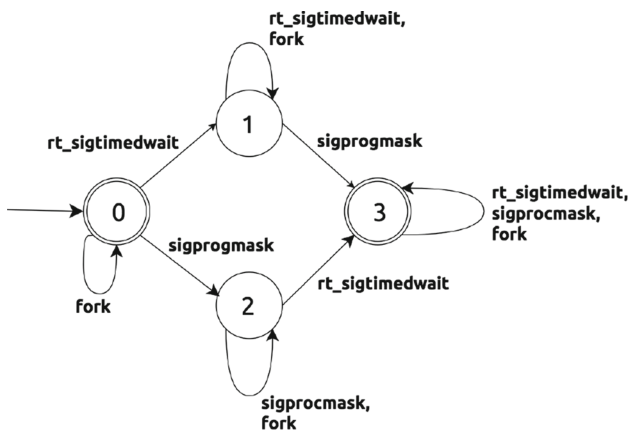


Fig. 3 An excerpt of the automaton obtained for the constraint *co-existence(rt_sigtimedwait, sigprocmask)* of the declare process depicted in Fig. 1

occurs, the automaton moves in state *2* where it remains until the occurrence of syscall *rt_sigtimedwait* occurs. Hence, from both states *1* and *2*, the automaton moves to the state *3* when the co-existence constraint is satisfied: in fact the transition to state *3* means that the trace contains both the syscalls *rt_sigtimedwait* and *sigprocmask*. In the proposed example, syscalls are always considered instantaneous. Moreover, transitions between states are reported with labeled directed arrows.

4 Approach

This section describes the proposed approach for Android malware detection and phylogeny tracking.

As mentioned in Sect. 1, the approach exploits process mining techniques to extract a characterization of a malware or trusted application from its system call traces. The produced characterizations can be profitably used to (i) recognize a malicious behavior at run-time, (ii) identify malware variants within the same family, and (iii) identify similarities across malware families. Moreover, the study of malware behavior can be used to support malware phylogeny by building "family trees" based on a similarity metric between malware characterizations.

The approach is based on two assumptions: (a) system call traces (or logs) captured from the execution of a mobile application can be used to characterize the application, including its malware behavior, if present; (b) system events reported in Table 2 and associated with the Android operating system represent the activation mechanism for malicious code present in Android mobile malware.

It is important to note that such characterization is not aimed at recovering a complete behavioral model of the application or malware payload since our aim is to obtain an effective fingerprint for malware detection. For this reason, we refer to system calls. The application code (especially malware payload in infected applications) is much more subject to obfuscation, whereas the sequence of run-time system calls executed to accomplish a (malicious) task is much more difficult to hide. This is the reason why our characterization is based on relationships among system calls as recovered from run-time traces. Starting from the above assumptions, the proposed approach mines system call traces produced by an application in response to system events listed in Table 2 with the aim to find recurring execution patterns and relationships between system calls characterizing the application or malware behavior. Such information represented as a Declare model [34] is named SEF—Syscalls Execution Fingerprint. SEFs can be compared with each other for malware detection and phylogeny analysis.

In the rest of the section, we describe (i) the formal definition of SEF, (ii) the malware detection process, and (iii) the process to build a phylogeny model for a family of malware.

4.1 Formal definition and notation for a SEF

We introduce here the formal definitions for the SEF associated to a mobile application and a malware family, and specify the notation we use to represent the two models.

Definition 1 (Declare model) Let be,

- T a set of system call traces;
- $C_{u_h} = \{S_A, R\}$ a unary constraint specifying a condition R, contained in Table 3, that holds for the occurrences of system call S_A over traces in T;
- $C_{b_k} = \{S_A, S_E, R\}$ a binary constraint specifying a relationship R, contained in Table 3, that holds between the occurrences of two system calls (i.e., S_A and S_E) over traces in T.

The Declare model associated to traces in T is defined as a set of h unary and k binary constraints:

$$D = \{C_{u_1}, \ldots, C_{u_h}, C_{b_1}, \ldots, C_{b_k}\}$$

□

Definition 2 (SEF of an application a) Let be,

- E the set of the n system events that can be sent to an application;
- t_j the execution trace (system calls trace) generated by the application a, in response to the event j;
- D_{a_j} the Declare model of the application a for the event j, mined from the set of traces $\{t_{j_1}, \ldots, t_{j_r}\}$ where r is the number of runs of the application.

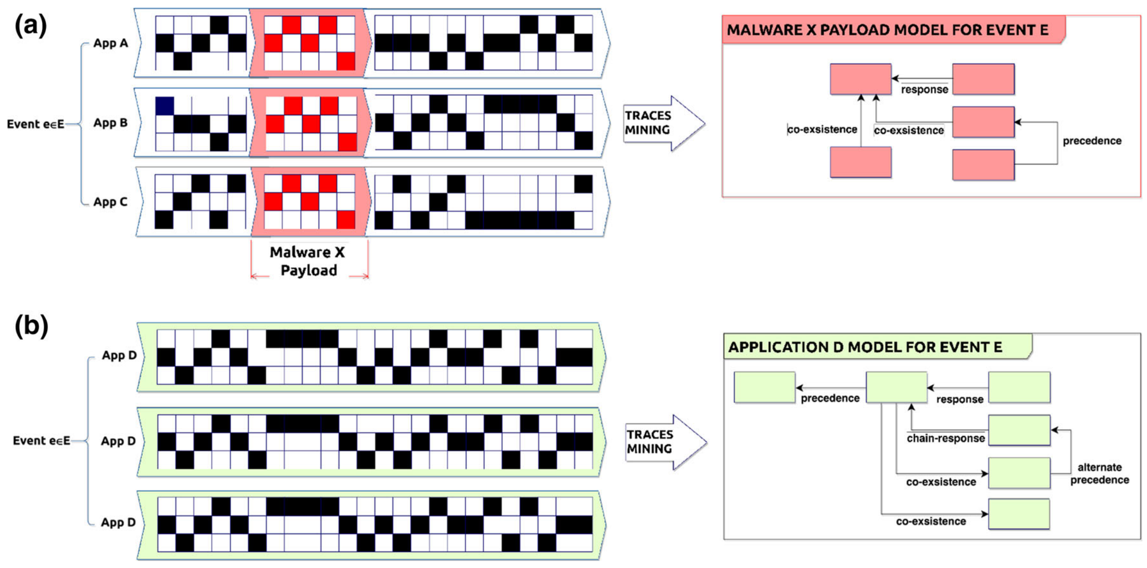


Fig. 4 The SEF construction for a malware family (**a**) and a single application (**b**)

The SEF$_a$ of the application a is the set of the declare models for all the system events, defined as follows:

$$\text{SEF}_a = \{D_{a_1}, \ldots, D_{a_n}\}$$

□

Definition 3 (SEF of a malware family M) Let be,

– A the set of m applications infected with the malware family M;
– E the set of the n system events that can be sent to an application;
– t_{ji} the execution trace generated by the i-th application of the set A in response to the j-th event of the set E;
– D_{M_j} the Declare model mined from the set of traces $\{t_{j1}, \ldots, t_{jm}\}$

We define:

$$\text{SEF}_M = \{D_{M_1}, \ldots, D_{M_n}\}$$

□

To better clarify the difference between the SEF of a malware family and the SEF of a single application we introduced Fig. 4. The upper side of the figure depicts the traces collected by different applications (i.e., app A, app B, app C) in response to the system event $e \in E$. Since the traces are generated by different applications, they contain different sequences of system calls (represented in the figure as black squares in different positions). Different applications infected with a given malware X share a common part corresponding to the malware X payload. Since the Declare mining process generates constraints that hold on each trace, the resulting model, in this case, will be a Declare model of the malware X payload in response to the event e. Conversely, the SEF of a single application D (infected or not) is generated as shown in the bottom of the Fig. 4. In this case, the traces are generated by the same application D. Hence, the common part corresponds to the entire trace. This means that the resulting model is a representation of the entire application in response to the event e. Please note that the number of traces used for Definitions 2 and 3 are different. The SEF of an application is recovered using, for each system event e, a set of r traces of the application. This is required since, in this case, we are mining the model of the entire application and hence several traces are needed to characterize its behavior using Declare rules. Conversely, we refer to the SEF of a malware as the model of the malicious payload that is common to a set of different applications infected with the same malware family. This means that, even with a single run for each application, we are able to mine the rules characterizing the common behavior among the applications, i.e. the malware payload.

4.2 Distance among SEFs

In order to define a distance for SEFs, we need to define a distance for Declare models. Given two Declare models D_i and D_j, we define the distance $d(D_i, D_j)$ between them as follows:

$$d(D_i, D_j) = \frac{\alpha(|\bigcup(D_i, D_j)| - |\bigcap(D_i, D_j)|)}{\alpha(|\bigcup(D_i, D_j)| - |\bigcap(D_i, D_j)|) + (1-\alpha)(|\bigcap(D_i, D_j)|)}$$

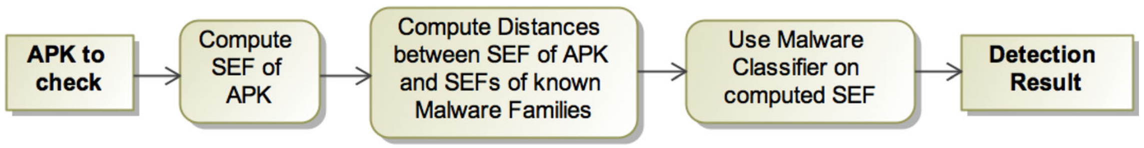

APK to check
Compute SEF of APK
Compute Distances between SEF of APK and SEFs of known Malware Families
Use Malware Classifier on computed SEF
Detection Result

Fig. 5 The malware detection approach in brief

where $\alpha \in [0.5, 1]$ is a parameter that allows to weigh differently constraints presence with respect to absence when evaluating the distance. In particular, for $\alpha = 0.5$ constraints presence and absence is equally weighted. The distance is proportional to the number of constraints that are not present in both models with respect to the total number of constraints. It represents a normalized measure of similarity among two models: when the two models have the same constraints the ratio is equal to zero, whereas when models have no common constraints the distance is one (i.e. maximum distance).

Starting from the definition of distance between Declare models reported above, we define the distance between two SEFs, SEF_A and SEF_B, as follows:

$$d(\mathrm{SEF}_A, \mathrm{SEF}_B) = \frac{\sum_{i=1}^{n} d(D_{A_{e_i}}, D_{B_{e_i}})}{n}$$

where $n = |E|$ is the cardinality of the considered set of system events.

To evaluate the distance between the SEF of a malware family and the SEF of a single application, the Declare rules that should be taken into account must be restricted to the set of rules that are present in the SEF of the malware. This is important since the SEF of a single application, even if infected with a malware includes the behavior of the original application in addition to that of the malicious code of the malware. The restriction step filters out, during distance evaluation, declare constraints between activities that are not present in the SEF model of the malware.

4.3 Malware detection

Figure 5 depicts how malware detection is accomplished. At the core of the approach is the definition of SEF reported in Sect. 4.1.

SEFs are used to characterize the behavior of an application to be checked. The matrix of distances between the SEF of the application and the SEFs of known malware families is calculated. This matrix is then provided as input to our malware classifier which, as a final result, indicates if the application is infected with a malware of a known family.

The malware classifier is based on the Weka data mining toolkit[2] and adopts different algorithms to classify the SEF of an application by looking at its distance from the SEFs of known malware families.

Several classification algorithms, available in the literature, are effective in performing the proposed classification [20]. As we will discuss in Sect. 5, in this work, six classification algorithms are used for generalizing and strengthening the internal validity of the obtained results.

In the rest of this subsection, we describe (i) the process to compute the SEF characterizing an application, (ii) the process to compute the SEF characterizing a family of malware, (iii) the process to build our malware classifier used for detection.

4.3.1 Computing of the SEF of an application

As more formally defined in Sect. 4.1, the SEF of an application (or APK[3]), is a collection of Declare models each of which characterizes the behavior of the application in response to one of the system events listed in Table 2. Such behavior is mined from syscall traces captured from the application in response to the specific system event. The process to compute the SEF associated with an application is depicted in Fig. 6. The main steps of the process are the following:

– *Syscall Traces Extraction* In this step syscalls traces generated by the APK in correspondence to the system events listed in Table 2 are captured and stored in a textual format. The APK is first installed and run on an Android device emulator prepared for the purpose. Then, a system event from the list in Table 2 is generated and sent to the emulator and the sequence of system calls made by the APK (syscall trace) in correspondence is gathered. We recall that system events listed in Table 2 are assumed to be the mechanism for a malware payload potentially present in an APK to get activated. Each event of the list is sent more than once in order to have a number of syscall traces from which to extract by factorization the behavior (in terms of syscalls) associated with it. Each time an event is sent, the APK is reinstalled and run in order to reduce as much as possible the influence of the APK state in the generated syscall trace. The step ends when the list of system events in Table 2 has been fully

[2] http://www.cs.waikato.ac.nz/ml/weka/.

[3] APK is the file format for an Android executable application. So, in the paper, we use the term APK as a synonym of Android application.

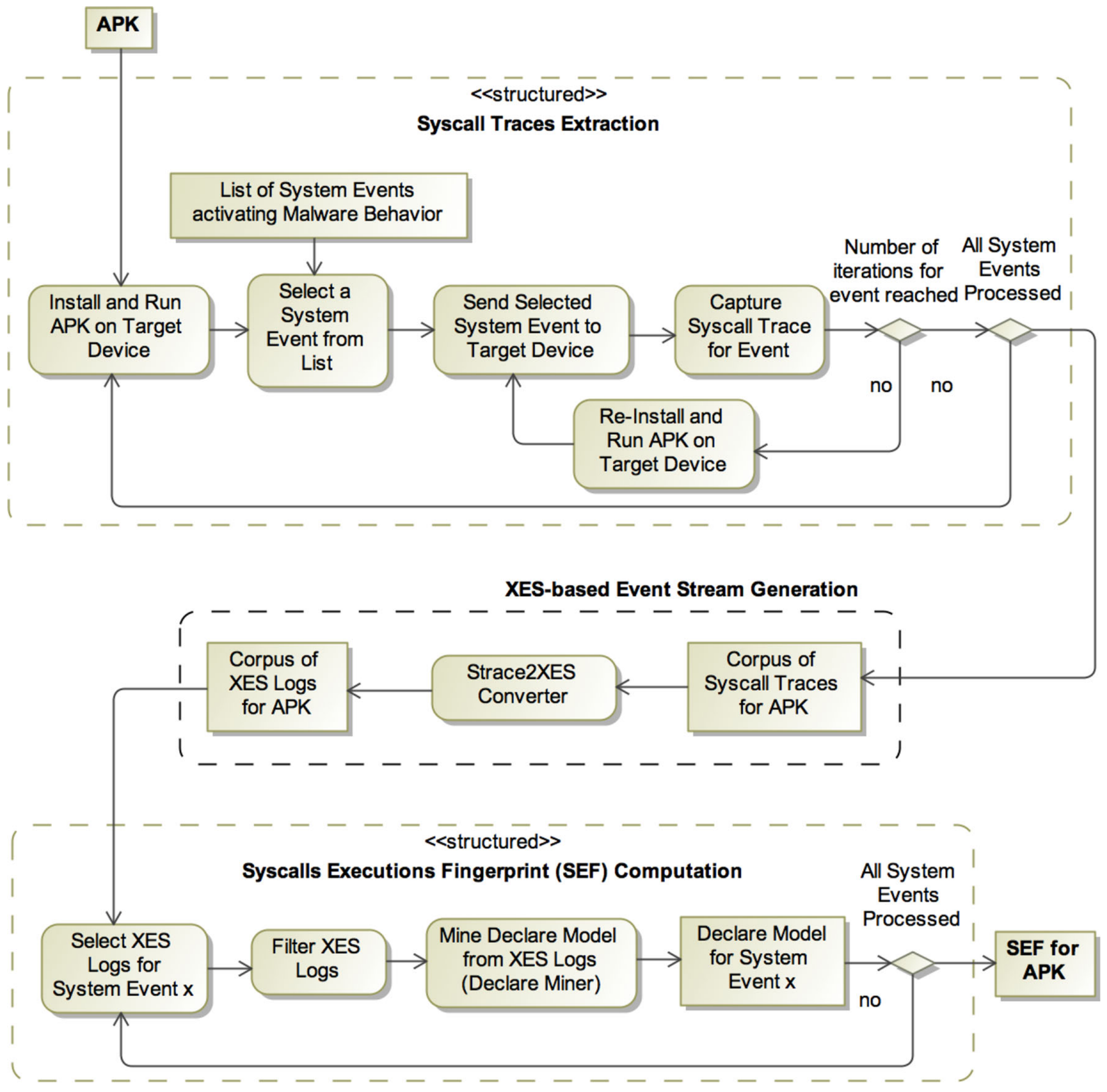

APK
<<structured>>
Syscall Traces Extraction
List of System Events activating Malware Behavior
Install and Run APK on Target Device
Select a System Event from List
Send Selected System Event to Target Device
Capture Syscall Trace for Event
Number of iterations for event reached
All System Events Processed
no
no
Re-Install and Run APK on Target Device
XES-based Event Stream Generation
Corpus of XES Logs for APK
Strace2XES Converter
Corpus of Syscall Traces for APK
<<structured>>
Syscalls Executions Fingerprint (SEF) Computation
All System Events Processed
Select XES Logs for System Event x
Filter XES Logs
Mine Declare Model from XES Logs (Declare Miner)
Declare Model for System Event x
no
SEF for APK

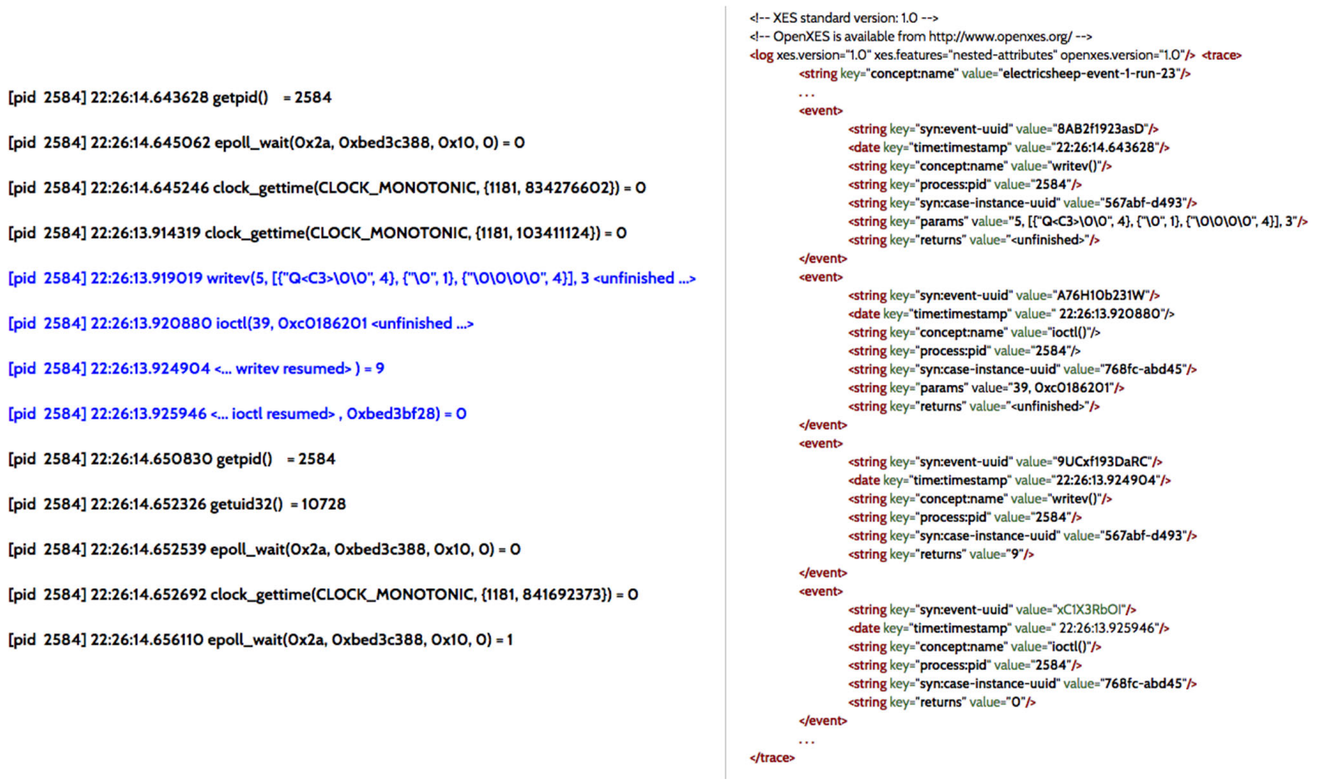
Fig. 6 The process for computing the SEF of an application

scanned. The step is handled by a set of shell scripts that perform the following actions:

1. start the target Android device emulator;
2. install and start the APK of the application on the device emulator;
3. wait until a stable state of the device is reached;
4. start the capture of syscall traces;
5. select one of the activation system events in Table 2
6. send the selected event to the application;
7. capture syscalls made by the application until a stable state is reached;
8. stop the syscall capture and save the captured syscall trace;
9. reinstall the APK and repeat the capturing for the selected event a fixed number of times (ten times in our evaluation);
10. select a new system event and repeat the steps above to capture syscall traces for this event;
11. repeat the step above until all system events in Table 2 have been considered.
12. stop the Android device and revert its disk to a clean baseline snapshot.

The script exploits the official Android emulator released by Google [4]. This emulator is able to simulate various Android smartphones, tablets and also wearable devices. It is able to provide almost all the capabilities of a real

[4] https://developer.android.com/studio/run/emulator.html.



```
[pid 2584] 22:26:14.643628 getpid()   = 2584

[pid 2584] 22:26:14.645062 epoll_wait(0x2a, 0xbed3c388, 0x10, 0) = 0

[pid 2584] 22:26:14.645246 clock_gettime(CLOCK_MONOTONIC, {1181, 834276602}) = 0

[pid 2584] 22:26:13.914319 clock_gettime(CLOCK_MONOTONIC, {1181, 103411124}) = 0

[pid 2584] 22:26:13.919019 writev(5, [{"Q<C3>\0\0", 4}, {"\0", 1}, {"\0\0\0\0", 4}], 3 <unfinished ...>

[pid 2584] 22:26:13.920880 ioctl(39, 0xc0186201 <unfinished ...>

[pid 2584] 22:26:13.924904 <... writev resumed> ) = 9

[pid 2584] 22:26:13.925946 <... ioctl resumed> , 0xbed3bf28) = 0

[pid 2584] 22:26:14.650830 getpid()   = 2584

[pid 2584] 22:26:14.652326 getuid32()  = 10728

[pid 2584] 22:26:14.652539 epoll_wait(0x2a, 0xbed3c388, 0x10, 0) = 0

[pid 2584] 22:26:14.652692 clock_gettime(CLOCK_MONOTONIC, {1181, 841692373}) = 0

[pid 2584] 22:26:14.656110 epoll_wait(0x2a, 0xbed3c388, 0x10, 0) = 1
```

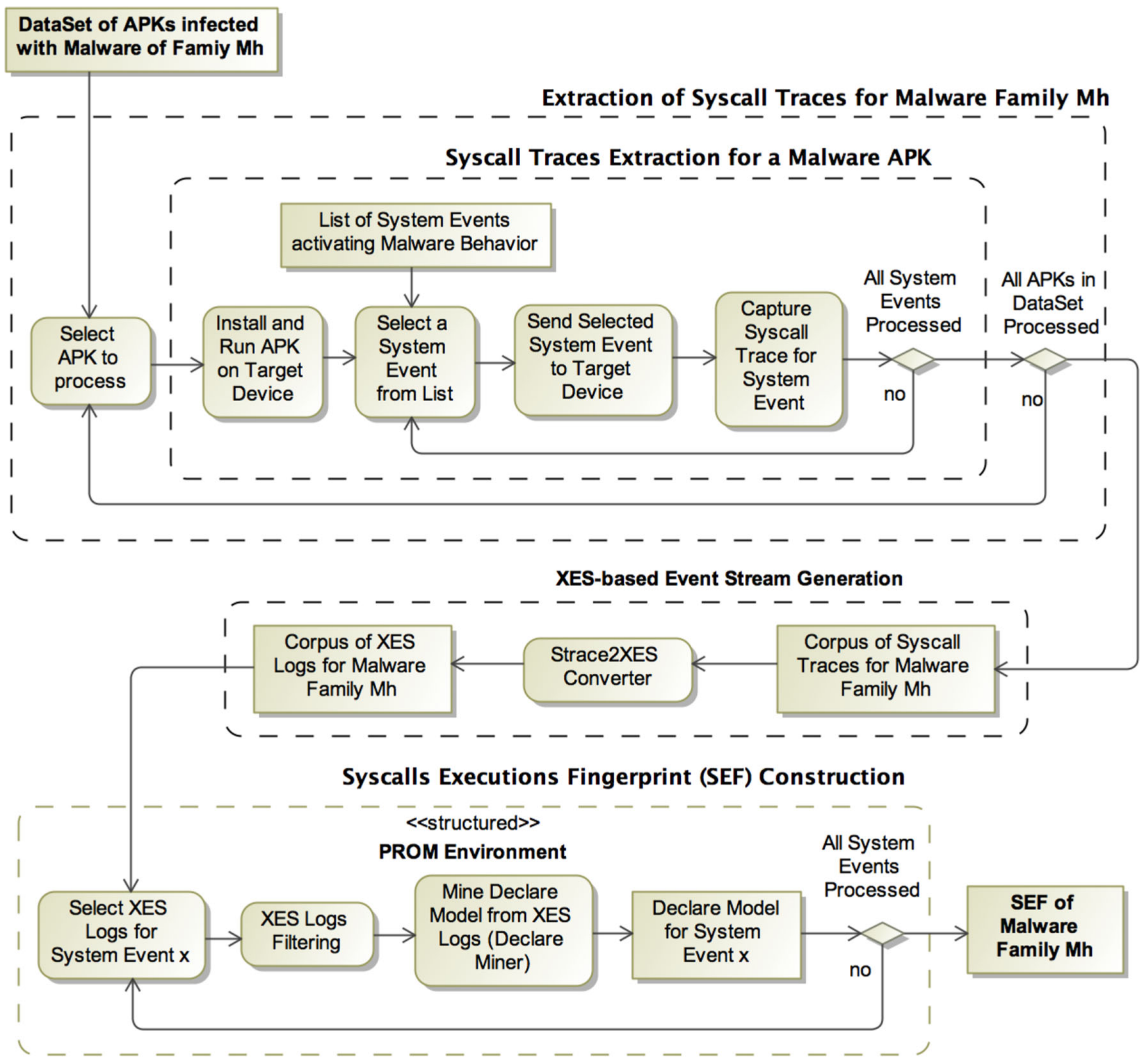
```
<!-- XES standard version: 1.0 -->
<!-- OpenXES is available from http://www.openxes.org/ -->
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0"/>  <trace>
        <string key="concept:name" value="electricsheep-event-1-run-23"/>
        . . .
        <event>
                <string key="syn:event-uuid" value="8AB2f1923asD"/>
                <date key="time:timestamp" value="22:26:14.643628"/>
                <string key="concept:name" value="writev()"/>
                <string key="process:pid" value="2584"/>
                <string key="syn:case-instance-uuid" value="567abf-d493"/>
                <string key="params" value="5, [{"Q<C3>\0\0", 4}, {"\0", 1}, {"\0\0\0\0", 4}], 3"/>
                <string key="returns" value="<unfinished>"/>
        </event>
        <event>
                <string key="syn:event-uuid" value="A76H10b231W"/>
                <date key="time:timestamp" value=" 22:26:13.920880"/>
                <string key="concept:name" value="ioctl()"/>
                <string key="process:pid" value="2584"/>
                <string key="syn:case-instance-uuid" value="768fc-abd45"/>
                <string key="params" value="39, 0xc0186201"/>
                <string key="returns" value="<unfinished>"/>
        </event>
        <event>
                <string key="syn:event-uuid" value="9UCxf193DaRC"/>
                <date key="time:timestamp" value="22:26:13.924904"/>
                <string key="concept:name" value="writev()"/>
                <string key="process:pid" value="2584"/>
                <string key="syn:case-instance-uuid" value="567abf-d493"/>
                <string key="returns" value="9"/>
        </event>
        <event>
                <string key="syn:event-uuid" value="xC1X3RbOI"/>
                <date key="time:timestamp" value=" 22:26:13.925946"/>
                <string key="concept:name" value="ioctl()"/>
                <string key="process:pid" value="2584"/>
                <string key="syn:case-instance-uuid" value="768fc-abd45"/>
                <string key="returns" value="0"/>
        </event>
        . . .
</trace>
```

Fig. 7 An excerpt of a syscall trace log (left side) and the corresponding XES (right side)

Android device (for instance, it simulates phone calls, text messages, localization service and different network speeds). After the device is started (step 1), the script installs and starts the application (step 2) and waits for a stable state (when in step 3, *epoll_wait* is executed and the application waits for user input or a system event to occur). An important step of this script deals with device application system event handling (each system event is related to an application handler). When an event is sent to the application (step 6), the handler gets executed and produces a stream of syscall in the strace log. The script is responsible to capture the syscall stream from the event sending up to the exit from the application handler, when a stable state is reached (step 8). Despite achieving a stable state, it is possible that syscalls are not related to the malicious payload (since they belong to the specific application). To mitigate this risk and be able to filter out such syscalls, for each application under analysis, we consider the syscall traces collected from ten different runs. From these ten executions, only the common part of the strace log is extracted.

– *XES-based Event Stream Generation* In this step syscall traces collected and saved in textual format in the previous step are converted into an eXtensible Event Stream (XES) log format [43], an XML-based standard for event logs[5]. This conversion step is required as the Declare Miner generates Declare models taking as input process logs encoded in the XES format. Syscall traces extracted from the operating system are in a textual format. An excerpt of a syscall trace is shown in the left side of Fig. 7), while the corresponding XES event stream obtained after the conversion step is reported in right side of Fig. 7. This is accomplished by the Strace2XES converter, a tool implemented as an Eclipse application. During this conversion, only useful information available in the trace is kept. This includes attributes associated with the entire trace (e.g., the id of the application from which the trace has been generated) and attributes associated to a single system call occurrence (e.g., the executed system call, its timestamp, a list of arguments, if present, and the id of the process requesting the call). This information is useful during the following constraint mining step of the process to correlate events and extract a Declare model from syscall traces. For example, consider the trace excerpt of Fig. 7, $t_0 =< writev, ioctl, writev, ioctl >$ (highlighted in blue in the left side of the figure) and the constraint *response(writev,ioctl)*: it is impossible to determine which of the two instances of the syscall $ioctl$ should be associated to the first occurrence

[5] Visit http://www.xes-standard.org/.



DataSet of APKs infected with Malware of Famiy Mh
Extraction of Syscall Traces for Malware Family Mh
Syscall Traces Extraction for a Malware APK
List of System Events activating Malware Behavior
Select APK to process
Install and Run APK on Target Device
Select a System Event from List
Send Selected System Event to Target Device
Capture Syscall Trace for System Event
All System Events Processed
no
All APKs in DataSet Processed
no
XES-based Event Stream Generation
Corpus of XES Logs for Malware Family Mh
Strace2XES Converter
Corpus of Syscall Traces for Malware Family Mh
Syscalls Executions Fingerprint (SEF) Construction
<<structured>>
PROM Environment
Select XES Logs for System Event x
XES Logs Filtering
Mine Declare Model from XES Logs (Declare Miner)
Declare Model for System Event x
All System Events Processed
no
SEF of Malware Family Mh

Fig. 8 The process for computing the SEF for a malware family

of the syscall *writev* in the excerpt. Such ambiguity prevents a correct evaluation of constraints in terms of satisfaction/violation. For data correlation, we adopted a reference-based correlation approach similar to the one proposed in [6,9]: two events correlate if they satisfy a correlation function that depends on a set of attributes of the first event (the identifier attributes) and on a set of attributes of the second event (the reference attributes). In our context, the identifier and the reference attributes are the process id (pid) and the event timestamp, respectively, and they coincide. For example, in Fig. 7, the second *writev* syscall is considered the same instance as the first one since its identifier and reference attributes are the same and the same happens for *ioctl*.

The correlation function requires that the pids of the two events are equal, whereas the timestamp of the second event is greater than the timestamp of the first one. This allows the mining algorithm to keep the identification of constraints separate for each process execution of a given application and avoids the detection of fake rules mined from the wrong correlation of system call events belonging to different processes.

– *Syscalls Execution Fingerprints (SEF) Computation* In this step, the SEF associated to the application provided as input to the process is computed. For each system event in Table 2 the associated XES logs are selected. The logs of each set are firstly processed in order to filter out useless ones, e.g., those shorter than a given threshold (calculated by evaluating the Gaussian distribution of the logs sizes and filtering out all the logs that are outside the 80th percentiles).

Such logs are mined using the Declare Miner, a plug-in for the ProM Process Mining Toolkit, in order to obtain a Declare model from them. The collection of the so obtained Declare models represents the SEF of the APK and the characterization that the approach uses for malware detection. Each model consists of a set of constraint rules expressed in the Declare language [34] that describes relationships among system calls holding in all the traces.

4.3.2 Computing of the SEF of a family of malware

As formally defined in Sect. 4.1, the SEF of a family of malware is a set of Declare models, each of which characterizes the behavior of the malware family in correspondence to one of the system events in Table 2. Each of these models is mined from a dataset of syscall traces generated from different applications infected with a specific malware family when 'stimulated' with one of the system events reported in Table 2. Since each model contains the constraints among system calls that hold in all the captured traces (no matter of the application they were captured from), it can be regarded as a representation of the malicious payload behavior in correspondence to a particular system event of the list.

Indeed, the behavior of the malicious payload is expected to be the only shared behavior among the variety of infected applications used to compute the SEF of the malware family. It is worth noting that, while the SEF of a malware family describes only the behavior of the malicious payload, the SEF computed for an infected application will include constraints deriving from both the application behavior and the malicious payload.

Figure 8 shows the process for computing the SEF associated to a malware family. The process looks similar to that for computing the SEF for a single application.

In this case, the mined Declare model is derived from a set of different applications infected with the same malware family. Each application of the set has only one part in common with the others: the malware payload. This means that syscall traces derived from such applications share only the part of the behavior associated with such malware payload. As a consequence, the mining process generates a model that retains the behavior of the malware payload (for which the Declare constraints support tends to be high) and discards the one that is specific to each application (for which the Declare constraints support is very low)[6].

The main steps of the process are the following:

– *Extraction of Syscall Traces for the Malware Family* In this step, a syscall trace is collected for each of the APK in the dataset of APKs provided as input for each of the system event in Table 2. All the APKs in the dataset must be verified in advance to be infected with a malware of the family Mh under analysis. In detail, this step is executed in a similar way to the step described in Sect. 4.3.1 for collecting syscall traces for an application, except that (i) it is repeated for several APKs, and (ii) for each APK and each system event in Table 2 only one trace is collected.

– *XES-based Event Stream Generation* Similarly to what happens in the process for computing the SEF of an application, in this step, the syscall traces collected and saved in textual format in the previous step are filtered and converted into an XES log format.

– *Syscalls Execution Fingerprint (SEF) construction* XES logs produced from the previous step groups together syscall traces associated with a given system event. Each group of XES logs is parsed using the Declare Miner. The so generated Declare model consists of a set of constraints expressed in the Declare language that describe the relationships between system calls that hold in all the analyzed traces. As such, the model describes the com-

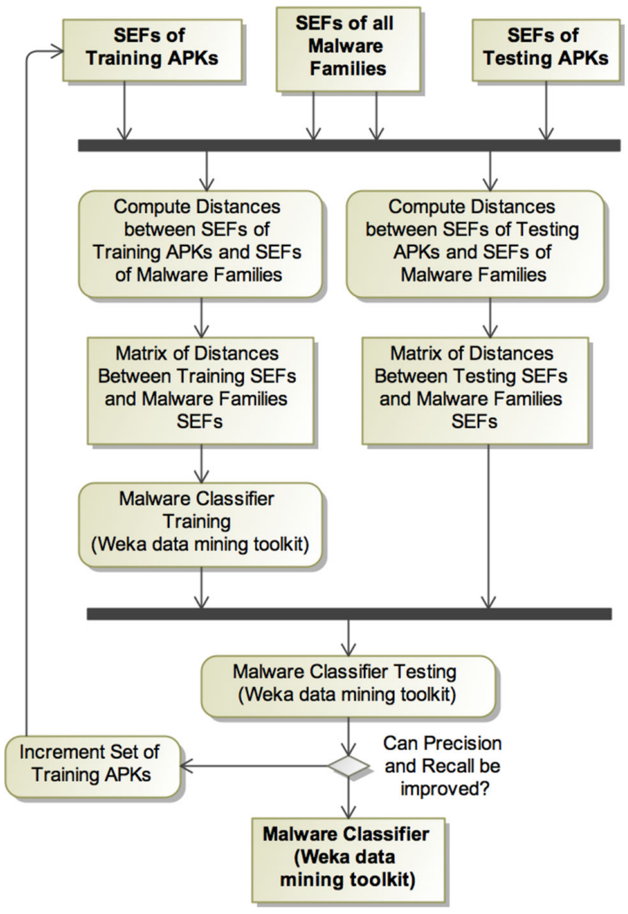

SEFs of Training APKs
SEFs of all Malware Families
SEFs of Testing APKs
Compute Distances between SEFs of Training APKs and SEFs of Malware Families
Compute Distances between SEFs of Testing APKs and SEFs of Malware Families
Matrix of Distances Between Training SEFs and Malware Families SEFs
Matrix of Distances Between Testing SEFs and Malware Families SEFs
Malware Classifier Training (Weka data mining toolkit)
Malware Classifier Testing (Weka data mining toolkit)
Increment Set of Training APKs
Can Precision and Recall be improved?
Malware Classifier (Weka data mining toolkit)

Fig. 9 Building of the malware classifier

6 There are two notions of support that can be defined by Declare language: one based on the percentage of constraints activations that leads to a fulfillment (called event-based constraint support) and the other based on the percentage of traces in which the constraint is satisfied (trace-based support). In our context, we considered trace-based support since we are interested in mining the behavior that is shared by all the traces (having assumed that, for different applications, such behavior models the malicious payload).

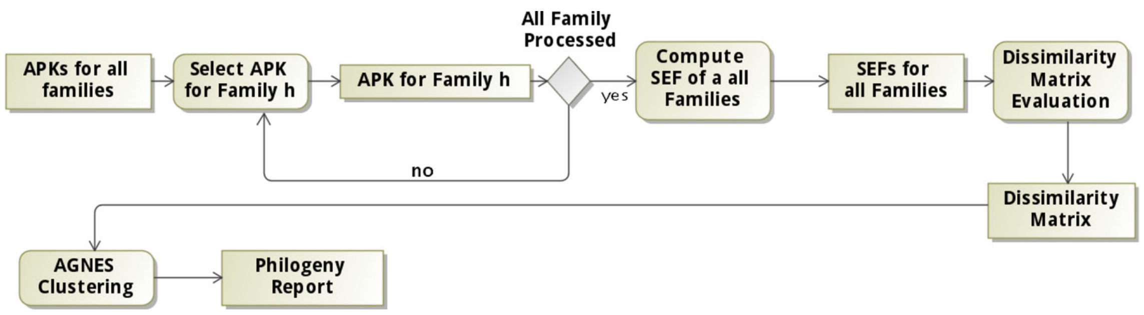


Fig. 10 Malware phylogeny process

mon behavior between the set of applications, which is expected to be the behavior of the malicious code associated with the malware. The collection of Declare models associated with the different system events in Table 2 are defined to be the SEF of the malware family. This model represents a characterization of the behavior of the malware family and hence is exploitable for detection and phylogeny tracking tasks.

4.3.3 Building of the malware classifier

Our malware classification is performed using a Weka data mining toolkit. In particular, Fig. 9 shows the process for training it. SEFs are computed for a dataset of training APKs, for the all known malware families, and for a test set of APKs. Then, the training process is started and repeated with an increased number of training APKs until the best values possible for precision and recall are obtained using the classifier over the testing APKs. The training is accomplished calculating the matrices of distances between the SEFs of the known malware families and the SEFs of the training APKs and the testing APKs. Several classification algorithms are used over these matrices.

4.4 Phylogeny tracking approach

The proposed approach for malware phylogeny tracking is reported in Fig. 10. The approach is based on the computation of a malware phylogeny model according to the following main activities:

- *Compute SEFs of a set of malware families* This activity shares the SEF construction process already shown in Sect. 8. In this case, the built SEFs are used to evaluate a dissimilarity matrix to be used in the subsequent clustering step.
- *Dissimilarity matrix evaluation* This activity takes as input the SEF models for the set of malware families and constructs a symmetric dissimilarity matrix in which each entry i, j reports the dissimilarity between the SEF model of the family i and that of the family j. It is based upon the distance definition given in Sect. 4.2.
- *Clustering* The final activity is aimed to recover a phylogeny model by applying the Hierarchical Agglomerative Clustering (HAC) algorithm described in [22] over the dissimilarity matrix computed in the previous step. This algorithm is commonly used in phylogeny model construction for both malware and biologic phylogeny classification and, in our context, provides very good results with the best trade-off with respect to performances. The resulting phylogeny model, however, is unable to represent multiple inheritances. This means that the lineage is always a linear path. For this reason, when a malware derives from several parents, the model allows only to select the closest parent.

5 Evaluation

The effectiveness and efficiency of the proposed approach have been evaluated using a dataset of 1200 malicious and trusted applications belonging to ten malware families[7]. The evaluation starts from the collected syscall traces structured in three sets (training, test and trusted sets) as specified in Sect. 4.

5.1 Dataset construction

The dataset used to empirically evaluate our approach includes malware and trusted applications that were collected as follows. Malware applications characterized by different nature and malicious intents (wiretapping, selling user information, advertisement, spam, stealing user credentials, ransom) have been downloaded from both Genoma [54] and Drebin [4] datasets. Trusted applications are the most downloaded from the Google Play store from July 2012 to September 2014 for different categories (call, contacts, education, entertainment, travel, Internet, lifestyle, news,

[7] An excerpt is available at https://github.com/mlbresearch/syscall-traces-dataset.

Table 4 The malware families of the dataset

Family	IT	#DS	#AS
Adrd	r	91	78
DroidDream	r	81	81
DroidKungFu1	r	34	34
DroidKungFu2	r	30	30
DroidKungFu3	r	304	67
DroidKungFu4	r	97	66
Fakeinstaller	s	925	101
Geinimi	r	92	42
Kmin	s	147	112
Opfake	r	613	428
Trusted	r	200	200

productivity, utilities, business, communication, messaging, fun, health and personalization). All the applications labeled in the dataset as trusted or malware were checked by the dataset producers. Moreover, trusted applications were also checked by Google Bouncer [33]. Additionally, we also performed a quality analysis to confirm that all the applications that were considered as trusted did not contain any malicious code and that all the applications considered as infected with a given malware family were indeed infected with that malware family. This quality analysis was performed by using 57 antimalware (running on VirusTotal service [45]) to check all the applications of the dataset. For malware applications, we filtered out all the applications that were not recognized as infected with the studied malware by at least five antimalware over the set of 57 ones. Similarly, we excluded from the dataset all the applications that were labeled as trusted but were not recognized as trusted by all the 57 antimalware. In this way, we strongly reduced the possibility to have wrong labeled applications in the considered dataset. The training set was used to generate the SEF model for each malware family. These families were obtained by grouping malware applications sharing common characteristics (i.e., payload installation, type of attack, and set of system events that trigger the malicious payload [54]). The list of the malware families considered in this study is reported in Table 4. We classified the applications contained in both Genoma and Debrin datasets in two groups according to the number of samples. We performed a random selection of the applications in each group. The test set is used to verify that the malicious applications' behavior of a given malware family is correctly represented by the SEF model obtained from the training set. In particular, the correctness is verified by analyzing the distribution of the distances between SEFs of applications belonging to the same malware family of the model itself. The distance has been evaluated by fixing α to 0.5. This allows to consider constraints presence and absence in syscall traces as equally weighted. Table 4 provides some descriptive statistics on our evaluation dataset specifying for each of the considered malware family the installation type (IT) (repackaging (r) or standalone (s)), the number of downloaded samples (#DS), and the number of analyzed samples (#AS) recognized as malware during the quality analysis. Following the process described in Sect. 4, the detection process is based on the evaluation and analysis of the distributions of intra-family distances, whereas the phylogeny model construction requires an inter-family similarity analysis by comparing malware family models with each other.

5.2 Intra-family distances distributions analysis

The distribution of intra-family distances for the families DroidKungFu$_{1-2}$ and Geinimi and for the three most discriminating events is shown in Fig. 11. We consider as discriminating all the system events for which we obtain statistically separated distance distribution. The results highlight how the SEF model can be used as a fingerprint of the malicious behavior, effectively. Looking at the figure, we can observe that for the described three events (BOOT_COMPLETED, BATTERY_LOW, and INPUT_MEDIA_CHANGED), the SEF for DroidKungFu$_1$ and DroidKungFu$_2$ is quite discriminating. Moreover, the boxplots show that for these events the median values of the distance distributions are well discerned: the value is 0.2 for the infected applications and 0.7 for the trusted applications. Even if the results on test applications are less consistent than on trusted distances, the medians value never overlap. Looking at the Geinimi family, the BOOT and SMS events are shown to be discriminating. This result is also confirmed by the first two boxplots of the third column of the matrix represented in Fig. 12. In this figure, the last boxplots show that for the SYS event nothing can be claimed since both medians overlap and the distance between the test and trusted models is close to the maximum. Finally, the results obtained for the other analyzed families are consistent and comparable, showing that they are well discriminated by at least one system event. This can be observed from the boxplots in Fig. 13. This figure highlights how different events show a different discriminating power (the figure shows only the most discriminating event across the considered families).

5.3 Inter-family similarity: distances among family models

In Fig. 14, the dissimilarity matrices for the set of considered families and the three most discriminating system events are reported. Each discriminating system event is used to relate similar families based on their behavior, that is expressed by the constraints among system calls executed in response to that event. The malware lineage can be determined by the

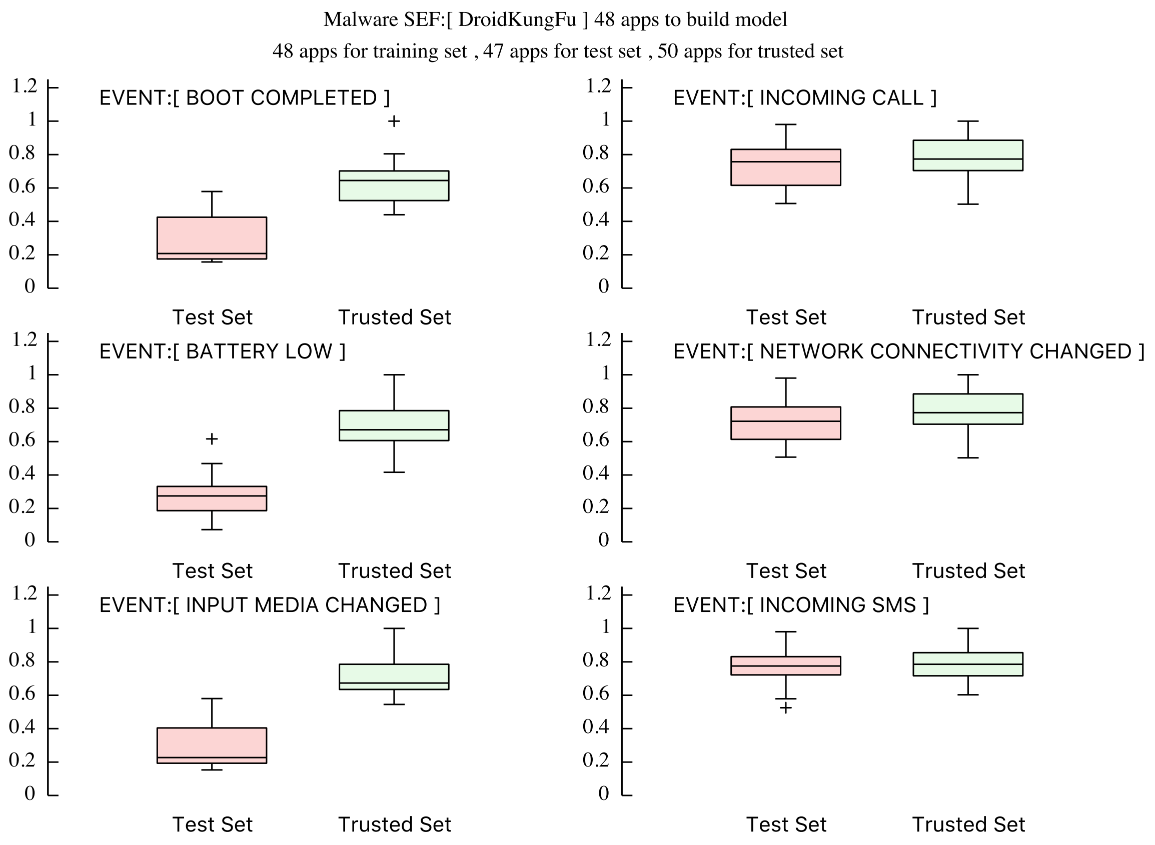


Fig. 11 The distance distributions among SEFs for the DroidKungFu3 malware over six events

validation of the cluster recovered from the SEF distance matrices using the malware discovery dates. The clusters of the dissimilarity matrices of the discriminating events are obtained using the following weights:

$$w_{i,j} = \frac{\#discriminative\ events}{\#total\ events}$$

if the event j is discriminating for the family i, while $w_{i,j} = 0$ if it is not.

The most discriminating event is the BATT event. It is also interesting to observe that in response to the BOOT event, the system calls execution relationships show that DroidKungFu$_4$ is more similar to DroidKungFu$_1$ than to DroidKungFu$_3$. With regards to the other families, the SEFs show that they are quite different (the agglomerative clustering step puts them together behind the cut). Figure 15 shows the dendrogram derived from the clustering step applied to the weighted dissimilarity matrix in Fig. 14. The dendrogram indicates that the three most discriminating events (BOOT_COMPLETED, BATTERY_LOW, and INPUT_MEDIA_CHANGED events) provide effective results by grouping together all the variants of DroidKungFu.

Moreover, looking at the dendrogram, we can observe that GinMaster and DroidDream are more similar in terms of system calls execution relationships with respect to Geinimi, even if the values are not discriminating.

5.4 Classification results

The classification has been performed using the six classification algorithms listed in the second column of Table 5. Two kinds of classification are executed: the first is based on a single binary classifier discriminating malware and trusted applications; the second uses a binary classifier for each family and identifies the particular family of a malicious application. The results obtained for these two kinds of classification are summarized in Table 5. The first column of the table reports the malware family (including a single "All families" classifier trained with malware samples from all the families included in the dataset). The second column lists the adopted classification algorithms for each family, while the four remaining columns report the classification results. The quality of the classification is evaluated by calculating precision (column three) and recall (column four) based on the following definitions. Let be:

- T_P: # of true positives (# of correctly classified occurrences, i.e., applications classified as malware, being actually malware),

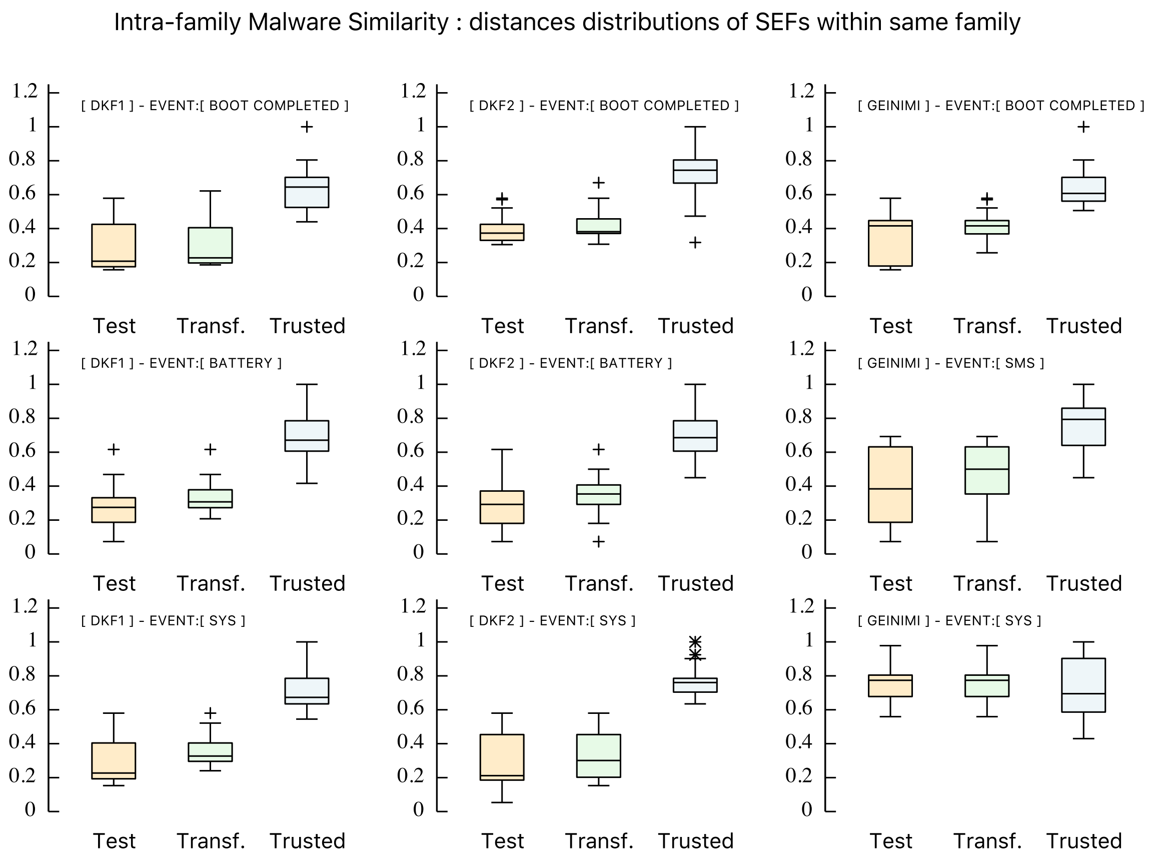


Fig. 12 The distances distributions among SEFs models for three malware families

- F_P: # of false positives (# of incorrectly classified occurrences, i.e., applications classified as malware, while they are trusted),
- F_N: # of false negatives (# of not classified occurrences, i.e., applications classified as trusted, while they are malware).

Precision is defined as the ratio of correctly classified occurrences to all occurrences provided by the algorithms and is given by:

$$\text{Precision} = \frac{T_P}{T_P + F_P}$$

Recall is the ratio of correctly classified occurrences to all correct occurrences and is given by:

$$\text{Recall} = \frac{T_P}{T_P + F_N}$$

The Gold Standard (GS) used as reference is the set of all correctly classified occurrences. Finally, the ROC Area [35] (column five) is evaluated. It is the area under the ROC curve (AUC) and is defined as the probability that a randomly chosen positive occurrence is ranked above a randomly chosen negative one.

Observing the first row of the table related to a "catch-all" classifier for "All families", Precision and Recall values are 0.903 and 0.938 (with the NBTree algorithm), respectively. The table also highlights that there is at least one classification algorithm for each considered malware family giving values of Precision and Recall greater than 0.85. Moreover, the ROC value shows that the probability of scoring malware applications higher than trusted ones is 0.9 (with the NBTree algorithm) and for all the malware families there is at least one classification algorithm resulting in a ROC value greater than 0.88. The obtained results are very promising if compared to similar approaches available in the literature (the most relevant are discussed in Sect. 2). Moreover, our approach is better suited to perform malware family identification. This is discussed in Sect. 6.

6 Robustness analysis

In order to demonstrate the effectiveness of our method in malware identification, we applied a set of well-known code transformations techniques [12,37,51] to the applications in

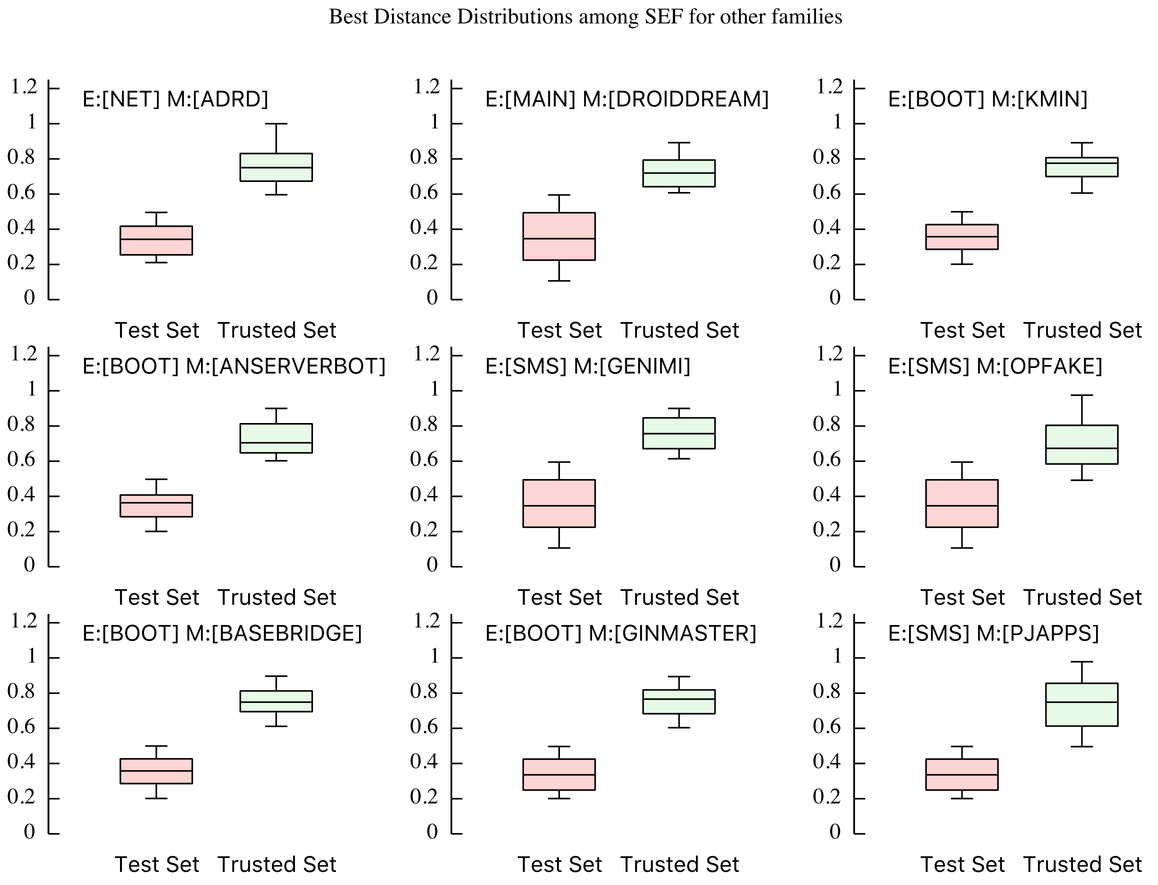
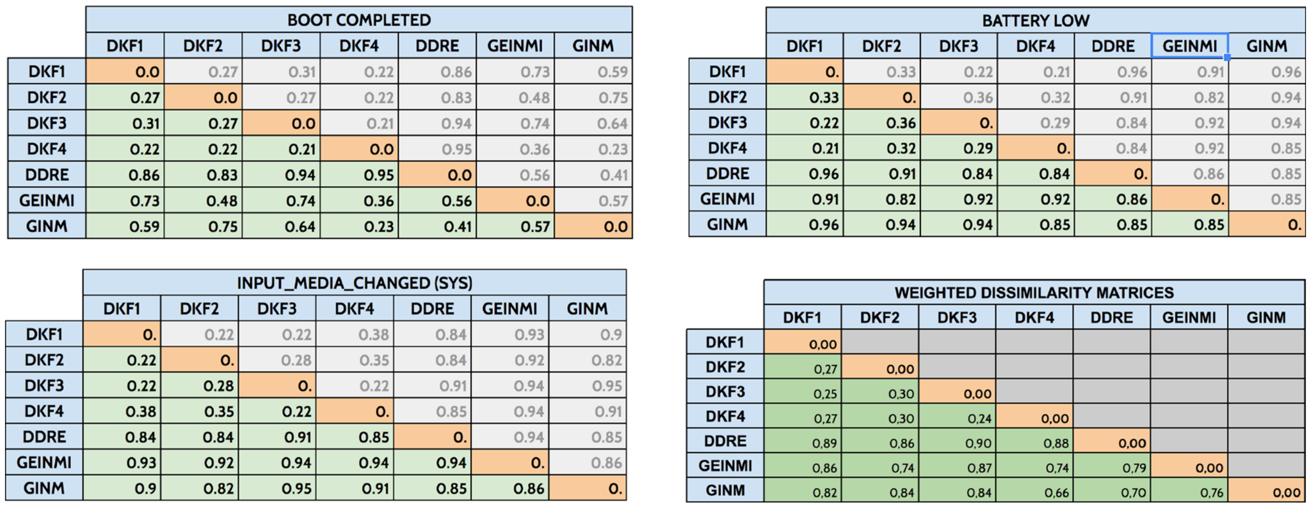


Fig. 13 The distance distributions among SEFs according to the most discriminating system event for each family



	BOOT COMPLETED						
	DKF1	DKF2	DKF3	DKF4	DDRE	GEINMI	GINM
DKF1	0.0	0.27	0.31	0.22	0.86	0.73	0.59
DKF2	0.27	0.0	0.27	0.22	0.83	0.48	0.75
DKF3	0.31	0.27	0.0	0.21	0.94	0.74	0.64
DKF4	0.22	0.22	0.21	0.0	0.95	0.36	0.23
DDRE	0.86	0.83	0.94	0.95	0.0	0.56	0.41
GEINMI	0.73	0.48	0.74	0.36	0.56	0.0	0.57
GINM	0.59	0.75	0.64	0.23	0.41	0.57	0.0

	BATTERY LOW						
	DKF1	DKF2	DKF3	DKF4	DDRE	GEINMI	GINM
DKF1	0.	0.33	0.22	0.21	0.96	0.91	0.96
DKF2	0.33	0.	0.36	0.32	0.91	0.82	0.94
DKF3	0.22	0.36	0.	0.29	0.84	0.92	0.94
DKF4	0.21	0.32	0.29	0.	0.84	0.92	0.85
DDRE	0.96	0.91	0.84	0.84	0.	0.86	0.85
GEINMI	0.91	0.82	0.92	0.92	0.86	0.	0.85
GINM	0.96	0.94	0.94	0.85	0.85	0.85	0.

	INPUT_MEDIA_CHANGED (SYS)						
	DKF1	DKF2	DKF3	DKF4	DDRE	GEINMI	GINM
DKF1	0.	0.22	0.22	0.38	0.84	0.93	0.9
DKF2	0.22	0.	0.28	0.35	0.84	0.92	0.82
DKF3	0.22	0.28	0.	0.22	0.91	0.94	0.95
DKF4	0.38	0.35	0.22	0.	0.85	0.94	0.91
DDRE	0.84	0.84	0.91	0.85	0.	0.94	0.85
GEINMI	0.93	0.92	0.94	0.94	0.94	0.	0.86
GINM	0.9	0.82	0.95	0.91	0.85	0.86	0.

	WEIGHTED DISSIMILARITY MATRICES						
	DKF1	DKF2	DKF3	DKF4	DDRE	GEINMI	GINM
DKF1	0,00						
DKF2	0,27	0,00					
DKF3	0,25	0,30	0,00				
DKF4	0,27	0,30	0,24	0,00			
DDRE	0,89	0,86	0,90	0,88	0,00		
GEINMI	0,86	0,74	0,87	0,74	0,79	0,00	
GINM	0,82	0,84	0,84	0,66	0,70	0,76	0,00

Fig. 14 The dissimilarity matrices associated to each of the events of the considered SEFs and the resulting weighted dissimilarity matrix

our dataset. Such techniques are used by malware writers to evade the signature-based detection approaches adopted by current antimalware.

In particular, in our experiment, we used the following code 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

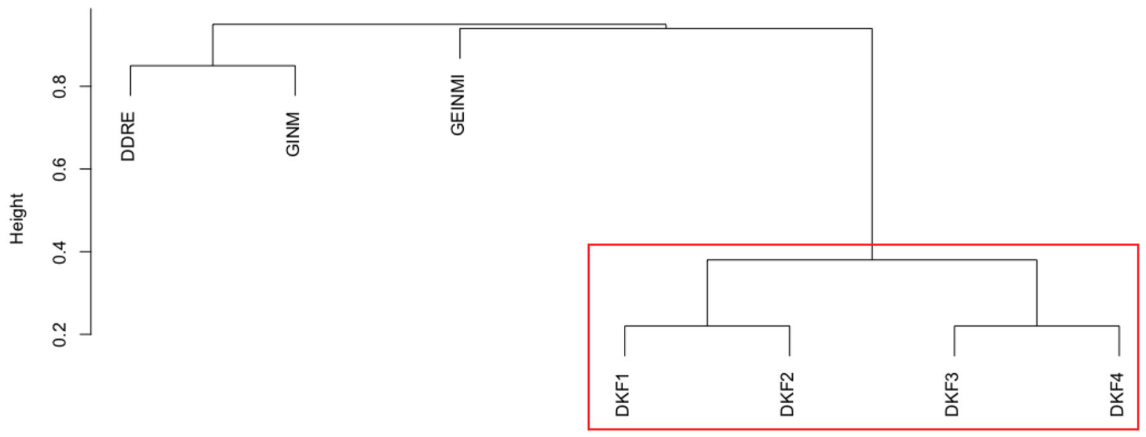


Fig. 15 The resulting classification dendrogram associated to the weighted dissimilarity matrix in Fig. 14

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*[8] 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. The aim of this transformation is to rename the application package name in both the Android Manifest file and all the classes of the application.
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* Some detection signatures could exploit the call graph of the application. To evade such signatures, a transformation is designed to mutate the original call graph of the application by modifying every method invocation in the *smali* code with a call to a new method inserted by the transformation which simply invokes the original method.
7. *Code Reordering* This transformation is aimed at modifying the instructions order in *smali* methods. A random reordering of instructions has been accomplished by inserting *goto* instructions with the aim of preserving the original run-time execution trace.
8. *Defunct Methods* This transformation adds new methods that perform defunct functions to *smali* code, while not changing the logic of the original source code.
9. *Junk Code Insertion* These transformations introduce those code sequences that have no effect on the function of the code. Detection algorithms relying on instructions (or opcodes) sequences may be defeated by this type of transformations. This type of transformations provides three different junk code insertions: (i) insertion of *nop* instructions into each method, (ii) insertion of unconditional jumps into each method, and (iii) allocation of three additional registers on which garbage operations are performed.
10. *Encrypting Payloads and Native Exploits* In Android, native code is usually made available as libraries accessed via JNI. However, some malware, such as DroidDream, also pack native code exploits meant to run from a command line in non-standard locations in the application package. All such files may be stored encrypted in the application package and be decrypted at run-time. Certain malware such as DroidDream also carry payload applications that are installed once the system has been compromised. These payloads may also be stored encrypted. Payloads are categorized and encryption as DSA is exploited because signature-based static detection is still possible based on the main application's bytecode. These are easily implemented and have been observed in practice as well (e.g., DroidKungFu malware uses encrypted exploit).
11. *Function Outlining and Inlining* In function outlining, a function is broken down into several smaller functions. Function inlining involves replacing a function call with the entire function body. These are typical compiler optimization techniques. However, outlining and inlining can also be used for call graph obfuscation.

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

Table 5 Classification results: precision, recall and ROC area for classifying Malware samples and families computed with different algorithms

Family	Algorithm	Precision	Recall	ROC area	% ROC area decrease in transformed apps
All families	J48	0.819	0.979	0.809	2.05
	HoeffdingTree	0.944	0.861	0.894	1.44
	NBTree	0.903	0.938	0.9	1.01
	RandomForest	0.872	0.928	0.883	0.67
	RandomTree	0.915	0.89	0.867	1.23
	RepTree	0.832	0.966	0.854	1.27
Adrd	J48	0.902	0.648	0.762	2.47
	HoeffdingTree	0.899	0.873	0.887	0.78
	NBTree	0.877	0.704	0.818	1.39
	RandomForest	0.955	0.296	0.824	2.84
	RandomTree	0.868	0.465	0.617	9.54
	RepTree	0.877	0.704	0.799	2.7
DroidDream	J48	0.965	0.696	0.769	2.53
	HoeffdingTree	0.798	0.949	0.956	2.88
	NBTree	0.938	0.949	0.96	1.8
	RandomForest	1	0.38	0.869	2.61
	RandomTree	0.956	0.544	0.608	8.99
	RepTree	0.982	0.709	0.81	0.51
DroidKungFu1	J48	1	0.688	0.846	4.85
	HoeffdingTree	0.595	0.781	0.839	1.18
	NBTree	0.926	0.781	0.917	1.24
	RandomForest	0.919	0.990	0.871	3.1
	RandomTree	0.88	0.688	0.842	0.34
	RepTree	0.96	0.75	0.857	2.52
DroidKungFu2	J48	0.923	0.75	0.949	4.14
	HoeffdingTree	0.925	0.835	0.844	1.73
	NBTree	0.938	0.938	0.938	1.52
	RandomForest	0.889	0.5	0.953	2.43
	RandomTree	0.818	0.563	0.878	7.37
	RepTree	0.917	0.688	0.877	2.05
DroidKungFu3	J48	0.98	0.845	0.911	3.04
	HoeffdingTree	0.754	0.845	0.901	0.35
	NBTree	0.825	0.897	0.915	1.13
	RandomForest	0.92	0.793	0.96	0.23
	RandomTree	0.978	0.759	0.914	11.5
	RepTree	0.98	0.862	0.935	0.5
DroidKungFu4	J48	0.82	0.79	0.88	3.2
	HoeffdingTree	0.932	0.845	0.898	0.28
	NBTree	0.81	0.788	0.88	2.3
	RandomForest	0.82	0.72	0.734	0.43
	RandomTree	0.95	0.784	0.904	8.3
	RepTree	0.92	0.87	0.91	1.2
FakeInstaller	J48	0.734	0.783	0.827	4.18
	HoeffdingTree	0.852	0.867	0.928	1.46
	NBTree	0.831	0.9	0.935	1.3

Table 5 continued

Family	Algorithm	Precision	Recall	ROC area	% ROC area decrease in transformed apps
	RandomForest	0.93	0.667	0.93	1.97
	RandomTree	0.914	0.533	0.817	8.87
	RepTree	0.758	0.833	0.868	0.63
Geinimi	J48	0.72	0.67	0.7	3.12
	HoeffdingTree	0.81	0.75	0.78	1.15
	NBTree	0.78	0.71	0.75	1.33
	RandomForest	0.83	0.87	0.84	1.81
	RandomTree	0.81	0.85	0.833	2.72
	RepTree	0.77	0.833	0.862	0.88
Kmin	J48	0.968	0.741	0.866	4.01
	HoeffdingTree	0.971	0.827	0.911	1.21
	NBTree	0.946	0.864	0.913	1.74
	RandomForest	0.961	0.914	0.959	1.01
	RandomTree	0.938	0.926	0.966	2.93
	RepTree	0.955	0.79	0.893	1.05
Opfake	J48	0.797	0.953	0.861	3.44
	HoeffdingTree	0.909	0.943	0.882	1.95
	NBTree	0.908	0.933	0.913	1.63
	RandomForest	0.834	0.966	0.917	1.83
	RandomTree	0.833	0.939	0.895	2.48
	RepTree	0.821	0.939	0.897	2.87

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.

6.1 Dataset construction for robustness analysis

We apply the full transformation set to our samples with the Droidchameleon [37] and the ADAM [51] tools. Furthermore, we use an obfuscation engine[9] able to inject six different obfuscation techniques in Android applications. Table 6 shows the obfuscation techniques implemented by the three tools.

The robustness analysis was performed following the process depicted in Fig. 16. As the figure shows, we combined together all the twelve transformations earlier discussed in this section in order to obtain an obfuscated dataset. The applications in the test set have been transformed to make detection much more difficult. The aim is to verify that our approach is insensitive to behavior-preserving static code transformations that do not alter the system calls sequences that are used by the malware payload to accomplish its malicious tasks. This experiment aims at verifying that such assumption holds on obfuscated real malware making our classifiers robust to code transformations that are typically exploited by malware developers to avoid detection. As shown in the process in Fig. 16, from the test set used in

Table 6 Comparison between the transformation techniques implemented in the three tools

Code Transformation	Obfuscation engine	DroidChamelon tool	ADAM tool
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	

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

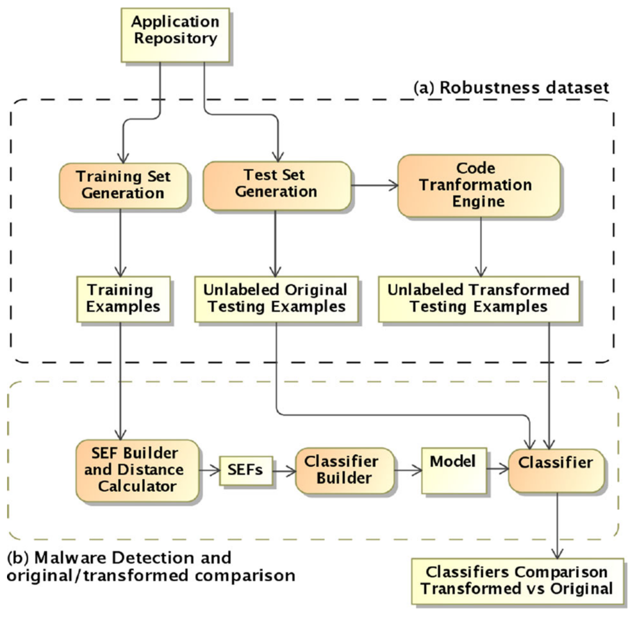

Application Repository
(a) Robustness dataset
Training Set Generation
Test Set Generation
Code Tranformation Engine
Training Examples
Unlabeled Original Testing Examples
Unlabeled Transformed Testing Examples
SEF Builder and Distance Calculator
SEFs
Classifier Builder
Model
Classifier
(b) Malware Detection and original/transformed comparison
Classifiers Comparison Transformed vs Original

Fig. 16 Robustness to code-transformation assessment process

Sect. 5 to validate the approach, we derive a transformed set by applying the code transformations. The validation step is executed on both the original and transformed sets to assess the performance loss of the classifiers on obfuscated applications.

6.2 Discussion of results

Figure 12 reports the distances distributions among SEFs models, including the transformed set. The comparisons between distances distributions of the test sets highlights, as expected, a small deterioration in the distribution parameters of the transformed ones. However, the deterioration in almost all the cases consists of a small increase of the inter-quartile range for transformed applications with respect to test applications that never causes an overlapping of median values with trusted applications. This means that the mined SEF models maintain the same discriminative properties assessed for the test applications. Looking at the boxplots for the transformed set, we can see that in most cases the third quartile is more stable than the first one across both events and families. This means that the transformed SEF models have distances, with respect to family models, that are greater if compared to test models, but still lower than the first quartile of the trusted models. The transformed set has been exploited to validate the classifier built with the original training set and the detection quality has been evaluated in order to assess the robustness of the approach with respect to code transformations.

The results of the family classifiers reported in Table 5 include in the last column the percentage decrease of the ROC Area (of each classifier) for the transformed applications. The decrease ranges between 0.34 and 11.5%, depending on the malware family and the classification algorithm, but on average, it is equal to 2.6%. This means that the ROC Area maximum decrease for transformation is lower than 0.1 (in the worst case). The first interesting thing to observe is that the worst decrease relate to classifiers that were already performing poorly (ROC is less than 0.6): they were already bad classifiers with test set and they remain bad on transformed samples. This is confirmed as a trend across all classifiers since decreases are higher for classifiers that have worse performances (lower ROC area). This can be seen looking at Fig. 17 that shows a direct comparison between the ROC area for test and transformed applications (Plain and Transformed bars). As it can be easily observed from the bar chart, most of the decreases are in correspondence with the lowest "Plain" values. To summarize, our analysis shows that, out of the total 60 classifiers[10], for 9 classifiers ROC decreases to a value less than 0.9, whereas for 3 classifiers it becomes lower than 0.8. The other classifiers (48) remain almost unchanged. These are very good results confirming that the approach is quite insensitive to code transformation techniques since it is based on a fingerprint of the application and malware behavior, which is left unchanged by behavior-preserving code transformations. More investigation could be performed on the individual learning algorithm sensitivity in order to reveal why we obtained the worst results for two specific algorithms: J48 (3.4%) and RandomTree (6.6%).

7 Threats to validity

The *construct validity* analysis showed that the syscall traces extraction could be imprecise in some circumstances. This is due to the trace capturing script that starts to capture the trace when an system event occurs (i.e., when it is sent by the sandbox to the application) and stops to capture when a new stable state is reached. The source of imprecision lies in the trace automatic cut which does not take into account what happens to the application during the capturing time. This entails that some incomplete traces could be captured and should be discarded from the set: for example, if an application is shut down due to an illegal behavior, an incomplete trace is captured. This issue was revealed performing outliers analysis and allowed to improve the approach introducing a trace validation step that is able to filter out incorrect traces from the dataset. Another source of imprecision is related to traces captured from malware applications that do not trigger the malicious behavior in response to some system events. However, this kind of issues are effectively

[10] The classifiers are 60 since we have 10 malware families classifiers plus a single "catch-all" classifier, multiplied by 6 learning algorithms.

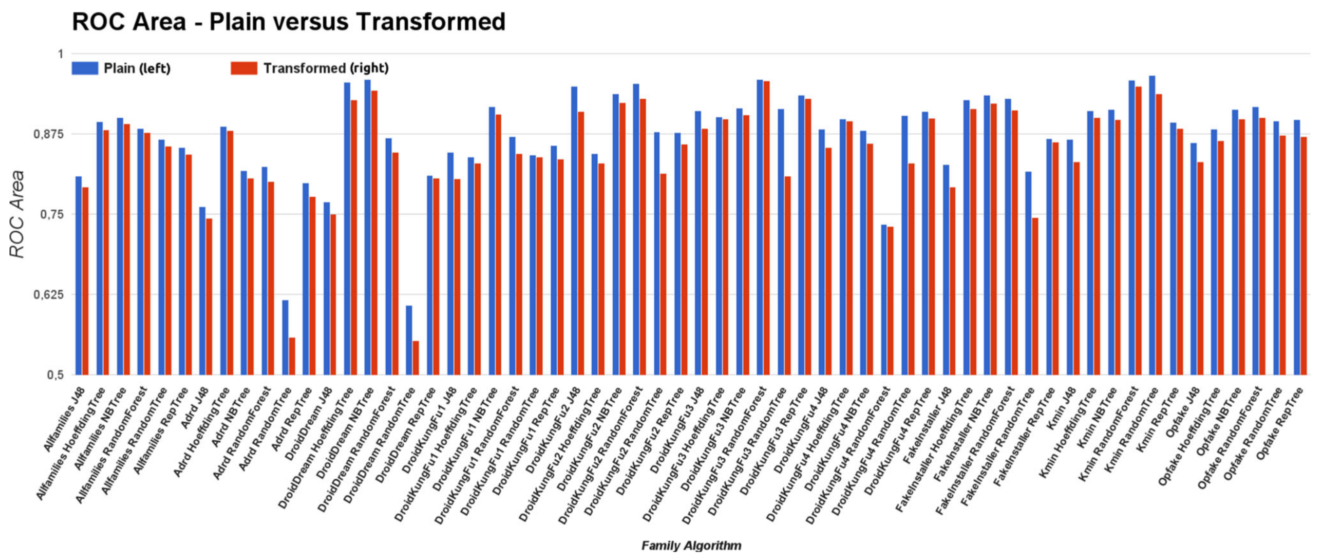


Fig. 17 Area under ROC curve variation between plain and transformed malware

detected during the training process looking at the distribution of distances among test and trusted applications. For example, looking at the boxplots of Fig. 11, we recognize that DroidKungFu malware is triggering the malicious behavior for BOOT_COMPLETED, BATTERY_LOW and INPUT_MEDIA_CHANGED events allowing an effective detection. For malware not triggering malicious behavior for any of the system events, our approach is not able to perform the detection but this eventuality can be revealed during the training phase.

Another construct validity threat lies in the way the dataset is obtained. It is assumed that the considered applications are malicious on the base of the response of some antivirus software that do not provide any assurance. To reduce mistakes, a combination of several antimalware is adopted for the quality verification step and the application is considered as infected if the infection is recognized by at least five different antimalware.

For what concerns the *external validity* threats and the generalization of our findings, our evaluation validates the approach implementation on more than 1200 applications of ten malware families. Even if this allows to obtain statistically significant results, an extension of these results to more malware families and to a larger set of applications is still desirable.

8 Conclusions

In this paper, we proposed an approach for dynamic malware detection and malware phylogeny tracking based on process mining techniques. It extracts a declarative model, named SEF, from system calls traces of malware and trusted applications, which represents a fingerprint of their respective dynamic behavior. Based on the distances between the mined SEF models, similarities across malware families are identified and several malware variants are characterized.

In terms of malware detection, the approach has been evaluated on a dataset of more than 1200 infected applications across ten malware families. The obtained results are encouraging and show that the approach is effective in detecting malware, thanks to the capability of the SEF models to adequately represent malware behaviors.

In order to assess the effectiveness and the efficiency of the approach in phylogeny tracking, we applied it to the same dataset of applications. The results show the capability to effectively discriminate the ten studied malware families and recognize variants of the same family of malware.

Finally, a robustness analysis against most common code obfuscation techniques has been performed on the same dataset of applications. In this study, SEF models have shown to be quite insensitive to behavioral-preserving code transformations techniques. As a consequence, the proposed malware detection and phylogeny tracking approach significantly reduce the false negatives in the presence of obfuscated malware and variants of malware families.

Acknowledgements This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.

References

1. Androguard. https://code.google.com/p/androguard/, last visit 24 November 2014
2. Anderson, B., Storlie, C., Lane, T.: Improving malware classification: bridging the static/dynamic gap. In: Proceedings of the 5th

ACM Workshop on Security and Artificial Intelligence, AISec '12, pp. 3–14, New York, NY, USA. ACM (2012)
3. Arora, A., Garg, S., Peddoju, S.K.: Malware detection using network traffic analysis in android based mobile devices. In: 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies (NGMAST), pp. 66–71 (Sept 2014)
4. Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., Rieck, K.: DREBIN: efficient and explainable detection of android malware in your pocket. In: Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS) (2014)
5. Battista, P., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Identification of android malware families with model checking. In: International Conference on Information Systems Security and Privacy. SCITEPRESS (2016)
6. Bernardi, M.L., Cimitile, M., Di Francescomarino, C., Maggi, F.M.: Do activity lifecycles affect the validity of a business rule in a business process? Inf. Syst. **62**, 42–59 (2016)
7. Bernardi, M.L., Cimitile, M., Di Lucca, G.A., Maggi, F.M.: Using declarative workflow languages to develop process-centric web applications. In: 16th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOC Workshops, Beijing, China, September 10–14, 2012, pp. 56–65 (2012)
8. Bernardi, M.L., Cimitile, M., Mercaldo, F., Distante, D.: A constraint-driven approach for dynamic malware detection. In: 14th IEEE Annual Conference on Privacy Security and Trust (2016)
9. Bose, R.P., Maggi, F.M., Aalst, W.M.P.: Enhancing Declare Maps Based on Event Correlations, chapter Business Process Management: 11th International Conference, BPM 2013, Beijing, China, August 26–30, 2013. Proceedings, pp. 97–112. Springer, Berlin (2013)
10. Burattin, A., Cimitile, M., Maggi, F.M., Sperduti, A.: Online discovery of declarative process models from event streams. IEEE Trans. Serv. Comput. **8**(6), 833–846 (2015)
11. Canfora, G., Mercaldo, F., Visaggio, C.A.: A classifier of malicious android applications. In: 2013 Eighth International Conference on Availability, Reliability and Security (ARES), pp. 607–614 (Sept 2013)
12. Canfora, G., Di Sorbo, A., Mercaldo, F., Visaggio, C.A.: Obfuscation techniques against signature-based detection: a case study. In: 2015 Mobile Systems Technologies Workshop (MST), pp. 21–26. IEEE (2015)
13. Canfora, G., Medvet, E., Mercaldo, F., Visaggio, C.A.: 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, chapter Detection of Malicious Web Pages Using System Calls Sequences, pp. 226–238. Springer, Cham (2014)
14. Canfora, G., Medvet, E., Mercaldo, F., Visaggio, C.A.: Detecting android malware using sequences of system calls. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015, pp. 13–20, New York, NY, USA, 2015. ACM (2015)
15. Carrera, E., Erdélyi, G.: Digital genome mapping—advanced binary malware analysis. In: Virus Bulletin Conference, Vol. 11 (2004)
16. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys '11, pp. 239–252, New York, NY, USA, 2011. ACM (2011)
17. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX Conference on Security, SEC'11, pp. 21–21, Berkeley, CA, USA, 2011. USENIX Association (2011)
18. Gartner Report of February 2017. http://www.gartner.com/newsroom/id/3609817 (2017)
19. Hayes, M., Walenstein, A., Lakhotia, A.: Evaluation of malware phylogeny modelling systems using automated variant generation. J. Comput. Virol. **5**(4), 335–343 (2008)
20. Holmes, G., Donkin, A., Witten, I.H.: Weka: A machine learning workbench. In: Proceedings of the Second Australia and New Zealand Conference on Intelligent Information Systems, pp. 357–361. Citeseer (1994)
21. Isohara, T., Takemori, K., Kubota, A.: Kernel-based behavior analysis for android malware detection. In: Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security, CIS '11, pp. 1011–1015, Washington, DC, USA, 2011. IEEE Computer Society (2011)
22. Jain, A.K., Murty, M.N., Flynn, P.J.: Data clustering: a review. ACM Comput. Surv. **31**(3), 264–323 (1999)
23. Jang, J., Brumley, D., Venkataraman, S.: BitShred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pp. 309–320, New York, NY, USA, 2011. ACM (2011)
24. Jeong, Y., Lee, H., Cho, S., Han, S., Park, M.: A kernel-based monitoring approach for analyzing malicious behavior on android. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, pp. 1737–1738, New York, NY, USA, 2014. ACM (2014)
25. Jiang, X., Zhou, Y.: Android Malware. Springer, New York (2013)
26. Karim, M.E., Walenstein, A., Lakhotia, A., Parida, L.: Malware phylogeny generation using permutations of code. J. Comput. Virol. **1**(1–2), 13–23 (2005)
27. Khoo, W.M., Lió, P.: Unity in diversity: phylogenetic-inspired techniques for reverse engineering and detection of malware families. In: 2011 First SysSec Workshop (SysSec), pp. 3–10. IEEE (2011)
28. Ma, J., Dunagan, J., Wang, H.J., Savage, S., Voelker, G.M.: Finding diversity in remote code injection exploits. In: Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement, IMC '06, pp. 53–64, New York, NY, USA, 2006. ACM (2006)
29. Mobile Threat Report. https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf, last visit 26 February 2016
30. Mario, F.M., Bernardi, L., Cimitile, M.: Process mining meets malware evolution: a study of the behavior of malicious code. In: 2015 Fourth International Symposium on Computing and Networking (CANDAR) (Dec 2016)
31. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Download malware? No, thanks. How formal methods can block update attacks. In: Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, pp. 22–28. ACM (2016)
32. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A.: Ransomware steals your phone. Formal methods rescue it. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, pp. 212–221. Springer (2016)
33. Oberheide, J., Mille, C.: Dissecting the android bouncer. In: SummerCon (2012)
34. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: full support for loosely-structured processes. EDOC **2007**, 287–300 (2007)
35. Picinbono, B.: On deflection as a performance criterion in detection. IEEE Trans. Aerosp. Electron. Syst. **31**(3), 1072–1081 (1995)
36. Rastogi, V., Chen, Y., Jiang, X.: Catch me if you can: evaluating android anti-malware against transformation attacks. IEEE Trans. Inf. Forensics Secur. **9**(1), 99–108 (2014)
37. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information,

Computer and Communications Security, ASIA CCS '13, pp. 329–334, New York, NY, USA, 2013. ACM (2013)
38. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of EuroSec (2013)
39. Sahs, J., Khan, L.: A machine learning approach to android malware detection. In: Proceedings of the European Intelligence and Security Informatics Conference (2012)
40. Schmidt, A.-D., Schmidt, H.-G., Clausen, J., Yuksel, K.A., Kiraz, O., Camtepe, A., Albayrak, S.: Enhancing security of linux-based android devices. In: Proceedings of 15th International Linux Kongress (2008)
41. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pp. 1808–1815, New York, NY, USA, 2013. ACM (2013)
42. Tchakounté, F., Dayang, P.: System calls analysis of malwares on android. Int. J. Sci. Tecnol. (IJST) **2**(9), 669–674 (2013)
43. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)
44. van Dongen, B.F., de Medeiros,A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The prom framework: a new era in process mining tool support. In: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets, ICATPN'05, pp. 444–454, Berlin, Heidelberg, 2005. Springer (2005)
45. Virustotal. https://www.virustotal.com/, last visit 1 March 2016
46. Walenstein, A., Lakhotia, A.: A transformation-based model of malware derivation. In: 2012 7th International Conference on Malicious and Unwanted Software (MALWARE), pp. 17–25 (Oct 2012)
47. Wang, X., Jhi, Y.-C., Zhu, S., Liu, P.: Detecting software theft via system call based birthmarks. In: Proceedings of the 2009 Annual Computer Security Applications Conference, ACSAC '09, pp. 149–158, Washington, DC, USA, 2009. IEEE Computer Society (2009)
48. Wei, T.-E., Mao, C.-H., Jeng, A.B., Lee, H.-M., Wang, H.-T., Wu, D.-J.: Android malware detection via a latent network behavior analysis. In: Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM '12, pp. 1251–1258, Washington, DC, USA, 2012. IEEE Computer Society (2012)
49. Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K.: Android malware detection based on system call sequences and LSTM. Multimedia Tools and Applications (Sept 2017)
50. Yan, L.K., Yin, H.: DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security'12, pp. 29–29, Berkeley, CA, USA, 2012. USENIX Association (2012)
51. Zheng, M., Lee, P.P.C., Lui, J.C.S.: 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. pp. 82–101. Springer (2012)
52. Zheng, M., Sun, M., Lui, J.C.S.: Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM '13, pp. 163–171, Washington, DC, USA, 2013. IEEE Computer Society (2013)
53. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY '12, pp. 317–326, New York, NY, USA, 2012. ACM (2012)
54. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, pp. 95–109, Washington, DC, USA, 2012. IEEE Computer Society (2012)