Machine Learning (2023) 112:1171–1199
https://doi.org/10.1007/s10994-022-06154-3

Data-aware process discovery for malware detection: an empirical study

Mario Luca Bernardi[1] · Marta Cimitile[2] · Fabrizio Maria Maggi[3]

Received: 1 March 2021 / Revised: 11 November 2021 / Accepted: 19 February 2022 /
Published online: 31 March 2022

Abstract
Mobile devices are undeniably becoming essential in our lives and our daily activities. The adoption of mobile applications increases the human computing experience and the capability to access and exchange data. However, mobile devices are also the target of several malware attacks, usually obtained by evolving existing malicious code. This allows researchers and practitioners to recognize malware applications based on their similarities with existing infected applications. This study uses a multi-perspective declarative language to model the behavior of infected and trusted applications by discovering it from their system call traces. The obtained models are used to classify malware applications and evaluate if they belong to a known malware family. The approach has been evaluated on a dataset obtained by capturing system call traces from more than $160K$ trusted and infected applications, the latter gathered from 27 known malware families. The empirical study shows the good performance of the approach in the identification of the infected applications and their membership to a specific malware family. In addition, the approach exhibits a high level of robustness to code transformations and major evasion techniques.

Keywords Malware detection · Declarative process mining · Data-aware process mining · Deep neural network.

Editors: Annalisa Appice, Grigorios Tsoumakas.

✉ Marta Cimitile
marta.cimitile@unitelmasapienza.it

Mario Luca Bernardi
bernardi@unisannio.it

Fabrizio Maria Maggi
maggi@inf.unibz.it

1 University of Sannio, Benevento, Italy

2 Unitelma Sapienza University, Rome, Italy

3 Free University of Bozen-Bolzano, Bolzano, Italy

1 Introduction

Mobile applications are becoming the main means to access and transfer information including personal and sensitive data. This causes an increasingly higher number of malware attacks finalized to steal information and money, promote fraudulent websites, or generate disruption. For this reason, in the last years, many research studies have focused on the detection of malware attacks. Existing approaches for malware detection are, in general, always aimed at discovering new malware on the basis of its derivation from existing malware families. This choice is motivated by the fact that the majority of these attacks are performed by using known infected code opportunely changed by automatic tools to become more unexpected and aggressive (Jang et al., 2011).

This paper starts from the idea that mobile application system calls can be modeled similarly to process activities in business process logs and business process models can be used as an abstraction of the behavior of an application. More precisely, the event-driven nature of mobile operating systems implies that applications participate in the system based on responses to specific activating events in a similar way process actors participate in a business process. As a consequence, process mining (van der Aalst, 2016) can be effectively used to analyze the behavior of trusted/infected applications from a set of system call traces obtained in response to these triggering activating events. Based on this idea, in (Bernardi et al., 2019), an approach is introduced to detect malware applications and their deriving families on the basis of similarities among process models mined from system call traces. However, the limit of that study concerns the mined models since they are expressed as constraints on the sequences of events and do not take into account data attributes attached to them. In contrast, in this paper, we analyze the system call traces also considering their data payloads making the classification more precise. Our hypothesis is that using the data payloads of events recorded in the system call traces gives further information about the malware behavior so that the malware is better characterized and, therefore, easier to detect. This concept is new also with respect to the similar studies (Alazab, 2015; Wu et al., 2012) that are also based on the analysis of syscall sequences to mine a fingerprint of the malware behavior but do not use the data perspective to represent the malware behavior captured from the syscall traces.

In particular, we represent the mobile application behavior as process models expressed using MP-Declare (Leno et al., 2020), a multi-perspective version of the declarative constraint-based process modeling language Declare (Pesic et al., 2007). It is the first time that MP-Declare is adopted in a malware detection context. The mobile application behavior is expressed as a set of constraints over system calls and their data payloads that we call *Data-aware System Calls Execution Fingerprint* (DSEF). The obtained models are then used to identify malware applications and evaluate their similarities with known malware families. To this aim, a Deep Neural Network (DNN) classifier is used and compared with other alternative classifiers.

The proposed empirical validation is performed on a new dataset obtained by capturing system call traces from more than 160K trusted and infected applications and aims at evaluating different aspects of the proposed approach. Since, in our approach, it is possible to adopt different types of classifiers, we test the level of accuracy that they can guarantee. In particular, we compute the F1-score of different types of neural networks and tree-based binary classifiers (i.e., DNN, CNN, J48, and RF) when detecting if an application is infected. Since the size of the datasets has an impact on the classification performance, in the evaluation, we also investigate the relation between the number of infected applications used to build the DSEF models and the final F1-score of the classifiers. In malware development, different code obfuscation approaches are used as evasion techniques to hide the malware. Therefore, our empirical validation also investigates to what extent such obfuscation techniques negatively affect the accuracy of the proposed malware detection approach. Finally, we investigate the accuracy of a different type of classification task that can be carried out with our approach, i.e., multinomial classification, aiming at detecting a specific malware family among a set of known malware families.

This work extends our previous work (Ardimento et al., 2020) by introducing several novelties:

- A significantly larger number of malware families are modeled and included in our approach (we consider 19 additional malware families with respect to the 8 families considered in our former study (Ardimento et al., 2020);
- Different classifiers are used and compared in the classification step;
- The performance of the classifier is evaluated also when different obfuscation techniques are used in the malware source code;
- The impact of the number of infected applications used to build the DSEF models on the classifier performance is evaluated (considering both binary and multinomial classification).

The approach is relevant for all kinds of mobile platforms, but this study regards the Android platforms since they are the favorite target of malware attacks (Mobile threat report, 2016).

It is important to point out that this approach cannot be used to detect unknown malware families since the DSEF models of the malware families need to be known beforehand. Furthermore, the approach is not suitable to be used as an antivirus tool as it requires the execution of malicious code on users' devices and the acquisition of post-infection traces to generate the models (two things that users would do quite reluctantly). On the other hand, this approach, being completely automatic, can be used very effectively at the application store level (e.g., Android Play Store, iOS App Store, or even third party store like APKMirror or Aptoide) where the new applications contributed by developers can be executed on a sandbox, their traces analyzed and, finally, approved only if no malicious behaviors similar to the ones of known malware families are found. This also includes zero-day malware derived from existing families and sharing significant parts of their behavior.

The paper is organized as follows. Section 2 introduces some background notions useful to understand the rest of the paper. Section 3 discusses the related work. In Sect. 4, the proposed approach is described. Section 5 reports the evaluation carried out using a dataset containing system call traces generated by trusted and malicious applications derived from different malware families. Section 6 discusses the threats to validity of the evaluation. Finally, Sect. 7 provides some conclusive remarks and future work discussion.

2 Background

In this section, we introduce the concepts needed to understand the rest of the paper. In particular, in Sect. 2.1, we introduce malware families and the corresponding events that trigger their malicious behavior. In Sect. 2.2, we introduce the obfuscation techniques that can be used by the malware developers to hide the malicious behavior. In Sect. 2.3, we present the process modeling language that we use as a basis of our malware detection approach to representing malware behavior.

2.1 Mobile malware families

We start by presenting the objective of our detection approach, i.e., the malware families. The presented technique has been developed to characterize the behavior of malware families and distinguish it from the behavior of trusted applications. Malware is any software intentionally designed to disrupt, damage, or gain unauthorized access to data. Malware families are defined as sets of malware with similar behavior and properties. Malware detection can benefit from the study and formalization of malware families' behavior since new malware is often obtained by evolving existing ones (Karim et al., 2005). According to (Fedler and Schütte, 2013), there are several installation techniques (ITs) for installing a malicious payload. A very common technique is repackaging (R), consisting of decompiling an application to get its source code and add the malicious payload. Further ITs are standalone (S) and update attacks (U) (Zhou and Jiang, 2012). The standalone IT consists of installing the malicious payload in a standalone application. In the update attack, the malicious payload is included in an update component allowing to fetch or download the malware at runtime. When the malicious payload is installed, the malicious behavior can be activated by an event. The activity events (AEs) are the events that are supposed to trigger the malicious behavior and that are sent to the mobile application during its execution to test its behavior (the behavior of the application after receiving the AE is recorded and used to detect the malware). AEs can be grouped into several broad classes, for Android OS these are: `BOOT`, `BATT`, `SMS`, `SYS`, `NET`, `CALL`. The `BOOT` class contains only the `BOOT_COMPLETED` event meaning that the operating system boot process has been completed. Concerning the `CALL` class, the possible AEs include incoming calls (`PHONE_STATE`) and outgoing calls (`NEW_OUTGOING_CALL`). The `SYS` class includes several AEs. `INPUT_METHOD_CHANGED` means that the operating system detects a change in any of the active input methods. The `USER_PRESENT` event occurs when the user unlocks the device while `SIG_STR` occurs when the signal strength changes. Another AE of this class is `SIM_FULL` meaning that the message storage is full. The AEs of the `SMS` class are the reception of an SMS message (`SMS_RECEIVED`) or of a WAP PUSH (`WAP_PUSH_RECEIVED`). The `BATT` class includes several AEs related to the possible battery status: in charge (`POWER_CONNECTED`), discharging (`POWER_DISCONNECTED`), fully charged (`BATTERY_OKAY`), at 50% (`BATTERY_LOW`), empty (`BATTERY_EMPTY`) or battery status changed (`BATTERY_CHANGED`). Finally, the `NET` class includes two possible AEs occurring when the wifi is started (`PICK_WIFI_WORK`) or when the connection state is changed (`CONNECTIVITY_CHANGE`).

The list of malware families considered in this study is shown in Table 1. The table reports, for each family, its description (second column), the installation types (ITs) in the third column, the classes of activating events (AEs) in the fourth column, and the number of infected applications contained in the dataset used in our experiments (fifth column).

2.2 Obfuscation techinques

The effectiveness of malware attacks can be improved by using obfuscation techniques. These techniques are aimed at hiding the presence of malware thus preventing malware detection instruments from detecting it. We introduce these techniques because, in our

Table 1 Malware families investigated in this study

Families	Description	ITs	Class of AEs	#App
Airpush	Unwanted ads are displayed without any consent	R	BOOT,BATT,SMS	15 690
DroidKungFu	It installs a backdoor allowing the access of attackers to the device	R	BOOT,BATT,SYS	7886
Dowgin	It displays third-party advertising that silently captures sensitive information	R	BOOT,SMS,SYS	7788
Fusob	It encrypts data and then forces victims to pay to unlock the device	R,U	BOOT,SMS,NET,BATT	4410
FakeInst	It sends SMS messages to premium-rate numbers or services	R,U	BOOT,CALL	4356
Mecor	It is a Trojan-Spy	S	BOOT	3848
Youmi	It monitors and captures user behavior and floods the device with unsolicited pop-up advertisements	R,U	BOOT,SMS,CALL	2838
Kuguo	It redirects the victim to malicious websites and continually display popup ads	R	BOOT,SMS,NET,BATT	2660
FakeDoc	It steals information and sends them to a remote server	R	BOOT, SMS, BATT	2562
FakeTimer	It installs a service on the device that provides access to an adult website and forwards information from the device to a remote server	R,U	BOOT,SYS,SMS	2262
FakePlayer	It pretends to be a media player but instead sends SMS	R,U	BOOT, SYS, BATT	2256
FakeUpdates	It makes personal information on a device vulnerable to a wide variety of additional security threats	R,U	BOOT,CALL	2226
Finspy	It steals sensitive information	R	BOOT,CALL	2054
Fjcon	It sends SMS messages to the device and installs packages without the user consent	R	BOOT	1982
Fobus	It sends premium SMS messages, makes calls without the user consent and steals private information	R	BOOT, SMS,NET,BATT	1966
GingerMaster	It provides the attackers with root-level access to the device	R	BOOT	1956
GoldDream	It monitors all in/outbound SMS and calls on the infected mobile device	R,U	BOOT,BATT,SMS	1752
Ksapp	It is a bot-net used to download new programs or launch DDOS attacks	R,U	BOOT,SMS,SYS	1724
Gopro	It is a small Trojan used to download other malware	R	BOOT,BATT,SYS	1698
Leech	It gains device root privileges	S	BOOT,CALL	1688
Kyview	It is a file infector	R	BOOT,SMS,NET,BATT	1652
Lnk	It is a shortcut file used to point to an executable file	R	BOOT	1570
Lotoor	It installs other malware or unwanted software in the device	R,U	BOOT,SMS,BATT	1556
Minimob	It exploits device vulnerabilities giving access to other malware	R	BOOT	1510
Winge	It is a Trojan Clicker	S	BOOT, SMS,BATT	1500
Zitmo	It is a keylogging malware stealing the device user banking credentials	R	BOOT,SYS,BATT	1460

Table 1 (continued)

Families	Description	ITs	Class of AEs	#App
Ztorg	It gains root rights on the infected device	R,U	BOOT,SYS,SMS	1428

study, we want to show that the malware detection approach we propose is effective even when obfuscation techniques are used.

There is a wide literature on how malware exploits well-known code transformation approaches (Zheng et al., 2012; Rastogi et al., 2014) as evasion techniques to avoid being identified.

The most used obfuscation approaches are the following:

1. *Disassembling & Reassembling* The compilation process of the application package may alter the structure and the representation of the resources including binary code files. This reduces the effectiveness of the approaches relying on these elements for malware detection (e.g., static signature-based approaches).
2. *Repacking* Every Android application has a developer signature key that can be removed during disassembling and reassembling the application. Using tools like `signapk`[1], it is possible to embed a new signature key in the reassembled application to avoid malware identification based on blacklisted keys.
3. *Package rename* Android applications are identified by a unique package name. Renaming the application package name in both the Android Manifest file and all the classes of the application avoids malware detection based on this identifier.
4. *Identifier renaming* This transformation renames symbols in the binary code files and metadata (e.g., the manifest and UI schemas) by using a random string generator. This negatively affects the performances of signature-based approaches that exploit string matching to identify malware.
5. *Data Encoding* Statically encoded strings are another common method used to create detection signatures to identify malware. This code transformation applies different encoding schemas to strings, to elude these signatures.
6. *Call indirections* Detection signatures often include a linearization of a portion of the application call graph. Behavior-preserving Graph-based transformations can be exploited to alter the original call graph of the application by modifying method invocations via indirection addition.
7. *Code Reordering* Using this transformation the order of the instructions in methods can be altered. Behavior-preserving random reordering of instructions can be effectively exploited using *goto* instructions leading to a different binary code that, when executed, generate the original run-time execution trace.
8. *Defunct Methods* This transformation adds dead code segments that do nothing but alter the structure of the binary code. Detection approaches that exploit checksum calculated on binary chunks are strongly impacted by this approach.
9. *Junk Code Insertion* With this transformation, code that does nothing useful is injected. Detection techniques that rely on opcodes sequences may be affected by this kind of alteration. There are three distinct junk code injections: (i) introduction of *nop* instructions within individual methods, (ii) injection of unconditional jumps into all methods, and (iii) allocation of additional registers on which useless operations are executed.
10. *Encrypting Payloads and Native Exploits* Native code is usually made available as libraries accessed via interfaces towards the underlying operating system (e.g., JNI on Android). Several malware families use this chance to pack native code exploits that can be executed from the command line and non-standard locations in the application

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

package. Such files are often encrypted in the application package and decrypted at run-time. In other cases, the additional native payload could even be downloaded after the malware is installed leaving the application package almost clean (except for the code that connects to the malicious command and control server). These techniques are easy to be implemented and have been recognized to be used frequently.

11. *Function Outlining and Inlining* Function outlining means splitting a function into many smaller functions. Conversely, function inlining means substituting a function call with the complete invoked function code (hence removing the call itself). These techniques can be effectively implemented to transform code thus making it not recognizable by a static detector.
12. *Reflection* This transformation changes any method call from static to dynamic thus exploiting the reflection language features. This makes the malicious code extremely difficult to be detected by statically analyzing the code.

Since malware writers in recent years have started to aggressively use the above obfuscation techniques, it becomes crucial, when evaluating a malware detection approach, to assess the robustness of the approach with respect to these techniques.

2.3 Multi-perspective process models

Here, we introduce the process modeling language at the basis of our malware detection approach. We use this language to represent the behavior of malware families (and trusted applications) as process models thus distinguishing malware behavior from trusted behavior by comparing these models.

This study proposes an approach that characterizes malware behaviors using data-aware rules. The language used to express these rules is Multi-Perspective Declare (MP-Declare) (Leno et al., 2020). This language is an evolution of the process modeling language Declare (Pesic et al., 2007) that models a process as a set of constraints to be satisfied throughout the process execution (Bernardi et al., 2012). All the sequences of events that do not violate the constraints are allowed. These constraints can be seen as concrete instantiations of parameterized constraint types called templates. Constraints inherit from templates both graphical representations and semantics based on LTL on finite traces (Pnueli, 1977) that ensure that the processes are verifiable and executable. The semantics of MP-Declare is based on Metric First-Order Linear Temporal Logic (MFOTL) (Chomicki, 1995). Using this logic, it is possible to enrich the standard control flow-based Declare language with the time and the data perspective. The parameters defined for MP-Declare constraints are an activation and a target. In particular, an *activation* is an event that triggers the constraint and its occurrence imposes some obligations on another event (the *target*) in the same process execution. MP-Declare constraints also specify conditions over data. An *activation condition* is a condition on the attributes of the activation payload and the constraint is activated only if, when the activation occurs, the activation condition holds. A *correlation condition* is a condition that correlates attributes of the activation payload and attributes of the target payload. The constraint is fulfilled only if, when the target occurs, the correlation condition holds.

We can give now a formal definition of an MP-Declare model.

Definition 1 **(MP-Declare model)** An MP-Declare model (MPD) is defined as:

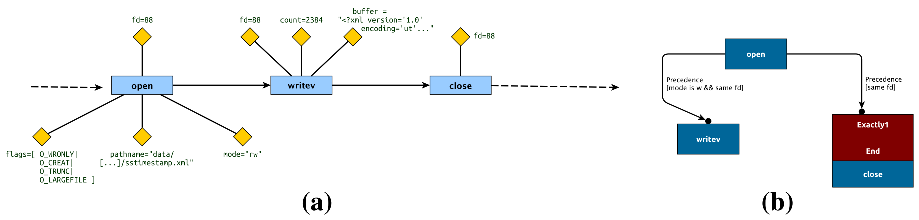

fd=88
fd=88
count=2384
buffer = "<?xml version='1.0' encoding='ut'..."
fd=88
open
writev
close
flags=[O_WRONLY| O_CREAT| O_TRUNC| O_LARGEFILE]
pathname="data/ [...]/sstimestamp.xml"
mode="rw"
(a)
open
Precedence [mode is w && same fd]
Precedence [same fd]
writev
Exactly1
End
close
(b)

Fig. 1 A syscall execution trace (**a**) conformant with an MP-Declare process model (**b**)

$$MPD = \{C_1, \dots, C_m\},$$

where $C_1 ... C_m$ are the constraints. Each constraint is a tuple $(T, S_0, S_t, P_{a_c}, P_{c_c})$ of type T (e.g., *precedence*) specifying an activation condition P_{a_c}, a correlation condition P_{c_c}, and having S_0 and S_t as activation and target syscall, respectively.

The MP-Declare model in Fig. 1b is composed of 4 constraints $MPD = \{C_1, C_3, C_3, C_4\}$. One of these constrant is of type precedence, with activation *writev*, target *open* and a correlation condition indicating that *open* must occur before *writev* with the same file id and mode containing ‘w’.

In this study, MP-Declare is used to model the application behavior. The syscalls of the applications can be considered as events and their relationships can be modeled as a set of constraints specified over events and their payloads. Consider, for example, the syscall sequence (syscall execution trace) described in Fig. 1a representing the trivial application that opens a file, writes some data into it, and finally, closes the file.

Each syscall corresponds to an event, which is associated with an event payload structure (attributes and types). For event *open* a file, the associated attributes are flags, pathname, mode, fd (file id). After the file opening, there is the file writing (*writev*) associated with another set of attributes (fd, count, and buffer). Finally, the file is closed (*close*). Note that the file id correlates all the events corresponding to the same file (fd=88 in the example).

The process execution represented in Fig. 1a can be characterized using MP-Declare rules as reported in Fig. 1b. The syscalls *open* and *writev* are related by a precedence constraint meaning that if *writev* occurs, then *open* occurs before with the same file id and mode containing ‘w’. Note that the precedence constraint does not force *open* to be necessarily followed by *writev*. The syscalls *open* and *close* are also related by a precedence constraint. This means that if *close* occurs, then *open* occurs before with the same file id. Also, in this case, the precedence constraint does not force *open* to be necessarily followed by *close*. Finally, the *close* event should occur only once as the last event. Note that these MP-Declare rules model the behavior where the same file can be opened and written several times and closed at the end.

In our study, we will take into consideration two approaches for the discovery of Declare constraints from logs: the approach presented in (Maggi et al., 2018) used in our baseline approach (Bernardi et al., 2019) for the discovery of control flow constraints and the approach introduced in (Leno et al., 2020) used in the approach presented in this paper for the discovery of MP-Declare models.

In the approach proposed in (Maggi et al., 2018), to extract a Declare model from an event log, first, a list of candidate constraints is generated by instantiating all possible constraint types over a selection of frequent activities. Then, the list of candidate constraints is pruned by checking their validity on the input event log and by considering only the ones that are valid in the majority of traces in the log. The approach presented in (Leno et al., 2020) starts from these control flow constraints and, in a second phase, enriches them with data conditions derived using classification techniques and redescription mining (Leno et al., 2020).

Note that, both the approaches for malware detection presented in (Bernardi et al., 2019) and the one presented in the current contribution use a declarative approach for the representation of the behaviors derived from syscall logs because this behavior is unpredictable and is characterized by high variability as it is common in all the so-called knowledge-intensive processes (Di Ciccio et al., 2015). Knowledge-intensive processes are not suitable to be represented using procedural process modeling languages since the process representation would become complex to understand for humans and difficult to process by machines. This is the reason why, for analyzing syscall logs, in (Bernardi et al., 2019) and in this paper, we use a declarative approach.

3 Related work

Mobile malware detection is becoming an important topic given the increasingly diffused adoption of mobile applications to access and transfer critical data. For this reason, several malware detection approaches have been proposed in the last few years. These approaches are mainly categorized as signature-based (Christodorescu et al., 2005) or behavioral (Arp et al., 2014; Bernardi et al., 2019) (according to the type of check they perform on the applications), and dynamic or static (based on whether the checks are performed at runtime or on the source code) (Bernardi et al., 2019).

Signature-based approaches classify an application as malicious if it contains in its instructions some predefined patterns (Feng et al., 2014). In the context of static signature-based approaches, DREBIN (Arp et al., 2014) and the approach proposed in (Talha et al., 2015) use authorization requests as input of machine learning algorithms to classify applications. Other useful sources for detection properties are control flow and data flow. In (Li and Li, 2015), the authors propose the use of a structured tree generated from data properties and use tree similarity to perform malware detection.

Contrary to the signature-based approaches, the behavioral approaches are focused on the behavior of the monitored application. Static behavioral approaches are presented in (Austin et al., 2013; Xu et al., 2016; Ding et al., 2014). In particular, in (Austin et al., 2013), a method using HMM (Hidden Markov Model) is proposed to analyze sequences of opcodes extracted from the .apk package for malware detection. The HMM model is trained with many malware applications and new incoming opcode sequences are classified according to their generative probabilities. ICCDetector (Xu et al., 2016) also extracts communication models called ICC (Inter Component Communication) from the source code to detect malware. Control flow analysis (Ding et al. 2014) was recently used to identify malicious payloads. The main disadvantage of such approaches is that they are very sensitive to obfuscation techniques that alter the structure and order of the source code instructions.

Dynamic behavioral methods (Jeong et al., 2014; Oak et al., 2019; Karbab et al., 2018; Su et al., 2016; Wei et al., 2012; Arora et al., 2014; Zhang et al., 2018; Canfora et al., 2015) analyze a possible infected application during its execution on a real or real-like device (our proposed approach belongs to this category). The approach described in (Jeong et al., 2014) detects malicious code by mining read/write operations from a real device having a customized kernel. Differently from this work, our proposed approach does not require devices to have a customized kernel, thus ensuring larger applicability. A dynamic analysis is also performed in (Oak et al., 2019), where deep learning techniques are used for malware detection. Similarly, in (Karbab et al., 2018), a deep learning malware detection framework using API call sequences as input, called *MalDozerare*, is introduced. Deep learning is also adopted in (Su et al., 2016), where a framework called *DroidDeep* is presented. This framework uses neural networks for feature extraction, and SVM to perform the classification. Several static and dynamic features are considered for characterizing the behavioral pattern of the applications.

Other approaches based on neural networks are proposed in (Wei et al., 2012; Arora et al., 2014). They mainly perform malware detection by analyzing data leakages. These approaches require more effort (in terms of time and resources) with respect to our approach since they need to recompile the kernel. Another recent study (Zhang et al., 2018) also puts together static and dynamic features to discover malware behavior. The approach is based on a procedure for the Android Application Package (APK) file decompiling. Also, in this case, the limit of the approach is that it is very time and resource-consuming. Another contribution to Android malware detection is presented in (Canfora et al., 2015). This study is based on the analysis of syscall sequences to mine a fingerprint of the malware. This approach is similar o the one proposed in (Bernardi et al., 2019) that uses process mining techniques to discover a model representing the malware behavior from a set of traces captured from trusted and infected running applications—the obtained model is called System Calls Execution Fingerprint (SEF). The SEF models are used to evaluate the similarities among malware families and are at the base of the classification derived from them.

The approach proposed in this paper can be seen as an evolution of the one reported in (Bernardi et al., 2019). In particular, the definition of SEF is now updated into the Data-aware System Calls Execution Fingerprint (DSEF). DSEF differs from SEF for the addition of data attributes as input for the analysis. This addition aims at improving the process mining step in terms of malware detection performance since the malicious behavior is often activated and conditioned by the data values associated with the syscalls. Moreover, with respect to (Bernardi et al., 2019), the classification is performed by using an approach based on neural networks and the obtained results are discussed and compared with other alternative classifiers. In our evaluation, we compare our findings with the results obtained in (Bernardi et al., 2019). This allows us to verify if the malware detection performance is influenced by the use of the data attributes.

The approach presented in this paper falls under the umbrella of the approaches that in the process mining field are called variant analysis approaches. As shown in (Taymouri et al., 2021), several mechanisms can be used for the analysis of process variants, and the comparison of process models derived from logs each including the behavior of a different variant is only one of the mechanisms that can be used to compare process behaviors. We are currently planning to provide benchmarks showing how the different variant analysis mechanisms presented in (Taymouri et al., 2021) can be applied in the context of malware detection. Finally, other approaches in the context of process mining that goes beyond variant analysis can be used for malware detection. For example, techniques for concept drift detection (Ostovar et al., 2020) or anomaly detection (Bezerra et al., 2009) can be used to distinguish different behaviors in syscall logs.

4 The malware detection approach

In this section, our malware detection approach is described. The approach aims at verifying if a given malware infected an Android Application Package (APK). The approach is divided into two phases. In the first phase, a classifier is trained to be able to discriminate between trusted and infected applications. In the malware detection phase, the classifier is used to classify a new application as trusted or infected (and with which malware family). In both phases, we need to evaluate the distance between the behavior of an APK application and the behavior of a malware family.

To this aim, we use Data-aware System Calls Execution Fingerprint (DSEF) models to represent the behavior of an APK application and the behavior of a malware family. Then, we define a notion of distance between DSEF models and use this notion to characterize the distances between the behaviors of trusted and infected APK applications and the behaviors of the malware families. These distances are, in turn, used to train a classifier able to distinguish between trusted and infected applications.

In the following, we first describe how to build a DSEF model of an APK application and of a malware family, and then how to use these models to train a classifier and use it to classify a new application as trusted or infected with a certain malware family.

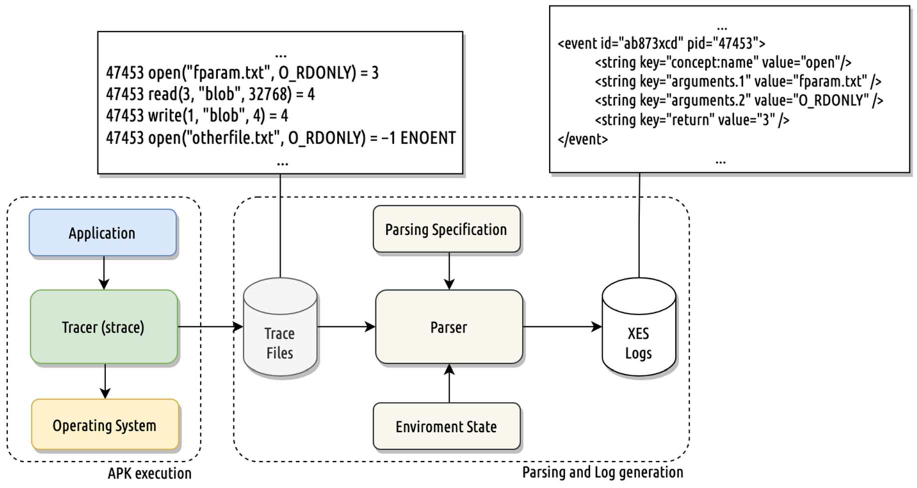
4.1 How to build a DSEF model

To build a DSEF model either of an APK application or of a malware family, we need to follow two steps: the syscall traces extraction and the MP-Declare mining.

4.1.1 Syscall traces extraction

Capturing traces in the Android platform can be achieved by using the Linux system trace utility called *strace*. It takes as arguments a command and runs it making use of a specialized kernel system call, called *ptrace*, to intercept all the system calls called by that command. The *strace* command prints the name of each intercepted system call, followed by its arguments and its return values, into a file provided as an option. More interesting, *strace* includes a verbose option which allows dumping all the arguments and the return values of the executed system calls. This approach is sufficient to retrieve the information needed to syscall traces from the Android platform in a very concise and parsing-friendly output format.

Figure 2 shows an example of *strace* output. Each line represents a recorded system call. In the first place, the id of the process in which the system call was executed is given. Then the name of the system call is dumped along with its arguments enclosed within brackets. Finally, the result of the system call is given right after the equal sign. In this example, an open system call is first recorded. This system call takes as arguments the string "fparam.txt" which is the path of the file to open, and the RDONLY flag telling that the file needs to be opened only for reading. The system call succeeds and returns a file descriptor (namely 3), pointing to the file just opened. In line 2, a read system call is issued on this file descriptor, and the text "data" is read from "fparam.txt". Our parser can understand the trace files and exploits parsing specifications related to the specific android platform in use and information from the environment state in


```
...
47453 open("fparam.txt", O_RDONLY) = 3
47453 read(3, "blob", 32768) = 4
47453 write(1, "blob", 4) = 4
47453 open("otherfile.txt", O_RDONLY) = -1 ENOENT
...
```

```
...
<event id="ab873xcd" pid="47453">
        <string key="concept:name" value="open"/>
        <string key="arguments.1" value="fparam.txt" />
        <string key="arguments.2" value="O_RDONLY" />
        <string key="return" value="3" />
</event>
...
```

Fig. 2 Syscall traces extraction example

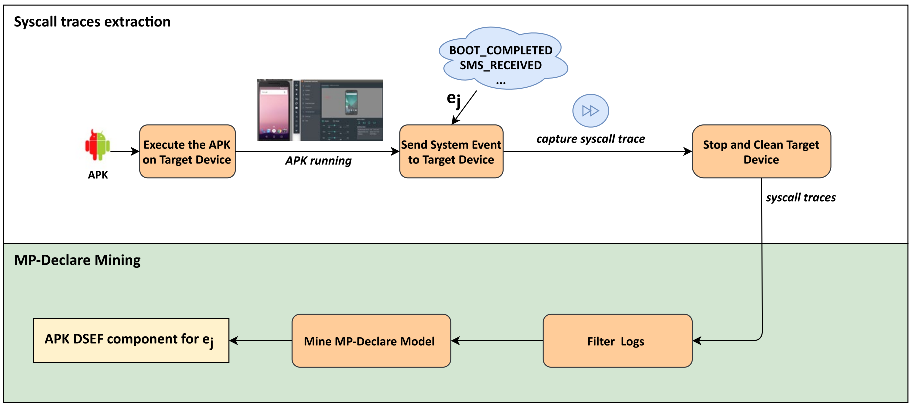

Syscall traces extraction
APK
Execute the APK on Target Device
APK running
BOOT_COMPLETED
SMS_RECEIVED
...
ej
Send System Event to Target Device
capture syscall trace
Stop and Clean Target Device
syscall traces
MP-Declare Mining
APK DSEF component for ej
Mine MP-Declare Model
Filter Logs

Fig. 3 Syscall traces generation and DSEF building process

which the application was executed so that it can accurately capture the right system call flow.

The syscall traces extraction uses an Android device emulator[2]. to generate the syscall traces from an APK. Figure 3 (upper side) describes the steps of the syscall traces extraction for an APK and a single AE (e_j). With this procedure, we capture, for an APK, the syscall traces obtained in response to all the AEs described in the background section. The process starts when the APK is installed and launched on a cleaned device (the disk is cleaned before the process starts). The trace capturing starts when the event e_j is sent to the device and stops when the APK state becomes stable. For this reason, each captured trace

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

is always related to a single AE according to several studies (Jiang and Zhou, 2013) showing that the malicious behavior is usually activated by a single system event.

When we need to build a DSEF model of an APK application, for each AE, we generate n traces by executing n runs of the application. Instead, when we need to build a DSEF model of a malware family, for each AE, we run a total number n of different APKs infected with the malware family. In all cases, when a trace is captured, the device is stopped, the disk is cleaned and the APK is newly installed to ensure that each run is not influenced by the previous one and that the initial conditions are always the same. Both for an APK and for a malware family, we obtain a log for each AE. From each of these logs, we mine an MP-Declare model representing their behavior when stimulated with the corresponding AE. Each of these models represents a component (a sub-model) of the DSEF model of the APK or of the malware family. Notice that the traces can continuously be generated to include running changes of APK and malware families.

4.1.2 MP-Declare mining

The MP-Declare mining task is performed to obtain the DSEF model of an APK and of a malware family. As already mentioned, each DSEF model is composed of DSEF components and each component is obtained for a specific AE. Figure 3 (lower side) describes the process for the extraction of an APK DSEF component. The input log contains the syscall traces captured, in the previous step, from the APK for a specific AE. The log is mined using the MP-Declare mining presented in (Leno et al., 2020). The so discovered models contain MP-Declare rules that describe the typical behavior of the considered APK in response to a given AE. Similarly, referring to the DSEF of a malware family, the input log is composed of the traces obtained by several applications infected with a given malware when each AE occurs. The main idea is that several applications infected with a given malware have a common part corresponding to the malware payload. It is worth noting that, when mining a log generated by the APKs infected with a malware family, the mined model is able to isolate only the behavior (common to all traces) corresponding to the malware action of that family (since the input traces refer to different APKs, the parts specific to each application are discarded).

Before starting the mining phase, the unuseful information (i.e., formatting characters, delimiters, redundant log entries) is filtered out, while some of the attributes associated with the session (e.g., the application run ID and UUID) are attached to the syscall occurrences together with attributes characteristic of the syscall (e.g., its name and timestamp, its ordered list of arguments and the identifier of its requesting process). This information generates the payload for each log event and is used in the process mining step to obtain the data (activation and correlation) conditions. Successively, the traces are filtered using the Gaussian distribution: all the traces with a size outside the 80th percentiles are removed since there is a high probability that, in these cases, the AE triggered a premature termination of the APK by the operating system and the generated trace is trivial.

Specifically, the 80th percentile threshold was determined to filter out the majority of traces generated by abnormal events or circumstances (e.g., application shutdown due to errors or illegal access, out of stack or heap memory, segmentation faults, or other situations leading to unrepresentative traces). We investigated the log size distributions of a sample of 10 APKs for each family analyzed (i.e., 270 APKs in total). This analysis revealed that most of the unrepresentative traces concentrate at the extremes of the distribution of the log sizes. The conservative choice of the 80th percentile implies filtering out a small percentage of correct traces thus favoring correctness over dataset size (which, however, can still be increased by adding more application executions).

We can give now a formal definition of a DSEF (of an APK and of a malware family).

Definition 2 **(DSEF model of an APK** *a***)** The $DSEF_a$ of an APK a is defined as:

$$DSEF_a = \{MPD_{a_1}, \dots, MPD_{a_k}\},$$

where:

- MPD_{a_j} is the MP-Declare model mined from traces $\{t_{j_1}, \dots, t_{j_n}\}$ of a when stimulated with the j-th AE;
- t_{j_i} is the execution trace captured from the i-th execution of a when stimulated by the j-th AE, with $i \in [1, n], j \in [1, k]$;
- n is the number of traces captured from a per AE;
- k is the number of the AEs sent to a ($k = |E|$);
- E is the set of the k AEs sent to each application.

Definition 3 **(DSEF model of a malware family** *M***)** The *DSEF* of M is defined as:

$$DSEF(M) = \{MPD_{M_1}, \dots, MPD_{M_k}\},$$

where:

- MPD_{M_j} is the MP-Declare model representing the behavior of M when stimulated with the j-th AE, mined from traces $\{t_{j_1}, \dots, t_{j_n}\}$;
- A_M is a set of applications infected by a malware belonging to the malware family M;
- t_{j_i} is the execution trace captured from the i-th application in A_M when stimulated by the j-th AE, with $i \in [1, n], j \in [1, k]$ and $n = |A_M|$;
- k is the number of the AEs sent to the applications in A_M ($k = |E|$).

Looking at the above definitions, we can observe that the DSEF model of an APK is computed, for each AE, using n traces obtained by executing that APK several times. Differently, the DSEF model of a malware family is computed by executing a single run for each application since the malicious behavior is the part shared by all the applications infected with that malware. Notice that when we generate the model of an APK, the k activating events are sent to the same application (executed n times). Conversely, to mine the model of a malware family M, the k events are sent to a set of different applications infected with the malware family M.

4.2 Classifier training

We can evaluate whether an APK application is infected with a certain malware, by querying a classifier trained with information about the distances between the DSEF models of trusted and infected APK applications and the DSEF models of the malware families.

To this aim, we need a notion of distance between DSEF models and, therefore, a notion of distance $\sigma(MPD_l, MPD_s)$ between two MP-Declare models MPD_l and MPD_s.

Definition 4 (Distance between MP-Declare models) The distance between two MP-Declare models MPD_l and MPD_s is defined as:

$$\sigma(MPD_l, MPD_s) = 1 - \frac{\sum_{h=1}^{m} \sigma(C_{l_h}, C_{s_h})}{|E_l| + |E_s| + \sum_{h=1}^{m} \sigma(C_{l_h}, C_{s_h})},$$

where:

- C_{l_h} and C_{s_h}, $h \in [1, m]$ are the m constraints of the same type, with the same activation and the same target present in both models and $\sigma(C_{l_h}, C_{s_h})$ represents the tree edit distance between the expression trees of the conjunction of activation and correlation conditions of constraints C_{l_h} and C_{s_h};
- E_l and E_s are the sets of constraints present, respectively, only in model MPD_l and in model MPD_s.

Note that the tree edit distance between expression trees has been evaluated as shown in (Pawlik and Augsten, 2016) and normalized as described in (Li, 2011). According to the above definition, similar MP-Declare models have a distance close to zero (meaning that the models have the same constraints and data conditions), while a higher distance (close to one) means that the MP-Declare models are very different (they have different data conditions and constraints).

Definition 5 (Distance between DSEF models) The distance between two DSEF models $DSEF_A$ and $DSEF_B$ is defined as:

$$\sigma(DSEF_A, DSEF_B) = \frac{\sum_{i=1}^{k} \sigma(MPD_{A_i}, MPD_{B_i})}{k},$$

where k is the number of the AEs used to build the two DSEF models.

The above notion of distance is used to build, for a given APK and for a malware family, a distance matrix having in the columns all the DSEF components (one for each AE) of the APK and in the rows the corresponding DSEF components of the malware family. The cells of the matrix contain the distances between corresponding DSEF components. The distance matrices are used to build a classifier able to discriminate between trusted and infected applications and to indicate, for the infected ones, the type of infection. Two types of classifications are considered in this study: binary classification and multinomial classification. The binary classification builds a classifier for each malware family able to evaluate if an application is infected or not with it. To train the binary classifier, for a given malware family M, we need the following inputs: (i) the DSEF model of malware M; (ii) the DSEF models of a certain number of applications infected with M (for compactness, we indicate this number as d in the following); (iii) the DSEF models of d trusted applications. The samples used to train the classifier are all the distances belonging to all the distance matrices obtained comparing the DSEF model of malware M with all the DSEF models of all trusted and infected applications (we have $2 * d$ matrices), labeled as “trusted” and “infected”, respectively.

Concerning the multinomial classification, a single classifier is trained for all families. The procedure is similar to the one described in the binary case. In this case, if we have

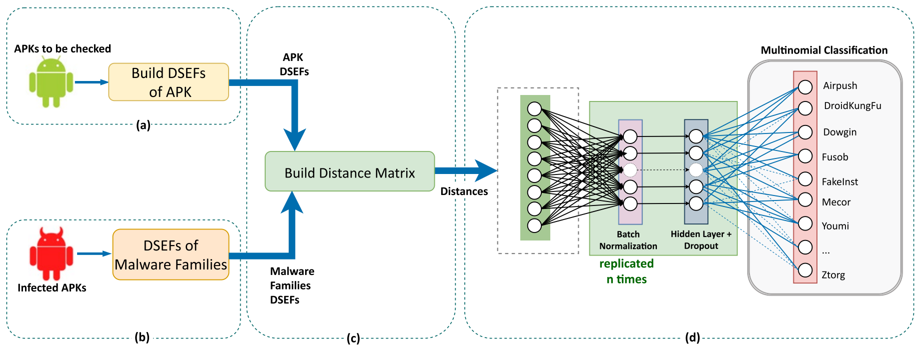


Fig. 4 The malware detection approach (multinomial classification)

r malware families, we have to build $2 * d * r$ distance matrices, and the labeling is not boolean anymore, but it specifies if the application is trusted or infected with a certain malware.

In this study, we use different types of classifiers: Decision Tree (J48), Random Forest (RF), Convolutional Neural Network (CNN), and Dense Neural Network (DNN). In Fig. 4, we show the multinomial classification with DNN. The overall DNN architecture we use is composed of:

- *Input layer*: The entry point of the network, with one node for each considered sample (labeled distance);
- *Batch Normalization layer*: Improves the training of deep feed-forward neural networks;
- a variable number of *Hidden layers*: Composed of artificial perceptrons, having as output the weighted sum of their inputs;
- a variable number of *Dropout layers*: Aiming at reducing over-fitting by including a regularization technique (one dropout layer always follows each hidden layer);
- *Output layer*: Composed of a fully connected layer and a *softmax* function and producing the final classification outcome.

4.3 Malware detection

When the classifier has been trained, it can be queried with new applications to determine if they are trusted or infected. Fig. 4 shows the following main steps of the proposed malware detection approach:[3]

- In Fig. 4a the DSEF model of the APK application to be checked is built. During this step, the syscall logs of the application to be checked are captured and the captured logs are mined to extract the DSEF model of the APK;

[3] The malware detection is described for the multinomial case. In the binary case, the approach is the same with the difference that we have one classifier per malware family and we have to query each classifier separately.

- In Fig. 4b, the DSEF models of the malware families built during the training step (by mining the syscall traces logs captured from infected APKs) are collected and sent to the subsequent distance computation step;
- The distance matrices are built in the step depicted in Fig. 4c, by evaluating the distance among the DSEF models. Specifically, the distances matrices between the DSEF model of the input APK application and the DSEF models of all known malware families are computed and sent to the trained classifier;
- The last step, depicted in Fig. 4d, performs the actual malware detection. The classifier is queried using the distance matrices to evaluate if the APK under examination is infected with one of the malware families used to train the classifier.

5 Experimental evaluation

In this section, we discuss the experiments carried out to evaluate the effectiveness and the robustness of the proposed approach.

5.1 Research questions

To evaluate the approach, the following four research questions (RQs) need to be answered to support the common goal of validating the performance and the robustness of our malware detection approach:

RQ1 What is the F1-score of different types of binary classifiers in detecting each considered malware family?

RQ2 To what extent we can decrease the number of infected applications used to build the DSEF models of the malware families still guaranteeing a reasonable F1-score of the binary classifiers?

RQ3 What is the F1-score of the binary classifiers when using different obfuscation techniques on the malware source code?

RQ4 What is the F1-score of multinomial classification in detecting each considered malware family?

5.2 Dataset description

The experiments are performed on a new dataset built according to the process described in Sect. 4.1.1.

The dataset is composed of a number of applications infected with the malware families reported and described in Table 1 (the table reports, in the last column, for each malware family (*M*), the number of considered infected applications). The dataset also includes a set of trusted applications balanced with the number of infected applications. This is in line with the existing literature (He and Garcia, 2009; Shimizu et al., 2018) suggesting that, when the minority classes are too underrepresented, to train a neural network effectively is necessary to perform balancing to have a more uniform representation of all classes. Therefore, we used 42 139 infected and 42 139 trusted applications for both the DSEF model generation of the malware families and training, reserving the remaining 42 139 infected and 42 139 trusted applications as a test set for the assessment.

The malware applications were downloaded from known datasets like Genoma (Zhou and Jiang, 2012; Drebin Arp et al., 2014) and ADM (Wei et al., 2017). The trusted applications were collected from the most downloaded applications of Google Play. We selected applications referring to different domains (i.e., health, productivity education, internet, news, traveling, business, communication, lifestyle) in order to generalize our findings.

All the applications were labeled as trusted or infected on the basis of the dataset producers' indications. Moreover, we further checked the application label correctness: (*i*) for trusted applications, the label correctness was verified by checking that both Google and all the anti-malware tools running on the VirusTotal service.[4] agreed that the application was not infected by any known malware family (if this was not the case, the application was filtered out); (*ii*) for malware applications, the label correctness was verified by running all the anti-malware tools on VirusTotal and all the applications that were labeled as not infected by at least five anti-malware were filtered out. As already mentioned, the size of the dataset, after filtering, is 168 556 trusted and infected applications.

5.3 Evaluation settings

The classification performance is evaluated by using the F1-score measure. To compare the F1-score of different experimental configurations, we use the Mann-Whitney and Kolmogorov-Smirnov tests with α fixed to 0.05 to check if the difference in the results obtained with different configurations is statistically significant. The adoption of two types of tests allowed us to gain a stronger internal validity.

RQ1 evaluates the F1-score of the binary classifiers. This means that for each malware family, there is a single classifier able to check if an application is infected with that malware or not. Different types of classification algorithms (J48, RF, DNN, CNN) are evaluated and compared to answer this question. Moreover, we compare the obtained F1-score with the results obtained in (Bernardi et al., 2019) using the SEF approach that does not consider data.

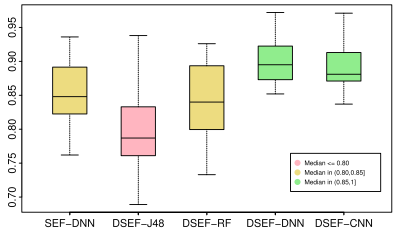
To answer RQ2, we need to compute the F1-score measure of the binary classifiers trained by creating the DSEF models of the malware families with an increasing number of infected applications. To accomplish this goal, we look at the boxplots of the F1-score distributions for different values of the cardinality of the set of infected applications.

RQ3 investigates the impact of the obfuscation techniques (Dong et al., 2018) on the binary classifiers' F1-score. To this aim, several obfuscation techniques are considered and, for each technique, the F1-score is computed. We compare the F1-score median value obtained with the obfuscation techniques with the one obtained by using plain malware applications.

Finally, RQ4 evaluates the F1-score of a multinomial classifier jointly trained on applications infected with all the malware families. To this aim, the F1-score measures were computed for each class (i.e., for each malware family) to verify if the approach is suitable to be adopted for multinomial classification. The classifiers used for answering RQ2, RQ3, and RQ4 are built using the best classification algorithm derived from RQ1.

[4] https://www.virustotal.com/gui/home/upload

Malware Family	DSEF				SEF
	J48	RF	DNN	CNN	DNN
Airpush	0,834	0,871	**0,942**	0,924	0,894
DroidKungFu	0,820	0,904	**0,972**	0,971	0,936
Dowgin	0,902	0,915	0,922	**0,929**	0,885
Fusob	0,938	0,926	**0,958**	0,947	0,844
FakeInst	0,782	0,853	**0,942**	0,929	0,904
Mecor	0,844	0,925	**0,923**	0,913	0,901
Youmi	0,905	0,912	**0,932**	0,924	0,846
Kuguo	0,787	0,858	**0,925**	0,912	0,899
FakeDoc	0,777	0,902	**0,921**	0,913	0,871
FakePlayer	0,809	**0,900**	0,895	0,883	0,802
FakeTimer	0,783	0,831	**0,909**	0,898	0,822
FakeUpdates	0,782	0,887	**0,902**	0,888	0,849
Finspy	0,755	0,875	0,888	0,881	**0,896**
Fjcon	0,807	0,803	**0,890**	0,873	0,792
Fobus	0,723	0,882	**0,901**	0,894	0,848
GingerMaster	0,913	0,791	**0,896**	0,878	0,891
GoldDream	0,778	0,827	**0,886**	0,879	0,774
Gopro	0,718	0,798	0,878	0,875	**0,892**
Ksapp	0,689	0,801	**0,861**	0,851	0,848
Kyview	0,789	0,766	0,880	0,877	**0,888**
Leech	0,802	0,813	0,884	0,875	**0,891**
Lnk	0,767	0,742	**0,853**	0,852	0,762
Lotoor	0,851	0,822	**0,852**	0,837	0,845
Minimob	0,712	0,733	**0,854**	0,84	0,815
Winge	0,721	0,739	**0,867**	0,855	0,823
Zitmo	0,832	0,840	**0,868**	0,869	0,811
Ztorg	0,702	0,771	**0,864**	0,862	0,832



Comparisons	MW Test	KS Test
DSEF-DNN vs DSEF-CNN	0.3197●	$p > 0.05$
DSEF-DNN vs DSEF-J48	2.4e−7●	$p < 0.05$
DSEF-DNN vs DSEF-RF	0.0006●	$p < 0.05$
DSEF-DNN vs SEF-DNN	0.0005●	$p < 0.05$

Fig. 5 F1-score of the binary classifiers (left) and F1-score distribution with respect to the classification method with Mann–Whitney and Kolmogorov–Smirnov tests against the best classifier (right)

5.4 Evaluation results

Here, we report and discuss, for each considered RQ, the obtained results.

5.4.1 RQ1: What is the F1-score of different types of binary classifiers in detecting each considered malware family?

The table on the left side of Fig. 5 shows the F1-score of different binary classifiers in detecting each considered malware family. In particular, in the table, we report, for each malware family, the F1-score obtained by using different classification algorithms (J48, RF, DNN, CNN). Moreover, the table shows (last column) the baseline results obtained using the approach proposed in (Bernardi et al., 2019) that exploits a simpler model based only on control flow and ignores data payloads. For this comparison, we replicated the experiments using the SEF distance as defined in (Bernardi et al., 2019) and performed the classification using the DNN classifier, which has been proven to yield the best results.

For all families (with the exception of Finspy, Gopro, Kyview, and Leech) the DSEF-based approach has a higher F1-score (highlighted in bold) with respect to SEF. The table shows that for all the classification algorithms, we obtained similar results, but DNN generally gives the best results. The best F1-score is obtained for the malware family DroidKungFu with DSEF and DNN. In this case, the F1-score is equal to 0.972. The improvement obtained by using DSEF is confirmed by looking at the F1-score distribution shown in Fig. 5 (right side): the best classifiers for almost all families are based on DSEF and DNN. The results also highlight that DSEF-DNN has a statistically significant difference with respect to DSEF-J48, DSEF-RF, and SEF-DNN (both tests have a $p - value$ lower than $\alpha = 0.05$).

Malware Families	F1-score(10)	F1-score(50)	F1-score(100)	F1-score(200)	F1-score(400)	F1-score(500)	F1-score(all)
Airpush	0,692	0,838	0,826	0,858	0,887	0,905	0,942
DroidKungFu	0,711	0,739	0,927	0,859	0,818	0,901	0,972
Dowgin	0,744	0,738	0,893	0,850	0,862	0,882	0,922
Fusob	0,541	0,833	0,847	0,853	0,863	0,895	0,958
FakeInst	0,502	0,775	0,73	0,838	0,904	0,913	0,942
Mecor	0,822	0,676	0,833	0,897	0,873	0,887	0,923
Youmi	0,675	0,851	0,878	0,884	0,894	0,895	0,932
Kuguo	0,662	0,757	0,802	0,862	0,912	0,915	0,925
FakeDoc	0,692	0,836	0,739	0,826	0,898	0,907	0,921
FakePlayer	0,523	0,808	0,841	0,871	0,886	0,889	0,895
FakeTimer	0,612	0,699	0,891	0,796	0,839	0,908	0,909
FakeUpdates	0,588	0,855	0,785	0,793	0,862	0,877	0,902
Finspy	0,577	0,672	0,74	0,792	0,844	0,886	0,888
Fjcon	0,539	0,738	0,793	0,820	0,887	0,889	0,890
Fobus	0,630	0,793	0,728	0,875	0,871	0,894	0,901
GingerMaster	0,535	0,799	0,749	0,862	0,892	0,885	0,896
GoldDream	0,625	0,744	0,709	0,817	0,824	0,886	0,886
Gopro	0,767	0,691	0,776	0,828	0,863	0,837	0,878
Ksapp	0,640	0,756	0,838	0,841	0,820	0,860	0,861
Kyview	0,565	0,654	0,782	0,850	0,842	0,856	0,880
Leech	0,716	0,687	0,869	0,886	0,831	0,841	0,884
Lnk	0,583	0,743	0,826	0,823	0,784	0,833	0,853
Lotoor	0,601	0,710	0,731	0,777	0,804	0,850	0,852
Minimob	0,783	0,663	0,787	0,857	0,850	0,829	0,854
Winge	0,736	0,844	0,671	0,771	0,868	0,841	0,867
Zitmo	0,753	0,660	0,815	0,838	0,818	0,868	0,868
Ztorg	0,722	0,724	0,864	0,763	0,828	0,848	0,864

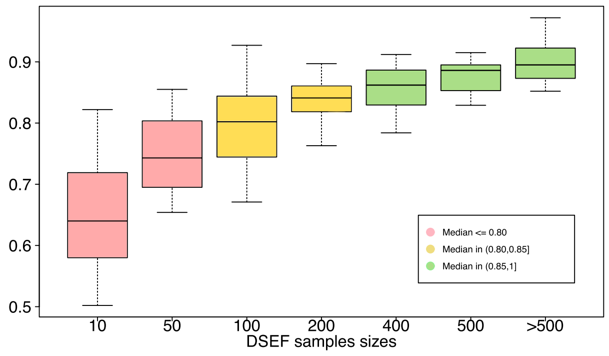
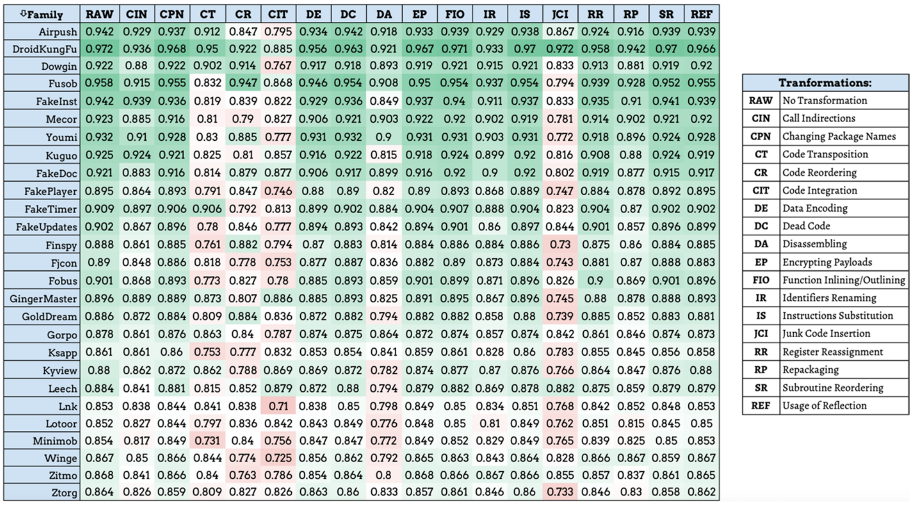


Fig. 6 Impact of the number of infected applications (DSEF sample sizes) used to build the DSEF models of the malware families on the F1-score of the DNN binary classifier

5.4.2 RQ2: To what extent we can decrease the number of infected applications used to build the DSEF models of the malware families still guaranteeing a reasonable F1-score of the binary classifiers?

The F1-score of the DNN binary classifier is further investigated by evaluating the F1-score obtained by decreasing the number of samples used to build the DSEF models of the malware families before training the classifier. The table in Fig. 6 (on top) reports the obtained results for each considered family. In the table, the columns describe the F1-score obtained with an increasing number of samples from 10 to all the ones available (see Table 1).

As expected, the results show for all families a higher F1-score when the number of infected applications used to build the DSEF models of the malware families increases. The best assessment is obtained when the full dataset is considered. The table also shows that, for the F1-score to be acceptable and stable across all families, at least 200 applications are needed for the definition of the DSEF models. The boxplots at the bottom of the figure confirm this trend: up to 200 applications the median value of the F1-score is always under 0.85. Over that threshold, irrespective of the malware family, the F1-score settles on a value of around 0.9. With respect to the best results (when using the full dataset), the



⇩Family	RAW	CIN	CPN	CT	CR	CIT	DE	DC	DA	EP	FIO	IR	IS	JCI	RR	RP	SR	REF
Airpush	0.942	0.929	0.937	0.912	0.847	0.795	0.934	0.942	0.918	0.933	0.939	0.929	0.938	0.867	0.924	0.916	0.939	0.939
DroidKungFu	0.972	0.936	0.968	0.95	0.922	0.885	0.956	0.963	0.921	0.967	0.971	0.933	0.97	0.972	0.958	0.942	0.97	0.966
Dowgin	0.922	0.88	0.922	0.902	0.914	0.767	0.917	0.918	0.893	0.919	0.921	0.915	0.921	0.833	0.913	0.881	0.919	0.92
Fusob	0.958	0.915	0.955	0.832	0.947	0.868	0.946	0.954	0.908	0.95	0.954	0.937	0.954	0.794	0.939	0.928	0.952	0.955
FakeInst	0.942	0.939	0.936	0.819	0.839	0.822	0.929	0.936	0.849	0.937	0.94	0.911	0.937	0.833	0.935	0.91	0.941	0.939
Mecor	0.923	0.885	0.916	0.81	0.79	0.827	0.906	0.921	0.903	0.922	0.92	0.902	0.919	0.781	0.914	0.902	0.921	0.92
Youmi	0.932	0.91	0.928	0.83	0.885	0.777	0.931	0.932	0.9	0.931	0.931	0.903	0.931	0.772	0.918	0.896	0.924	0.928
Kuguo	0.925	0.924	0.921	0.825	0.81	0.857	0.916	0.922	0.815	0.918	0.924	0.899	0.92	0.816	0.908	0.88	0.924	0.919
FakeDoc	0.921	0.883	0.916	0.814	0.879	0.877	0.906	0.917	0.899	0.916	0.92	0.9	0.92	0.802	0.919	0.877	0.915	0.917
FakePlayer	0.895	0.864	0.893	0.791	0.847	0.746	0.88	0.89	0.82	0.89	0.893	0.868	0.889	0.747	0.884	0.878	0.892	0.895
FakeTimer	0.909	0.897	0.906	0.906	0.792	0.813	0.899	0.902	0.884	0.904	0.907	0.888	0.904	0.823	0.904	0.87	0.902	0.902
FakeUpdates	0.902	0.867	0.896	0.78	0.846	0.777	0.894	0.893	0.842	0.894	0.901	0.86	0.897	0.844	0.901	0.857	0.896	0.899
Finspy	0.888	0.861	0.885	0.761	0.882	0.794	0.87	0.883	0.814	0.884	0.886	0.884	0.886	0.73	0.875	0.86	0.884	0.885
Fjcon	0.89	0.848	0.886	0.818	0.778	0.753	0.877	0.887	0.836	0.882	0.89	0.873	0.884	0.743	0.881	0.87	0.888	0.883
Fobus	0.901	0.868	0.893	0.773	0.827	0.78	0.885	0.893	0.859	0.901	0.899	0.871	0.896	0.826	0.9	0.869	0.901	0.896
GingerMaster	0.896	0.889	0.889	0.873	0.807	0.886	0.885	0.893	0.825	0.891	0.895	0.867	0.896	0.745	0.88	0.878	0.888	0.893
GoldDream	0.886	0.872	0.884	0.809	0.884	0.836	0.872	0.882	0.794	0.882	0.882	0.858	0.88	0.739	0.885	0.852	0.883	0.881
Gorpo	0.878	0.861	0.876	0.863	0.84	0.787	0.874	0.875	0.864	0.872	0.874	0.857	0.874	0.842	0.861	0.846	0.874	0.873
Ksapp	0.861	0.861	0.86	0.753	0.777	0.832	0.853	0.854	0.841	0.859	0.861	0.828	0.86	0.783	0.855	0.845	0.856	0.858
Kyview	0.88	0.862	0.872	0.862	0.788	0.869	0.869	0.872	0.782	0.874	0.877	0.87	0.876	0.766	0.864	0.847	0.876	0.88
Leech	0.884	0.841	0.881	0.815	0.852	0.879	0.872	0.88	0.794	0.879	0.882	0.869	0.878	0.882	0.875	0.859	0.879	0.879
Lnk	0.853	0.838	0.844	0.841	0.838	0.71	0.838	0.85	0.798	0.849	0.85	0.834	0.851	0.768	0.842	0.852	0.848	0.853
Lotoor	0.852	0.827	0.844	0.797	0.836	0.842	0.843	0.849	0.776	0.848	0.85	0.81	0.849	0.762	0.851	0.815	0.845	0.85
Minimob	0.854	0.817	0.849	0.731	0.84	0.756	0.847	0.847	0.772	0.849	0.852	0.829	0.849	0.765	0.839	0.825	0.85	0.853
Winge	0.867	0.85	0.866	0.844	0.774	0.725	0.856	0.862	0.792	0.865	0.863	0.843	0.864	0.828	0.866	0.867	0.859	0.867
Zitmo	0.868	0.841	0.866	0.84	0.763	0.786	0.854	0.864	0.8	0.868	0.866	0.867	0.866	0.855	0.857	0.837	0.861	0.865
Ztorg	0.864	0.826	0.859	0.809	0.827	0.826	0.863	0.86	0.833	0.857	0.861	0.846	0.86	0.733	0.846	0.83	0.858	0.862

Tranformations:	
RAW	No Transformation
CIN	Call Indirections
CPN	Changing Package Names
CT	Code Transposition
CR	Code Reordering
CIT	Code Integration
DE	Data Encoding
DC	Dead Code
DA	Disassembling
EP	Encrypting Payloads
FIO	Function Inlining/Outlining
IR	Identifiers Renaming
IS	Instructions Substitution
JCI	Junk Code Insertion
RR	Register Reassignment
RP	Repackaging
SR	Subroutine Reordering
REF	Usage of Reflection

Fig. 7 DNN binary classifier F1-score in the presence of obfuscation

Mann-Whitney and Kolmogorov-Smirnov tests applied to the F1-score distributions over the number of infected applications exhibit a statistically significant difference for a number of samples lower than 400 meaning that for a number of samples higher than 400 the classifier performance is very close to the best one. The provided results are useful to size appropriately the dataset and to identify the right trade-off between the effort of building the models and the effectiveness of final end-to-end detection.

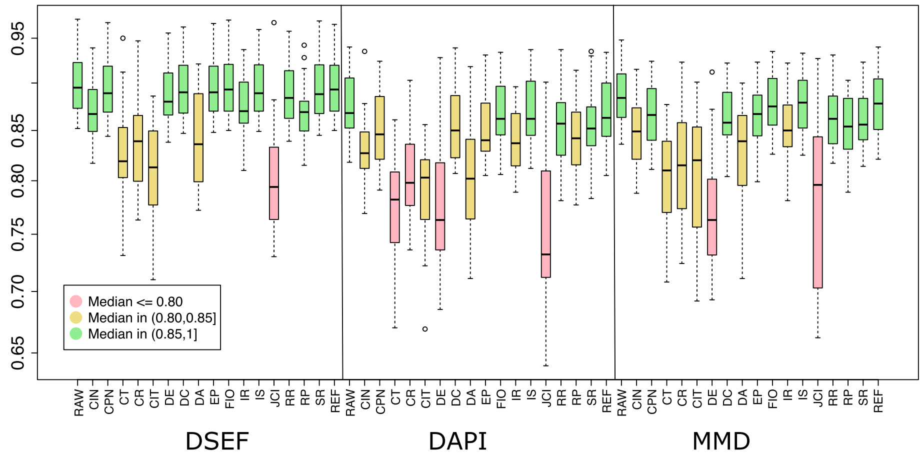
5.4.3 RQ3: What is the F1-score of the binary classifiers when using different obfuscation techniques on the malware source code?

RQ3 investigates the impact of obfuscation techniques on the F1-score of the binary classifiers. This is a very critical aspect due to the habit of several hackers of using obfuscation techniques to make malicious code harder to detect. Therefore, we evaluate the capability of the DNN binary classifier to discover an application infected with a given malware and transformed by a given obfuscation technique.

We applied a set of common transformation techniques (listed in Fig. 7 on the right-hand side) to our samples by using several obfuscation tools (e.g., ADAM (Zheng et al., 2012) and Droidchameleon (Rastogi et al., 2013) and used an open-source obfuscation engine[5] to inject and repackage the obfuscated source code in the APKs.

The table in Fig. 7 reports the F1-score obtained for each malware family for all the considered transformations. The second column (RAW) of the table reports the results obtained when no transformations are applied. We can observe that the score is never lower than 0.71 (obtained for the LnK malware family and the JCI transformation). With respect to the best results (RAW), the Mann-Whitney and Kolmogorov-Smirnov tests exhibit a p-value greater than 0.05 for almost all the source code transformations (no statistically

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



Comparisons	DSEF vs DAPI	DSEF vs MMD
RAW	$p < 0.05$🟢	$p > 0.05$🔴
CIN	$p < 0.05$🟢	$p < 0.05$🟢
CPN	$p < 0.05$🟢	$p < 0.05$🟢
CT	$p > 0.05$🔴	$p > 0.05$🔴
CR	$p < 0.05$🟢	$p > 0.05$🔴
CIT	$p < 0.05$🟢	$p > 0.05$🔴
DE	$p < 0.05$🟢	$p < 0.05$🟢
DC	$p < 0.05$🟢	$p > 0.05$🔴
DA	$p > 0.05$🔴	$p < 0.05$🟢
EP	$p < 0.05$🟢	$p > 0.05$🔴
FIO	$p < 0.05$🟢	$p < 0.05$🟢
IR	$p < 0.05$🟢	$p < 0.05$🟢
IS	$p < 0.05$🟢	$p > 0.05$🔴
JCI	$p < 0.05$🟢	$p > 0.05$🔴
RR	$p < 0.05$🟢	$p < 0.05$🟢
RP	$p < 0.05$🟢	$p < 0.05$🟢
SR	$p < 0.05$🟢	$p < 0.05$🟢
REF	$p > 0.05$🔴	$p > 0.05$🔴

Fig. 8 Comparison, including KS Test, of F1-scores of different classifiers using alternative obfuscation techniques

significant difference) except for CIT, CR, CT, DA and, in the worst case, for JCI (see box-plot distributions at the bottom of Fig. 7).

Some transformations like CIT and JCI are harder to deal with since they inject additional behavior that is not malicious and could alter the relationships mined in the training phase. These techniques are very sophisticated since they alter the syscall sequences, which is difficult to obtain with an automatic process and requires complex malware dissection and manual transformations. However, even in these worst cases, the proposed approach exhibits good levels of robustness achieving a median F1-score value of 0.82 for CIT and 0.78 for JCI.

Figure 8 shows the box plots of the F1-score values for all the considered transformations obtained by using the DSEF models and two alternative approaches respectively called DroidApiMiner (DAPI) (Aafer et al., 2013) and MaMaDroid (MMD) (Onwuzurike et al., 2019). Both DAPI and MMD exploit syscalls to identify malicious behaviors but while DAPI is based on the analysis of system call sequences, MMD is based on Markov chains of behavioral models. The figure reports, for both untransformed and transformed applications, the boxplots of the F1-score over the analyzed families.

The figure shows that, in general, all the approaches have similar F1-scores for samples when no transformations are applied ('RAW' columns), with DAPI performing slightly

Table 2 Multinomial classifier confusion matrix

			TRUE CLASSES																										
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
PREDICTED	Airpush	1	7784	1	3	1	4	4	8	9	1	3	1	8	8	1	5	2	8	0	3	5	2	5	4	6	9	4	5
	DroidKungFu	2	3	3917	1	7	2	2	4	3	2	1	8	9	6	0	3	8	5	6	6	3	1	1	0	6	5	6	4
	Dowgin	3	6	2	3863	1	3	8	4	4	4	0	2	5	3	3	1	5	0	0	8	5	4	3	7	3	8	0	8
	Fusob	4	12	0	1	2178	8	4	4	0	6	6	9	9	4	6	1	5	7	4	9	9	0	5	3	6	4	7	6
	FakeInst	5	6	1	3	2	2131	4	6	8	9	2	1	9	6	1	5	5	2	9	6	5	2	4	9	2	2	7	1
	Mecor	6	1	3	0	0	2	1822	8	6	6	1	2	2	2	2	2	6	2	2	0	0	7	1	2	6	1	8	8
	Youmi	7	0	2	3	0	0	7	1302	3	2	7	7	7	4	9	3	4	4	4	3	7	6	7	5	6	5	8	6
	Kuguo	8	2	0	0	1	2	4	7	1201	4	9	8	8	7	0	4	8	8	0	4	3	1	6	8	0	8	1	7
	FakeDoc	9	0	1	1	0	0	4	7	6	1176	5	0	1	9	4	2	7	4	2	6	6	9	3	6	1	9	1	2
	FakePlayer	10	1	1	3	2	1	6	0	1	7	1023	1	7	3	2	0	2	6	0	5	2	8	3	0	1	6	6	1
	FakeTimer	11	8	0	0	1	0	5	7	9	5	6	1007	5	5	1	0	8	6	5	5	4	6	5	7	4	4	3	4
	FakeUpdates	12	2	0	1	0	2	8	8	8	0	4	5	987	1	4	7	9	7	1	0	6	6	3	7	5	4	2	5
	Finspy	13	2	1	2	0	2	3	3	2	6	4	1	5	923	9	8	9	8	1	9	5	7	3	6	0	6	5	2
	Fjcon	14	2	1	2	1	0	7	2	7	3	4	5	8	8	897	7	5	8	7	3	1	3	3	5	9	2	2	8
	Fobus	15	1	0	1	2	1	7	0	3	0	8	1	6	1	8	876	0	1	2	9	8	8	3	3	5	2	8	7
	GingerMaster	16	0	1	0	0	1	0	6	2	3	0	6	8	6	1	9	854	0	3	4	1	3	8	7	8	4	2	7
	GoldDream	17	0	2	1	1	2	0	1	9	1	6	1	0	8	7	0	5	767	7	5	1	6	4	7	7	4	0	3
	Gopro	18	0	0	2	0	0	5	8	5	5	1	8	4	2	4	5	0	5	763	2	6	8	3	7	6	1	2	6
	Ksapp	19	2	2	0	2	0	4	8	6	5	6	9	5	0	1	6	0	9	0	743	0	1	4	3	7	7	9	3
	Kyview	20	1	0	0	1	3	1	2	1	7	6	8	2	4	3	3	7	0	3	3	722	3	2	3	7	7	3	0
	Leech	21	8	1	1	0	0	2	4	4	9	8	6	4	2	6	0	9	5	2	1	8	701	4	6	3	0	9	0
	Lnk	22	0	0	2	1	4	4	6	4	6	4	3	4	1	0	9	3	0	6	6	1	0	690	3	7	7	9	7
	Lotoor	23	0	5	0	0	0	5	8	8	1	3	2	7	3	1	7	7	5	2	3	7	9	3	650	7	9	6	4
	Minimob	24	1	0	1	2	5	2	0	3	0	1	8	7	2	6	9	1	2	9	5	7	7	2	7	633	6	8	5
	Winge	25	0	1	0	1	2	0	3	8	6	4	2	5	7	6	0	3	6	2	8	6	7	1	7	8	621	4	3
	Zitmo	26	1	0	2	0	2	2	1	1	0	6	0	9	1	8	1	9	0	6	6	9	6	4	5	0	8	604	4
	Ztorg	27	2	1	1	1	1	4	2	9	7	0	2	0	1	1	5	2	1	3	0	7	5	5	1	2	1	6	598
	Samples		7845	3943	3894	2205	2178	1924	1419	1330	1281	1128	1113	1131	1027	991	978	983	876	849	862	844	826	785	778	755	750	730	714

worse and DSEF-DNN slightly better than MMD. This is probably due to the simpler underlying model of DAPI that is based only on system call sequences. However, when the code transformations are taken into account, the DSEF-DNN model exhibits significantly greater robustness with respect to both DAPI and MMD that obtain worse performances for several transformations (CPN,CI,CR,CT,DE,DA,EP,IR,JCR). MMD is still able to perform better than DAPI, confirming that its underlying model is able to better capture behavioral aspects that characterize the malicious payload even when the source code is (to some extent) transformed. As a matter of fact, DSEF-DNN is more resistant, in the majority of cases, to source code transformations with respect to both DAPI and MMD. It is also interesting to note that all the approaches are sensitive, with different degrees, to JCI transformations where junk code snippets are manually crafted to inject useless system calls in the malicious behavior. This highlights that the more the model is capable of intercepting complex relationships among the syscalls, the less it will be sensitive to trivial alterations of the system calls executed by the malicious payload.

Figure 8 also reports a quantitative analysis using the KS test to investigate the F1-scores of DSEF, DAPI, and MMD for transformed applications. This analysis confirms that:

- For plain infected applications, when no evasion techniques are applied, all approaches exhibit similar F1-scores with DSEF and MMD providing slightly better performance than DAPI;
- DSEF has better F1-scores for most of the transformations (in 15 out of 18 cases for DAPI and in 9 out of 18 cases for MMD) performing similarly in the remaining cases.

5.4.4 RQ4: what is the F1-score of multinomial classification in detecting each considered malware family?

The performance of the multinomial DNN classifier for each malware family is reported in Table 2 and Fig. 9. The results show that the best F1-score of the classifier is obtained for families containing the highest number of training samples. In particular, the F1-score is never lower than 0.844 (this value is obtained for the Lootor and Winge malware families).

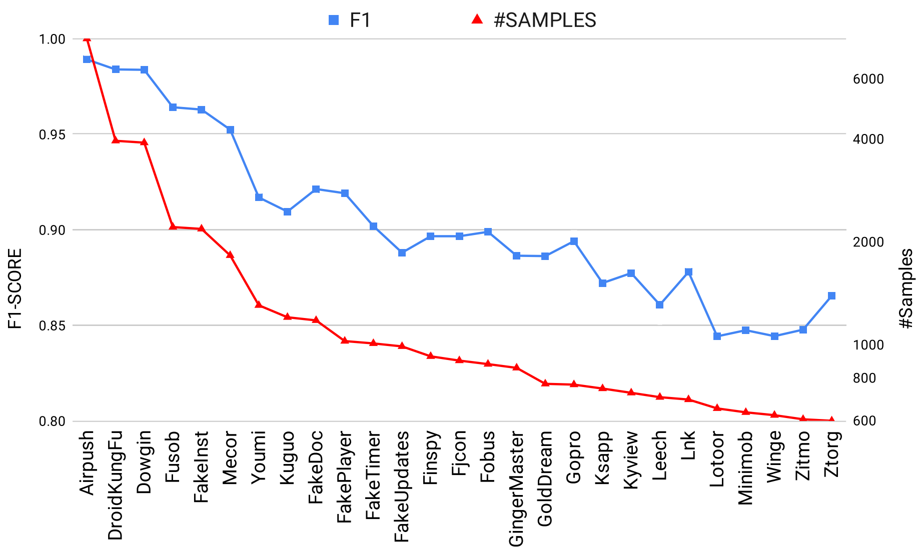


Fig. 9 F1-score by number of training samples

The best F1-score is obtained for Airpush. In Fig. 9, we show the trend of the F1-score for different training sample sizes: with more than 2000 samples, the F1-score is always above 0.9 and its value decreases when the number of available training samples decreases.

6 Threats to validity

Construct validity

The process of syscall traces extraction could be affected by some imprecision since it starts to capture the syscall traces when an AE is sent to the application, which could cause the application to crash. For this reason, the dataset construction requires filtering of both applications and traces. In particular, applications that have inherent problems, e.g., applications that do not run correctly, generate segmentation faults or are unstable need to be filtered out. In addition, since traces are obtained for a given run of an application in response to a given event, in some cases, they can be empty, incomplete, or incorrect if the application exhibits a failure related to unexpected situations. These traces, as already pointed out in Sect. 4.1.2, are eliminated. Both these filtering procedures aim at improving the quality of the dataset and never lead to filtering malicious behavior.

Another issue related to construct validity is the authenticity of the adopted malware family labels. As explained in Sect. 4, the labels are assigned to the applications by the producers. However, this classification could be incorrect. To reduce this risk, a large set of antimalware tools was used to check the applications' labels. In particular, applications that were not classified as infected by at least 5 antimalware tools were excluded. This step is important to ensure high-quality datasets where malware family labels are reliable.

Another threat to construct validity is connected to how traces were generated from the infected applications. It could happen that none of the system events is able to unleash the malware logic encoded in the payload. In this case, the proposed approach cannot be applied to detect such a family. However, this circumstance is revealed during the training phase by studying the distances between trusted and infected applications: when they are not significantly different, for all the system events, the malware should be considered as not detectable.

External validity To make the obtained results generalizable, we used a large set of malware families. Moreover, the malware families considered in this study are quite different in terms of goals and producers. However, a larger number of families could be useful to make the results more generalizable.

Internal validity The obtained results are dependent on the classification algorithm adopted and, to investigate this dependence, we have studied four different classification algorithms using both decision tree-based approaches (J48, RF) and neural networks (CNN and DNN).

7 Conclusion

This study proposes and tests an approach based on data-aware process discovery able to identify infected applications and the type of infection. To this aim, a behavioral model called DSEF is mined from a set of syscall traces coming from trusted applications and applications infected with a given malware. The classification is performed using binary and multinomial classifiers. The evaluation has been performed by considering 27 malware families and a dataset containing more than 160 000 trusted and infected applications. The obtained results clearly show an improved performance over the baseline (Bernardi et al., 2019) of both binary and multinomial classifiers (with the best results obtained using DNN) and very good robustness to basic obfuscation techniques. Even in the worst case of manually transformed source code, the approach exhibits good levels of robustness to behavior-preserving code transformations.

In future work, we will investigate the use of behavioral models specified using both procedural and declarative patterns. These richer representations could improve their capability of characterizing malicious behavioral fingerprints. The idea of modeling malware behavior as a process can be extended to the entire malware analysis domain. To this aim, we would need to generalize the concepts of AE and DSEF that are quite specific to the mobile context. More complex and larger neural-based classifiers based on transformers and Graph Neural Networks will be experimented with to learn the characteristics of trusted and infected applications directly from the MP-Declare models' structure. This will allow us to identify complex relations that cannot be pinpointed by the metric approach based on the distance between models.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection were performed by Mario Luca Bernardi whereas the analysis were performed by Marta Cimtile and Fabrizio Maria Maggi. The first draft of the manuscript was written by Mario Luca Bernardi and Marta Cimtile and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding The authors did not receive support from any organization for the submitted work.

Availability of data and material Not applicable.

Declarations

Conflict of interest The authors have no conflict of interest to declare that are relevant to the content of this article.

Ethical approval The authors declare that they are compliant with the ethical standards of Machine Learning journal, have no conflicts of interest and give their informed consent.

Consent to participate The authors give their consent to partecipate.

Consent for publication The authors give their consent for publication.

Code availability Not applicable.

References

Aalst van der, W. M. P. (2016). *Process Mining - Data Science in Action, Second Edition. Springer 2016, ISBN 978-3-662-49850-7, pp. 3-452.*

Aafer, Y., Du, W., Hin, Y. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In T. Zia, A. Zomaya, V. Varadharajan, & M. Mao (Eds.), *Security and privacy in communication networks* (pp. 86–103). Springer International Publishing.

Alazab, M. (2015). Profiling and classifying the behavior of malicious codes. *Journal of Systems and Software, 100*, 91–102.

Ardimento, P., Aversano, L., Bernardi, M. L., Cimitile, M. (2020). Data-aware declarative process mining for malware detection. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*, pages 1–8. IEEE, 2020.

Arora, A., Garg, S., Peddoju, S. K. (2014). Malware detection using network traffic analysis in android based mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 66–71.

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

Austin, T. H., Filiol, E., Josse, S., Stamp, M. (2013). Exploring hidden markov models for virus analysis: A semantic approach. In *2013 46th Hawaii International Conference on System Sciences*, pages 5039–5048.

Bernardi, M. L., Cimitile , M., Di Lucca, G. A., Maggi, F. M. (2012). 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*, pages 56–65.

Bernardi, M. L., Cimitile, M., Distante, D., Martinelli, F., Mercaldo, F. (2019). Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security, 18*(3), 257–284.

Bezerra, F., Wainer, J., van der Aalst, W. M. P. (2009). Anomaly detection using process mining. In *Enterprise, Business-Process and Information Systems Modeling* (pp. 149–161). Springer.

Canfora, G., Medved, E., Mercaldo, F., Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, pages 13–20, New York, NY, USA. ACM.

Chomicki, J. (1995). Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transaction Database Systems, 20*(2), 149–186.

Christodorescu, M., Jha, S., Seshia, S. A., Song, D., Bryant, R. E. (2005). Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S P'05)*, pages 32–46.

Di Ciccio, C., Marrella, A., Russo, A. (2015). Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches. *Journal of Data Semantics, 4*(1), 29–57.

Ding, Y., Dai, W., Zhang, Y. (2014). Control flow-based opcode behavior analysis for malware detection. *Computers & Security, 44*, 65–74.

Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K. (2018). Understanding android obfuscation techniques: A large-scale investigation in the wild. In Raheem Beyah, Bing Chang, Yingjiu Li, & Sencun Zhu (Eds.), *Security and privacy in communication networks* (pp. 172–192). Springer International Publishing.

Fedler, R., Schütte, J. (2013). On the effectiveness of malware protection on android an evaluation of android antivirus.

Feng, Y., Anand, S., Dilling, I., Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA. ACM.

He, H., Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering, 21*, 1263–1284.

Jang, J., Brumley, D., Venkataraman, S. (2011). Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA. ACM.

Jeong, Y., Lee, H., Cho, S., Han, S., Park, M. (2014). A kernel-based monitoring approach for analyzing malicious behavior on android. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1737–1738, New York, NY, USA. ACM.

Jiang, X., Zhou. Y. (2013). *Android malware*. Springer Briefs in Computer Science.

Karbab, E. B., Debbabi, M, Derhab, A., Mouheb, D. (2018). Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation, 24*, S48–S59.

Karim, Md. E., Walenstein, A., Lakhotia, A., Parida, L. (2005). Malware phylogeny generation using permutations of code. *Journal in Computer Virology, 1*(1–2), 13–23.

Leno, V., Dumas, M, Maggi, F. M., La Rosa, M., Polyvyanyy, A. (2020). Automated discovery of declarative process models with correlated data conditions. *Information Systems, 89*, 101482.

Li, Q., Li, X. (2015). Android malware detection based on static analysis of characteristic tree. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 84–91, 2015.

Maggi, F. M., Di Ciccio, C., Di Francescomarino, C., Taavi, K. (2018). Parallel algorithms for the automated discovery of declarative process models. *Information Systems*, 74(Part):136–152.

Marioconti, E. , Onwuzurike, L., Andriotis P., De Cristofaro, E., Ross, G., Stringhini, G. (2019). Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2).

Mobile threat report (2016). https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf, last visit 26 February 2016.

Oak, R., Du, M., Yan, D., Takawale, H. C., Amit, I. (2019). Malware detection on highly imbalanced data through sequence modeling. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, AISec'19, page 37–48, New York, NY, USA. Association for Computing Machinery.

Ostovar, A., Leemans, S., La Rosa, M. (2020). Robust drift characterization from event streams of business processes. *ACM Transaction Knowledge Discovery Data, 14*(3), 30.

Pawlik, M., Augsten, N. (2016). Tree edit distance: Robust and memory-efficient. *Information System, 56*, 157–173.

Pesic, M., Schonenberg, H., van der Aalst, W. M. P. (2007). Declare: Full support for loosely-structured processes. *In EDOC, 2007*, 287–300.

Pnueli, A. (1977). The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society.

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

Rastogi, V., Chen, Y., Jiang, X. (2013). Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA. ACM.

Shimizu, R., Asako, K., Ojima, H., Morinaga, S., Hamada, M., Kuroda, T. (2018). Balanced mini-batch training for imbalanced image data classification with neural network. In *2018 First International Conference on Artificial Intelligence for Industries (AI4I)*, pages 27–30.

Su, X., Zhang, D., Li, W., Zhao, K. (2016). A deep learning approach to android malware feature learning and detection. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 244–251.

Talha, K. A., Alper, D. I., Aydin, C. (2015). Apk auditor: Permission-based android malware detection system. *Digital Investigation, 13*, 1–14.

Taymouri, F., La Rosa, M., Dumas, M., Maggi, F. M. (2021). Business process variant analysis: Survey and classification. *Knowledge Based System, 211*, 106557.

Wei, T., Mao, C., Jeng, A. B., Lee, H., Wang, H., Wu, D. (2012). 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, pages 1251–1258, Washington, DC, USA. IEEE Computer Society.

Wei, F., Yuping, L., Sankardas, R., Xinming, O., Wu, Z. (2017). Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’17)*, pages 252–276, Bonn, Germany. Springer.

Wu, D., Mao, C., Wei, T., Lee, H., Wu, K. (2012). Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69.

Xu, K., Li, Y., Deng, R. H. (2016). Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security, 11*(6), 1252–1264.

Yujian, L., Chenguang, Z. (2011). A metric normalization of tree edit distance. *Frontiers of Computer Science, 5*(1), 119.

Zhang, J., Zou, F., Zhu, J. (2018). Android malware detection based on deep learning. In *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, pages 2190–2194.

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

Zhou, Y., Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP ’12, pages 95–109, Washington, DC, USA. IEEE Computer Society.