applied sciences

MDPI

Article

Call Graph and Model Checking for Fine-Grained Android Malicious Behaviour Detection

Giacomo Iadarola [1,*,†], Fabio Martinelli [1,†], Francesco Mercaldo [1,2,*,†] and Antonella Santone [2,†]

[1] Institute for Informatics and Telematics, National Research Council of Italy, 56124 Pisa, Italy; fabio.martinelli@iit.cnr.it
[2] Department of Medicine and Health Sciences "Vincenzo Tiberio", University of Molise, 86100 Campobasso, Italy; antonella.santone@unimol.it
* Correspondence: giacomo.iadarola@iit.cnr.it (G.I.); francesco.mercaldo@unimol.it (F.M.)
† All authors contributed equally to this work.

Received: 30 September 2020; Accepted: 5 November 2020; Published: 10 November 2020

Abstract: The increasing diffusion of mobile devices, widely used for critical tasks such as the transmission of sensitive and private information, corresponds to an increasing need for methods to detect malicious actions that can undermine our data. As demonstrated in the literature, the signature-based approach provided by antimalware is not able to defend users from new threats. In this paper, we propose an approach based on the adoption of model checking to detect malicious families in the Android environment. We consider two different automata representing Android applications, based respectively on Control Flow Graphs and Call Graphs. The adopted graph data structure allows to detect potentially malicious behaviour and also localize the code where the malicious action happens. We experiment the effectiveness of the proposed method evaluating more than 3000 real-world Android samples (with 2552 malware belonging to 21 malicious family), by reaching an accuracy ranging from 0.97 to 1 in malicious family detection.

Keywords: malware; model checking; formal methods; security; Android; mobile

1. Introduction

Mobile devices, such as smartphones but also smart TVs, wearables and voice assistants, play a fundamental role in our increasingly connected world. As a matter of fact, nowadays we use mobile devices for the most disparate tasks, such as turning on the heating in the house while we are coming back from the office or sharing the path taken by running with people who share the same passion.

Clearly, the amount of information we daily share, even unknowingly, is really impressive. Indeed, 500 million tweets are posted every day and 294 billion emails are sent [1]. Four petabytes of data are created alone by Facebook users. Moreover, 65 billion messages are estimated on WhatsApp [2]. Following this data prediction, in 2025 we could estimate a trend of 463 exabytes of daily information: the equivalent of 212,765,957 DVDs per day [3]. Obviously, this great amount of private and sensitive information is of interest for attackers and, generally speaking, malicious users that are developing every day more and more aggressive techniques to exfiltrate data from our mobile devices. Even critical infrastructure are afflicted by malicious attacks, such as military [4], government [5] and healthcare [6]: these contexts are really crucial because attacks can have direct consequences also on human lives.

Typically, malicious code developers focus on the platform that allows them to have the greatest attack surface. Indeed, the Android operating system, since 2017 the most widespread operating system in the world [7,8] has largely attracted the interest of malicious developers. According to the

security experts of AV-Test, the cumulative number of Android malware was around 480 thousand samples in March 2020, with "trojans" as the most common type of malware [9].

The current techniques employed in commercial use to detect malicious mobile application are the same techniques adopted for malicious computer software (i.e., signature-based approach). In practice, the security analyst receives an application to analyse, and through a typically manual inspection process, detects the payload or the malicious code of the application. Subsequently, a signature will be generated from this code, which will be used to detect the same malicious code on user's devices. The main problem with this paradigm is that the malicious applications arrive in the labs of security experts, typically, when they are already widespread on the user's devices. Moreover, the attackers can generate obfuscated versions of the original malware and then bypass the antimalware checks. Trivial obfuscation and perturbation techniques, such as code reordering or junk code insertion, can effectively change the signature of malware, which will be no longer detected by antimalware.

Based on these considerations, there is a need for novel methods for the detection of malware, especially in the Android environment due to its diffusion.

In this paper, we propose an approach exploiting model checking for the detection of malicious families in Android malware. Basically, we represent an Android application as an automaton and, by exploiting a set of Temporal Logic formulae, we consider a formal verification environment to detect whether the behaviour codified in the formulae is exhibited from the automaton. Moreover, we improve the efficiency and precision of the detection by adopting two graphs data structure; we apply static analysis for reducing the search space and also detect the malicious code to a lower level.

In detail, two different representations of the same application are considered. The first automaton is built starting from the Control Flow Graph (CFG) containing all the bytecode instructions; the second one is built starting from the Call Graph (CG), representing the call relationships between methods (and classes) of the application under analysis. For this reason, two different sets of formulas are considered: the first set, related to the automaton built from the CFG, aims to verify whether a certain, possible, malicious action is present into the application, while the second set aims to check if this behaviour is invoked in a certain path of the application.

We demonstrate the effectiveness of the proposed model checking approach in the experimental analysis, obtaining an accuracy ranging from 0.97 to 1 in the evaluation of 2552 malware samples belonging to 21 different Android malware families and 500 more samples of legitimate applications. The malware families exhibit different behaviour, from payload able to extort private information to most recent locker and cypher ransomware.

In the last years, researchers developed several methods aimed to detect malware in the mobile environment [10,11], mainly based on machine learning techniques [12]. The main criticism of these techniques is the lack of explainability and interpretability. Indeed, machine learning models usually output a prediction (i.e., a label) without explaining the reason why some decisions were made [13]. Contrarily, formal methods can offer the opportunity to better understand the rationale behind a Temporal Logic formula, also through the adoption of the counterexample that can help the analyst to understand the reason why a formula is resulting false on a certain automaton [8]. Moreover, with the proposed method, we are able to localise the class, the method and also the instruction into the bytecode symptomatic of the malicious behaviour, reaching a very fine grain in the malware analysis. We summarize the main distinctive points of this work as follows:

- we adopt two different graph data structures (CFGs and CGs) to describe the malware functionalities, using intra- and inter-procedural analysis;
- we exploit the strengths of the two graph data structures and the formal methods to reduce the search space and then mitigate one of the biggest weakness of the static analysis approaches;
- we propose a method able to precisely distinguish between malware and trusted application, and then also classify the malware into families;
- we propose a method able to precisely localize the malicious code in the malware under analysis;

- we support the formal methods approaches as an interpretable approach to address malware classification tasks. Indeed, the Temporal Logic formulae are able to classify malware and also provide precise and soundness explanation of the outcome by using the counter-example;
- the experimental results reach an accuracy of 0.988, on a dataset of more than 3000 real-world Android samples, constituting 2552 malware belonging to 21 malicious families and 500 trusted applications.

The paper proceeds as follows: the next section presents a short background about model checking, in order to introduce relevant notions for the readers who are not familiar with model checking. Section 3 describes the proposed method for malicious family detection in Android environment, while the experimental analysis results performed on a real-world Android dataset are discussed in Section 4. Section 5 reports a comparison with the current state-of-the-art literature and, finally, conclusion and future research works are discussed in Section 6.

2. Background

2.1. Model Checking

Below, background information focused on the model checking largely apply by the proposed approach are provided. In order to resort to model checking, there is the need of a Formal Model (used to abstract the Android application under analysis), a Temporal Logic (aimed to represent a malicious behaviour in mobile environment) and a Formal Verification Environment (aimed to verify whether the behaviour is implemented in the Formal Model representing the Android application under analysis).

In order to obtain a Formal Model, i.e., a formal description of the application under analysis, specification is exploited. The specification exploits a language with a well defined syntax and semantics. The application behaviour is represented as a Labelled Transition System (LTS) formed by nodes and labelled edges aimed to connect the nodes. Basically, the Android application state is shown as a node, differently the transition of the application from a specific state to the consecutive one is shown as a labelled edge.

If the Android application represented in terms of LTS shows an $s \xrightarrow{a} s'$ edge, the application under analysis is able to move from the first state (i.e., s) into the consecutive one (i.e., s') by exploiting the a action, ranging in the sets of $\mathcal{A}$ actions. The LST initial state is the so-called root state. To represent LTS as processes, the Milner's Calculus of Communication Systems [14–16] (CCS) is exploited.

A process in CSS (for instance, p) is defined by exploiting several conditional rules aimed to describe the transition of the automaton (i.e., the Android application under analysis) related to the behaviour expression defining the p process.

CCS considers several operators for the representation of finite processes, describing composition of parallel and the choice between different actions.

The automaton we considered is the standard transition system for the p process and is represented as $\mathcal{S}(p)$. More information are available in [14].

The proposed approach exploits the following CCS operators:

- "$+$": $p + q$ represents a process that exhibit a non-deterministic behaviour either as p or as q;
- "$;$": this CCS operator is considered to formalise the sequentialization related to two different processes. The $p;q$ process is symptomatic that the p process must mandatory terminate before the second process (i.e., q) can perform its execution;
- "$\|$": the $p\|q$ process is aimed to represent the execution of the parallel of the p and q process. This parallel terminates only if the p and the q processes terminate;
- "$DONE$": is related to a process whose task is terminated.

Once a Formal Model is obtained, there is the need of Temporal Logic, to express properties (i.e., the behaviour) in a well defined notation. Temporal Logics [17–20] show several constructs with

the aim to allow to state in a formal way. In the proposed approach, we exploit μ-calculus logic [17]. Below, we describe its syntax. Below, we consider that Z can range on a set of variables, i.e., K and R are able to range over sets of $\mathcal{A}$ actions.

$$\phi \quad ::= \quad \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \vee \phi \mid \phi \wedge \phi \mid [K]\,\phi \mid \langle K \rangle\,\phi \mid \nu Z.\phi \mid \mu Z.\phi$$

The satisfaction of a formula (for instance, ϕ) by a state (for instance, s) of a transition system, denoted by $s \models \phi$, is expressed in the following way:

- all states satisfy `tt` and there is no state satisfying `ff`, as shown in the first two rows of Table 1;
- a state is able to satisfy $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) whether it is able to satisfy ϕ_1 or/ and ϕ_2, as shown in the third and fourth rows of Table 1;
- in Table 1 are shown $[K]\,\phi$ and $\langle K \rangle\,\phi$,representing the modal operators in particular:

  $[K]\,\phi$ is satisfied by a state that, for each performance of an action in K, is able to evolve in a state obeying ϕ.
  $\langle K \rangle\,\phi$ is satisfied by a state that is able to evolve to a state obeying ϕ by executing an action in K.

In Table 1 we show the definition of the satisfaction of a closed formula φ by an s state (indicated as $s \models \varphi$).

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

$p \not\models$	`ff`	
$p \models$	`tt`	
$p \models \varphi \wedge \psi$	iff	$p \models \varphi$ and $p \models \psi$
$p \models \varphi \vee \psi$	iff	$p \models \varphi$ or $p \models \psi$
$p \models [K]\,\varphi$	iff	$\forall p'.\forall \alpha \in K.p \xrightarrow{\alpha} p'$ implies $p' \models \varphi$
$p \models \langle K \rangle\,\varphi$	iff	$\exists p'.\exists \alpha \in K.p \xrightarrow{\alpha} p'$ and $p' \models \varphi$
$p \models \nu Z.\varphi$	iff	$p \models \nu Z^n.\varphi$ for all n
$p \models \mu Z.\varphi$	iff	$p \models \mu Z^n.\varphi$ for some n

where:

- for each n, $\nu Z^n.\varphi$ and $\mu Z^n.\varphi$ are defined as:

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

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

$\mu Z.\phi$ and $\nu Z.\phi$ are related to the fixed point of the formulae, where μZ (νZ) binds free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ (νZ). A formula is considered as closed whether the formula contains no free variables. $\mu Z.\phi$ represents the least fix-point of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one. A transition system T is satisfying a ϕ formula, indicated as $T \models \phi$, if and only if $q \models \phi$, where q represents the initial state of T. A CCS process p satisfies ϕ if $S(p) \models \phi$.

The following abbreviations are exploited (where K is able to range over sets of actions and $\mathcal{A}$ is representing the full set of the actions):

$$\begin{aligned} \langle \alpha_1, \ldots, \alpha_n \rangle \, \varphi &\stackrel{\text{def}}{=} \langle \{\alpha_1, \ldots, \alpha_n\} \rangle \, \varphi \\ \langle - \rangle \, \varphi &\stackrel{\text{def}}{=} \langle \mathcal{A} \rangle \, \varphi \\ \langle -K \rangle \, \varphi &\stackrel{\text{def}}{=} \langle \mathcal{A} - K \rangle \, \varphi \end{aligned}$$

Once the Formal Model and the Temporal Logic are defined, we need a Formal Verification Environment. The Formal Verification Environment is exploited to evaluate whether the property is satisfied on the automata representing the application under analysis. To this aim, we resort to a formal verification environment, i.e., a software considering mathematical reasoning to check if a model satisfies some the properties (expressed in Temporal Logic).

A plethora of different verification techniques are proposed by the research community. The proposed method exploits the model checking one [21]: this technique requires that the properties are expressed in terms of Temporal Logic, where each property is evaluated with the Formal Model (i.e., LTS) model. The verification environment is able to accept a Formal Model and a Temporal Logic property and it outputs "true" if the model satisfies the Temporal Logic property, otherwise "false". The check performed by the formal verification environment (know also as Model Checker) represents an exhaustive search related to the state space aimed to guarantee the termination (as a matter of fact, the model is finite).

2.2. Control Flow Graphs and Call Graph

The CFG [22,23] represents all the paths that might be traversed during program execution, and this kind of graph was widely adopted in malware analysis approaches [24,25]. Each node in the graph represents a statement or a block of code, while the direct edges represent the transition of the program execution flow from one statement to another. A CFG has also two special nodes to control respectively the entry and the exit of the control flow execution. In this paper, we applied an intra-procedural analysis for building the CFG, thus each graph represents the paths within a single method or function. In short, we need the CFG to describe the order in which code statements are executed within a single method, and also the required conditions to take a particular path of execution.

On the other hand, we adopted the CGs to represent the entire functionalities of the application under analysis [26,27]. In a CG, the nodes represent the methods of the application and the direct edges represent their connection (i.e., caller and callee). The CGs are built by looking for the invoke statements in the code, that link one method to another, and describes an inter-procedural analysis.

3. The Method

In the following section, we present the proposed methodology, devoted to Android malicious family detection. The methodology can be split into two steps: the first step aims to identify the Potentially Malicious Application (PMA), while the second step is related to the fine-grained malicious family detection. The two steps were summarized respectively in Figures 1 and 2.

3.1. Potentially Malicious Application Detection Step

As shown in Figure 1, we obtain through a reverse engineering process the Java bytecode for the Android application under analysis (detailed information in Section 4.3).

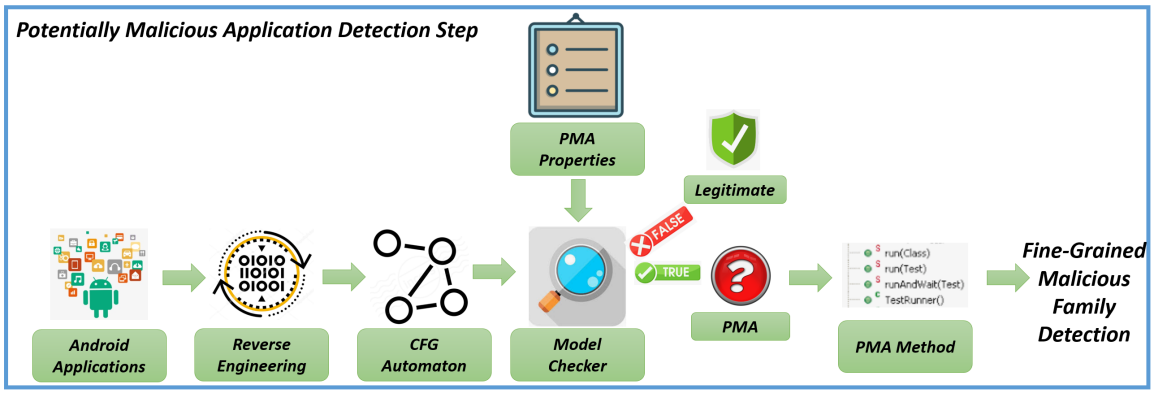

Figure 1. Potentially malicious application detection step.

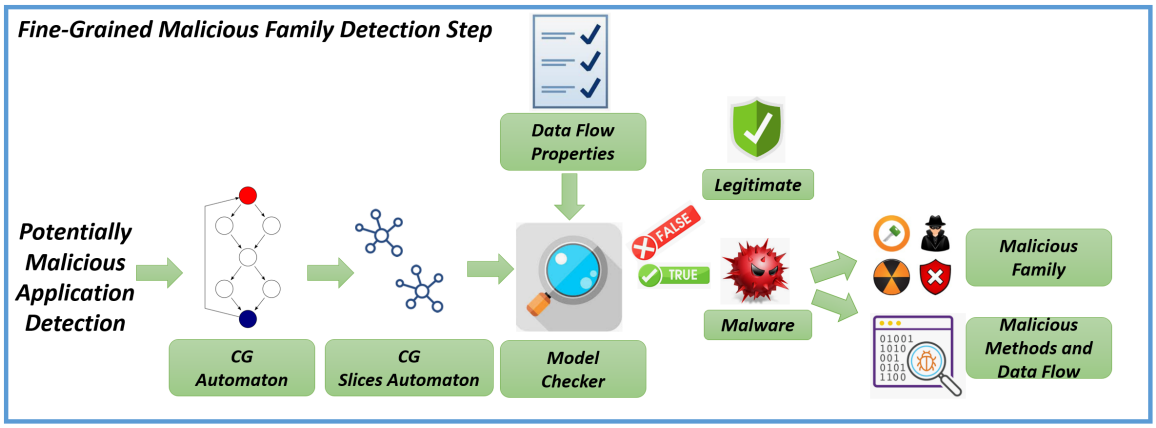

Figure 2. Fine-grained malicious family detection step.

Once the Java byte-code of the installed applications is obtained, we generate a model for each application (i.e., CFG Automaton step in Figure 1). In the following, we explain how we generate the automata, based on the CFG. We represent each method of an Android application as a CCS model. We obtain the CCS models by traducing each java byte-code instruction as a CCS process. In reference [28], more details about the correspondence between java bytecode instructions and CCS process are available.

Relating to the sequential Java bytecode instructions, the following translation is exploited:

$$proc\ x_{current} = instruction.x_{next}$$

where, $x_{current}$ represents the current instruction, x_{next} represents the process related to the next instruction, $instruction$ represents the name of the byte-code instruction. We highlight that in order to define constant definitions, we resort to the syntax of the Concurrency Workbench of New Century [14], the Model Checker we exploit for the experiments. This is the reason why we write $proc\ x = p$, instead of $x \stackrel{\text{def}}{=} p$.

In Listings 1 and 2 are depicted two examples of translation of sequential instructions.

Listing 1: CCS process for Listing 1.

```
1 proc M1 = pop.M2
2 proc M2 = push.M3
3 proc M3 = getfield.nil
```

Listing 2: CCS process for Listing 2.

```
proc N1 = push.N2
proc N2 = invokeinit.N3
proc N3 = invokejavaioFile.N4
proc N4 = getfield.N5
proc N5 = load.N6
proc N6 = dup.N7
proc N7 = pop.nil
```

The instructions related to branches are exploited to alter the execution of the instruction. To represent the branch, as explained in Section 2, the CCS + operator is considered.

For each method of the application, a CCS process is generated. Let us consider *aua* an application under analysis. Considering that the *aua* exhibits n methods, i.e., $F_1, \ldots, F_n$, the *aua* CCS representation has n $M_1, \ldots, M_n$ CCS processes.

Once the CCS processes is generated, where each process is generated for each method of the *aua*, we invoke the Model Checker with a set of properties (PMA Properties in Figure 1) aimed to detect whether the analysed application exhibit a potentially malicious behaviour. The PMA Properties were manually created after code inspection, details are reported in Section 4.2.

Thus, if the Model Checker outputs *true*, this is symptomatic that one of the PMA Properties is verified in at least a CCS process (i.e., a method of the application) and, for this reason, the application under analysis is marked as PMA (as shown in Figure 1) and it is send to the next step, the fine-grained malicious family detection, for a deeper analysis. Considering that we built a CCS process for each method of the application, the PMA Properties are verified on each CCS process and, consequently, we are able to perform the analysis at method grain. For this reason the output of the potentially malicious application detection step is a list of methods potentially malicious for each analysed applications. Otherwise, whether all the CCS processes of the application under analysis are resulting false when the Model Checker checks the PMA properties, the application is labelled as legitimate and the analysis terminates.

This list of PMA with the related methods are sent to the next step, the fine-grained malicious family detection one, shown in Figure 2.

3.2. Fine-Grained Malicious Family Detection Step

While the first step of the methodology aims to detect potentially malicious application by looking for methods that manifest malicious behaviour, the second step takes into account the entire application and looks for malicious sequences of operations to classify the malware into behavioural families.

Most of the time, the harm of an application comes from sequences of methods and operation, instead of a single method itself. Every operation could be legitimate for some functionalities or performances reason. For instance, reading SMS messages can be considered as a legitimate action when performed during a verification step for login into an application, while the operation should be marked as malicious if performed together with others, like reading the information in the background, or sending data without user interaction and permission.

Nevertheless, inter-procedural analyses are tasks requiring a lot of computational time which require deeper checks of all the possible sequence of operations. The number of paths to test grows exponentially, suffering from the path explosion problem, which makes the analysis unfeasible with a brute-force approach. In light of the above considerations, the second step of the methodology aims to address this problem by using the information provided by the first step and completing the classification in a feasible way; while the first step is essential to improve efficiency and reduce false positive, the second one concludes the methodology by performing the classification and by analysing only a subset of methods.

First of all, a static analysis is performed on the potentially malicious application and a CG is generated. The methods are analysed at the code level and all the call/invoke statements are extracted. A node in the graph represented a method and is linked to all the other methods that invokes in the code, and thus to all the other methods to which is passing values and data. The CG represents the entire structure of the application from an execution flow and data point of view. We already had the information on single methods from the first step, and now we extend our knowledge on the application with the interactions between all the methods.

Then, the generated CG is visited and we look for the interactions of pair of nodes, and the paths between them. By using the list of malicious methods provided by the first step, the graph is sliced and many subgraphs are generated, one for each couple of malicious methods linked. This subset of nodes and edges contains all the possible paths only between the malicious methods selected previously, and this operation greatly cut down the number of paths to analyse.

A CSS process is built for each selected path of the graph, and then the Model Checker evaluates if the process exhibits some data flow properties typical of a particular malware family. If so, the application is marked as a malware of that family, and the methods which constitute the malicious path can be manually analysed.

4. Experimental Analysis

The experiment we propose to evaluate the performance of our approach is discussed in this section.

4.1. Dataset

The real-world Android applications considered in the experimental analysis were obtained from three different application repositories. The first repository is composed by freely available samples with ransomware behaviours belonging to 11 diffused malicious families and gathered using the VirusTotal web service [29]. In particular, we consider the following ransomware families: Doublelocker, Koler, Locker, Fusob, Porndroid, Pletor, Lockerpin, Simplelocker, Svpeng, Jisuit and Xbot. Ransomware behaviour in Android environment can basically exhibit two main malicious actions: the first one aimed to lock the device, and the second one is devoted to cipher the user files. Both of these actions ask to the infected user for paying a ransom (typically in bitcon) in order to use their own devices and to access their own files. In particular, the Fusob, Koler, Lockerpin, Locker, Porndroid and Svpeng families exhibit the locker behaviour (i.e., they do not perform any ciphering operation but prevent victims from accessing their devices), the Pletor, Doublelocker and Simplelocker payloads are able to cipher the user files (but when infected with samples belonging to this family users are able to use the device), while samples of the Jisuit and Xbot malicious family exhibits both the locker and cipher typical ransomware behaviours.

The second repository we considered is Drebin [30,31], a widespread collection of malware considered by researchers for the evaluation of malware detection methods, including several Android malicious families. No ransomware samples are comprised in the Drebin dataset.

In all the exploited malicious datasets, each sample is labelled with respect to the belonging malware family. Basically, each family is grouping applications to the same malicious payload.

In Table 2, a brief description of the malicious behaviours and the number of the applications involved in the experimental analysis is provided.

As shown in Table 2, a total of 2552 malicious samples, belonging to 21 different malicious families, are exploited in the experimental analysis.

The last dataset we exploit is composed of 500 legitimate Android applications that we obtained from Google Play, by invoking a script exploiting a python API [32] with the aim to search and download apps. The downloaded applications belong to all the 26 different available categories, for instance Comics, Music and Audio, Games, Local Transportation, Weather, Widgets). We considered for each category the most downloaded free applications.

The goodware applications were crawled between January 2020 and March 2020. To confirm the trustworthiness of the Goole Play applications we considered the VirusTotal service, aimed to check the applications with 57 different antimalware for instance, Kasperky and Symantec and many other: this analysis confirmed that the goodware applications did not exhibit malicious payload. We take into account this dataset in order to check the false positives but also the true negatives.

The (malicious and legitimate) dataset we obtained is composed by 3052 real-world Android applications.

Table 2. Malicious families' description with the relative number of samples.

Family	Description	#
FakeInstaller	server-side polymorphic behaviour	50
Plankton	it exploits class loading to send user information	50
DroidKungFu	it executes a backdoor	50
GinMaster	it executes a malicious background service for devices rooting	50
BaseBridge	it sends sensitive information to a Command & Control server	50
Adrd	it steals user and device information	50
Kmin	it forwards user information by sending messages to premium-rate numbers	50
Geinimi	first Android botnet	50
DroidDream	able to obtain root accesses	50
Opfake	first server side polymorphic Android malware	50
Doublelocker	able to change the PIN and to cipher device files by exploiting the AES algorithm	3
Fusob	It asks to pay a ransom from $100 to $200 United States dollars	271
Jisuit	it is aimed to show user information on the QQ social network	173
Koler	porting in the Android world of the Reveton ransomware	183
Locker	it shows a ransom messages quoting a Criminal Code article	248
Lockerpin	it is aimed to change the PIN on the infected device, but also to change it if it was already enabled	181
Pletor	mobile porting of the Cryptolocker ransomware	278
Porndroid	able to reset the PIN lock of the screen	272
Simplelocker	able to cipher the SD card files with the AES algorithm with a key hardcoded in the samples.	221
Svpeng	a keylogger able to steal bank credentials	186
Xbot	it exploits phishing to steal credit card data and banking credentials	36

4.2. The μ-Calculus Formulae

In this section, we show an example of μ-calculus formula we exploited for the detection of malicious payloads in Android environment. The formulae are generated from authors by deeply inspecting the code of a couple of malicious samples for family. The idea is to codify the malicious payload in a Temporal Logic formula to easily verify the maliciousness of an application without additional work from the security analysts, providing also a method for end user for malware detection. We recall also that the proposed method is aimed to exactly detect the package, the class and the method with the related Java byte code instructions performing the harmful action.

As previously stated in the method section, we represent Android applications in terms of automaton and we verify several Temporal Logic properties (expressed in μ-calculus to verify in a first step whether the application under analysis exhibit behaviour potentially malicious and, in a second step, to effectively check the maliciousness). We formulate two properties for each considered family: the first property is aimed to detect if the *aua* can exhibit a potentially malicious behaviour belonging to this family, while the second one to confirm the maliciousness of the analysed application. In particular, with the first property, we aim to detect if in the application exists a method showing a behaviour that can be sued for malicious purposes then, with the second property, we verify if this behaviour is effectively considered for malicious purposes. We evaluate the first property on an automaton build by exploiting the CFG of the application, while the second property is evaluated on an automaton built on the application CG.

Below, we show a series of code snippets to better understand how we built both the properties and the related automata.

In detail, the following snippets are related to a malicious Android sample identified by the *6fdbd3e091ea01a692d2842057717971* hash, belonging to the Simplelocker family.

Listing 3 shows the Java code snippet obtained for the sample belonging to the Simplelocker application. Basically, it exhibits several instructions aimed to cipher a file using the AES encryption.

Listing 3: A Java code snippet of a samples belonging to the Simplelocker family.

```
public a(final String s) {
    final MessageDigest instance = MessageDigest.getInstance("SHA-256");
    instance.update(s.getBytes("UTF-8"));
    final byte[] array = new byte[32];
    System.arraycopy(instance.digest(), 0, array, 0, array.length);
    this.a = Cipher.getInstance("AES/CBC/PKCS7Padding");
    this.b = new SecretKeySpec(array, "AES");
    this.c = new IvParameterSpec(new byte[16]);
}
```

The bytecode obtained from the Java code snippet in Listing 3 is shown in Listing 4.

Listing 4: A Java bytecode snippet of a samples belonging to the Simplelocker family.

```
public a(java.lang.String arg0) { // <init> //(Ljava/lang/String;)V
    aload0 // reference to self
    invokespecial java/lang/Object.<init>()V
    ldc "SHA-256" (java.lang.String)
    invokestatic java/security/MessageDigest.getInstance(Ljava/lang/String;)Ljava/security/MessageDigest;
    astore2
    aload2
    aload1
    ldc "UTF-8" (java.lang.String)
    invokevirtual java/lang/String.getBytes(Ljava/lang/String;)[B
    invokevirtual java/security/MessageDigest.update([B)V
    bipush 32
    newarray 8
    astore1
    aload2
    invokevirtual java/security/MessageDigest.digest()[B
    iconst_0
    aload1
    iconst_0
    aload1
    arraylength
    invokestatic java/lang/System.arraycopy(Ljava/lang/Object;ILjava/lang/Object;II)V
    aload0 // reference to self
    ldc "AES/CBC/PKCS7Padding" (java.lang.String)
    invokestatic javax/crypto/Cipher.getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
    putfield org/simplelocker/a.a:javax.crypto.Cipher
    aload0 // reference to self
    new javax/crypto/spec/SecretKeySpec
    dup
    aload1
    ldc "AES" (java.lang.String)
    invokespecial javax/crypto/spec/SecretKeySpec.<init>([BLjava/lang/String;)V
    putfield org/simplelocker/a.b:javax.crypto.spec.SecretKeySpec
    aload0 // reference to self
    new javax/crypto/spec/IvParameterSpec
    dup
    bipush 16
    newarray 8
    invokespecial javax/crypto/spec/IvParameterSpec.<init>([B)V
    putfield org/simplelocker/a.c:java.security.spec.AlgorithmParameterSpec
    return
}
```

As shown from the Java bytecode snippet in Listing 4, there are several *ldc* instructions, aimed to push a one-word constant into the operand stack, containing all the arguments used by the ciphering operation. Moreover, in the bytecode we found also the invocation of classes considered for ciphering, such as MessageDigest, Cipher and SecretKeySpec. As stated in the method section, we resort to bytecode because it is always obtainable, even if the application is obfuscated with strong morphing techniques.

In Figure 3, there is a fragment of CCS automaton obtained from the Java bytecode snippet shown in Listing 4.

```
1  ...
2  proc org_simplelocker_publicvoid_a_4=pushSHA256.org_simplelocker_publicvoid_a_6
3  proc org_simplelocker_publicvoid_a_6=invokegetInstance.org_simplelocker_publicvoid_a_9
4  proc org_simplelocker_publicvoid_a_9=store.org_simplelocker_publicvoid_a_10
5  proc org_simplelocker_publicvoid_a_10=load.org_simplelocker_publicvoid_a_12
6  proc org_simplelocker_publicvoid_a_12=pushUTF8.org_simplelocker_publicvoid_a_14
7  proc org_simplelocker_publicvoid_a_14=invokegetBytes.org_simplelocker_publicvoid_a_17
8  proc org_simplelocker_publicvoid_a_17=invokeupdate.org_simplelocker_publicvoid_a_22
9  proc org_simplelocker_publicvoid_a_22=newarray.org_simplelocker_publicvoid_a_25
10 proc org_simplelocker_publicvoid_a_25=load.org_simplelocker_publicvoid_a_26
11 proc org_simplelocker_publicvoid_a_26=invokedigest.org_simplelocker_publicvoid_a_29
12 proc org_simplelocker_publicvoid_a_29=arraylength.org_simplelocker_publicvoid_a_34
13 proc org_simplelocker_publicvoid_a_34=invokearraycopy.org_simplelocker_publicvoid_a_37
14 proc org_simplelocker_publicvoid_a_37=load.org_simplelocker_publicvoid_a_38
15 proc org_simplelocker_publicvoid_a_38=pushAESCBCPKCS7Padding.org_simplelocker_publicvoid_a_40
16 proc org_simplelocker_publicvoid_a_40=invokegetInstance.org_simplelocker_publicvoid_a_43
17 proc org_simplelocker_publicvoid_a_43=newjavaxcryptospecSecretKeySpec.org_simplelocker_publicvoid_a_50
18 proc org_simplelocker_publicvoid_a_50=pushAES.org_simplelocker_publicvoid_a_54
19 proc org_simplelocker_publicvoid_a_54=invokeinit.org_simplelocker_publicvoid_a_57
20 proc org_simplelocker_publicvoid_a_57=newjavaxcryptospecIvParameterSpec.org_simplelocker_publicvoid_a_64
21 ...
22
```

Figure 3. A Control Flow Graph (CFG) automaton fragment obtained from the Java bytecode shown in Listing 4.

The fragment basically shows the sequence of the bytecode instruction, with the only exception that the bytecode ldc instructions are replaced with the push ones.

Table 3 shows the Temporal Logic property for the identification of potentially malicious behaviour belonging to the Simplelocker family.

Table 3. Temporal Logic property for Simplelocker potentially malicious behaviour detection.

$$
\begin{array}{lcl}
\varphi & = & \mu X.\, \langle pushSHA256 \rangle\, \varphi_2 \vee \langle -pushSHA256 \rangle\, X \\
\varphi_2 & = & \mu X.\, \langle invokegetInstance \rangle\, \varphi_3 \vee \langle -invokegetInstance \rangle\, X \\
\varphi_3 & = & \mu X.\, \langle pushAESCBCPKCS7Padding \rangle\, \varphi_4 \vee \langle -pushAESCBCPKCS7Padding \rangle\, X \\
\varphi_4 & = & \mu X.\, \langle newjavaxcryptospecSecretKeySpec \rangle\, \varphi_5 \vee \langle -newjavaxcryptospecSecretKeySpec \rangle\, X \\
\varphi_5 & = & \mu X.\, \langle pushAES \rangle\, \varphi_6 \vee \langle -pushAES \rangle\, X \\
\varphi_6 & = & \mu X.\, \langle invokeinit \rangle\, \mathtt{tt}\, \langle -invokeinit \rangle\, X
\end{array}
$$

The model is resulting true if the different actions (i.e., bytecode instructions) expressed in the property are present in the model, regardless of the number of other actions that are within the model. In this case, the CFG automaton is resulting true, because it exhibits the following actions: pushSHA256 (in the row 2 of the fragment in Figure 3), invokegetInstancepushAESCBCPKCS7Padding (in the row 15 of the fragment in Figure 3), newjavaxcryptospecSecretKeySpec (in the row 17 of the fragment in Figure 3), pushAES (in the row 18 of the fragment in Figure 3) and invokeinit (in the row 19 of the fragment in Figure 3).

In this case, we mark the application under analysis as potentially malicious. As a matter of fact, there are several applications that can be performed for legitimate purposes ciphering operations; for instance, for data protection. For this reason, as depicted from the proposed method main pictures, shown in Figures 1 and 2, we perform a deep analysis of the application, by building the CG and the related automaton, to understand all the application paths invoking the method (i.e., the CFG automaton) resulting true to the previous property.

Coherently with the proposed method, once at least one method is found (i.e., a CFG automaton) resulting true to a property, we found all the paths in the application under analysis invoking the potentially malicious method.

In Listing 5, we show two methods invoking the ciphering method: the first method, starting from line 23 in Listing 5, is looking for all the file paths in the device and stores the file paths in an ArrayList variable (declared as private instance variable).

Listing 5: Two methods invoking the ciphering method.

```
1  private void a(final File file) {
2      final File[] listFiles = file.listFiles();
3      for (int i = 0; i < listFiles.length; ++i) {
4          final File file2 = new File(file.getAbsolutePath(), listFiles[i].getName());
5          if (file2.isDirectory() && file2.listFiles() != null) {
6              this.a(file2);
7          } else {
8              final String absolutePath = file2.getAbsolutePath();
9              final String substring = absolutePath.substring(absolutePath.lastIndexOf(".") + 1);
10             if (this.c.contains(substring)) {
11                 this.b.add(file2.getAbsolutePath());
12             } else if (org.simplelocker.b.a.contains(substring)) {
13                 this.a.add(file2.getAbsolutePath());
14             }
15         }
16     }
17 }
18
19 private static boolean c() {
20     return "mounted".equals(Environment.getExternalStorageState());
21 }
22
23 public final void a() {
24     if (!this.d.getBoolean("FILES_WAS_ENCRYPTED", false) && c()) {
25         final a a = new a("jndlasf074hr");
26         for (final String s : this.a) {
27             a.a(s, String.valueOf(s) + ".enc");
28             new File(s).delete();
29         }
30         j.a(this.d, "FILES_WAS_ENCRYPTED");
31     }
32 }
```

We highlight that the filenames are stored only if the file extension (computed in line 32 starting of Listing 5) is belonging to a value of the *org.simplelocker.b.a* list, containing all the file extensions to select for the ciphering operations. For reader clarity, in Figure 4 we show the values of this list, i.e., the list of file extension candidates for the ciphering operation.

```
1  proc M0=M36+M39+M37+M43+M19+M10+M9
2  proc M36=noArgs.javautilArrayList_init.nil+javaioFile.M37
3  proc M39=noArgs.LjavaioFile_delete.nil+javalangString.M9+javalangString.javalangString.M10+
4      androidcontentSharedPreferences.javalangString.M43
5  proc M37=javaioFile.M37+javalangString.javalangString.javaioFile_init.nil
6  proc M43=noArgs.androidcontentSharedPreferencesDDOLLAROEditor_commit.nil
7  proc M19=noArgs.M39+androidcontentContext.M36+javalangString.javalangString.LandroidutilLog_d.nil
8  proc M10=byte.int.int.javaxcryptoCipherOutputStream_write.nil
9  proc M9=byte.javasecurityMessageDigest_update.nil
```

Figure 4. A Call Graph (CG) slice automaton obtained from the Java code shown in Listing 5.

The second method, starting from line 43 in Figure 3, is aimed to retrieve the ArrayList with the list of the file candidate for the ciphering operation and, by invoking the potentially malicious method in a cycle (i.e., one time for each file), it performs the cipher operation. We highlight that in the Simplelocker family the ciphering password is also hard coded in the application, as shown from line 48 of Listing 5. Other ransomware we considered employ more obfuscation techniques, for instance they require the password from a command and control server, that is able to generate ad-hoc passwords for each infected device (usually by considering the IMEI device). We note also, as shown from lines 51 and 52 of Listing 5 thT, once a file is ciphered, the original file is deleted and the ciphered file is stored with the *.enc* extension.

From the Java code snippet in Listing 5, we generate the Call Graph to build the CG automaton shown in Listing 6.

Listing 6: Java code snippet related to the list of file extensions considered for the ciphering operations.

```
1  package org.simplelocker;
2
3  import java.util.*;
4
5  public final class b
6  {
7      public static final List a;
8
9      static {
10         a = Arrays.asList("jpeg", "jpg", "png", "bmp", "gif", "pdf", "doc", "docx", "txt",
11             "avi", "mkv", "3gp", "mp4");
12     }
13 }
```

From the CG CCS automaton in Figure 2 emerge several considerations: first of all, there is the creation of an ArrayList (the one containing the list of files to cipher), as shown from line 2. Moreover line 5 is symptomatic of a cyclic invocation on a file (the M37 proc is returning on itself). In line 3 (i.e., proc M39) there is also the delete operation on a file (*javaioFile_delete*) and in lines 8 and 9 there are also several action symptomatics of the ciphering operation.

In this case, the property for the identification of the malicious behaviour is the one shown in Table 4.

Table 4. Temporal logic property for Simplelocker malicious behaviour detection.

ψ_1	$=$	$\mu X.\langle javautilArrayList\_init\rangle\,\mathtt{tt} \vee \langle -javautilArrayList\_init\rangle\, X$
ψ_2	$=$	$\mu X.\langle javaioFile\_delete\rangle\,\mathtt{tt} \vee \langle -javaioFile\_delete\rangle\, X$
ψ_3	$=$	$\mu X.\langle javaioFile\rangle\,\mathtt{tt} \vee \langle -javaioFile\rangle\, X$
ψ_4	$=$	$\mu X.\langle javaxcryptoCipherOutputStream\_write\rangle\,\mathtt{tt} \vee \langle -javaxcryptoCipherOutputStream\_write\rangle\, X$
ψ_5	$=$	$\mu X.\langle javasecurityMessageDigest\_update\rangle\,\mathtt{tt} \vee \langle -javasecurityMessageDigest\_update\rangle\, X$
ψ	$=$	$\psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5$

The temporal property is aimed to confirm that the sample under analysis is effectively performing a ciphering operation coherently with the Simplelocker family behaviour; in particular, the property verifies that the CG slice automaton simultaneously exhibits the javautilArrayList_init, javaioFile_delete, javaioFile, javaxcryptoCipherOutputStream_write and the javasecurityMessageDigest_update actions, symptomatic of the operation related to file reading, ciphering and deleting.

4.3. Implementation

For the generation of the CFG automata, we obtain the Java bytecode representation for each method of Android applications by invoking the *dex*2*jar* tool [33]. Starting from the *.dex* file (the executable file of the Android virtual machine), the tool extracts the *.class* files zipped as *.jar*, which makes possible to represent the Java byte-code representations of the class files stored in the *.jar*.

The Soot framework was exploited to generate the CGs. Soot is a framework developed by the Sable Research Group at McGill University [34]. It supports the analysis and transformation of Java bytecode for optimization tasks and it is an open-source software available online [35]. We implemented a static analysis to generate the CG subgraph and to output the result as a CSS process. We are extending the Soot framework with more operations for handling, generate and export graph to CCS, our tool is available on Github [36].

Soot provides four different intermediate representations for Java bytecode. Among these, we used Jimple, a typed 3-address intermediate representation that forms a simplified version of Java source code. To apply Soot graph analyses, Java bytecode is converted to Jimple, where all the call statements (*InvokeStmt* in Jimple) are marked to create the CG. Our tool implements a

SceneTransformer in the whole-jimple transformation pack (*wjtp*) to perform analyses to generate incrementally CFG, Program Dependence Graph and then the CG.

4.4. Experimental Results

Below, we present the results obtained by the proposed approach. To evaluate the effectiveness in term of malicious family detection, we compute the precision, recall, F-measure and accuracy metrics.

The *Precision* represents the proportion of the Android applications truly belonging to a certain family among all those which were labelled to belonging to this family. It is the ratio of the number of relevant applications retrieved to the total number of irrelevant and relevant applications retrieved:

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

where *tp* represents the true positives number and *fp* represents the false positives number.

The *Recall* is defined as the proportion of Android applications assigned to a certain malicious family, among all the Android applications truly belonging to the family under analysis; in other words, how many samples of the family under analysis were retrieved. It represents the ratio of the number of relevant applications retrieved to the total number of relevant applications:

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

where *fn* is the false negatives number.

The *F-Measure* represents the weighted average between the recall and the precision metrics:

$$F - Measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

The *Accuracy* represents the classifications fraction resulting as correct and it is calculated as the sum between *tp* and *tn* divided by the full set of the the Android applications considered:

$$Accuracy = \frac{tp + tn}{tp + fn + fp + tn}$$

where *tn* represents the true negative number.

In Table 5, we present the results we obtained using the proposed approach.

The proposed approach reaches an accuracy ranging between 0.97 and 1. In particular, for several families (Plankton, Opfake, Doublelocker, Locker and Simplelocker) we obtain an accuracy equal to 1, symptomatic that we are able to detect all the samples belonging to the specific family without misclassifying samples belonging to others families. An accuracy of 0.97 is obtained for the FakeInstaller family, while for the DroidKungFu, GinMaster,Adrd, Kmin, Geinimi, Koler, Pletor and Svpeng families we obtain an accuracy of 0.98. The remaining families (i.e., DroidDream, Fusob, Jisuit, Lockerpin and Porndroid) obtain an accuracy of 0.99, showing the ability of the proposed method to correctly detect most of the samples with the right belonging family.

Table 5. Performance evaluation.

Family	Precision	Recall	F-Measure	Accuracy
FakeInstaller	0.97	0.97	0.97	0.97
Plankton	1	1	1	1
DroidKungFu	0.98	0.99	0.98	0.98
GinMaster	0.98	0.98	0.98	0.98
BaseBridge	0.98	1	0.98	0.99
Adrd	0.97	0.98	0.97	0.98
Kmin	0.99	0.97	0.97	0.98
Geinimi	0.98	0.97	0.97	0.98
DroidDream	0.99	0.99	0.99	0.99
Opfake	1	1	1	1
Doublelocker	1	1	1	1
Fusob	0.98	1	0.98	0.99
Jisuit	0.98	0.99	0.98	0.99
Koler	0.98	0.98	0.98	0.98
Locker	1	1	1	1
Lockerpin	0.99	1	0.99	0.99
Pletor	0.98	1	0.98	0.98
Porndroid	0.99	1	0.99	0.99
Simplelocker	1	1	1	1
Svpeng	0.98	1	0.98	0.98
Xbot	0.99	1	0.99	0.99

5. Related Work

Previous researches demonstrated that formal verification can help to detect malware [8,37,38]. In the following, we describe malware detection methodologies employing formal methods, and briefly compare them to our approach. This section focuses mainly on Android malware detection with static analysis approaches, to compare our methodology with similar ones.

We adopted the process algebra as the Formal Model to describe the functionalities of the malware under analysis, but many more approaches were adopted in the literature. For instance, the work by Song et al. [39] models the inner-working of the applications as a sequential program (PushDown System), and then classifies the applications by checking the model over Linear Temporal Logic formulas that express malicious behaviour. Similarly to our approach, the applications were first disassembled to smali code into Control Flow Graphs. Nevertheless, including the different Formal Models adopted, the two approaches differ also on the detection task. In our methodology, we tried to find formulas able to describe the behaviours of an entire malware family, while in the approach of Song et al. each formula expresses a specific malicious behaviour, not related to any malware family or malware variants. By doing so, they have to analyse the entire application in order to detect at least one of the malicious behaviours; contrarily, our approach focuses on specific behaviour per family and splits the analysis into two granularity levels, thus, it can efficiently detect malware by analysing only subsets of the Formal Model.

Similarly to our approach, the Leila tool [28] also classifies Android apps into malware families by using a behavioural based approach. The tool is able to classify the application as belonging to a malware family and also localize the payload and the malicious code. The Leila tool adopts the process algebra to convert malicious behaviour to Temporal Logic formula, and their methodology can be easily compared to our second step of the analysis, the potentially malicious detection step. Nevertheless, our first step of the methodology narrows down the number of methods to analyse and make the entire approach liWeghter than the Leila one, preserving all the benefits. Moreover, the Leila tool was tested on 300 malware samples split among 6 different families, and 4 of them are included also in our dataset (*FakeInstaller*, *Plankton*, *GinMaster* and *Opfake*); we achieved better performance comparing the evaluation results of the four families in common between our datasets.

In addition, the paper by Battista P. et al. [38] detected Android malware and classified them in families using formal methods. Our dataset includes also the two families analysed by their approach (*DroidKungFu* and *Opfake*), and we outperform their results on both of the families.

A similar approach is applied by the Talos tool [8], which is focused on ransomware. It is able to pinpoint the payload location, achieves an accuracy of 0.99 and is also robust to obfuscation. The possibility to identify the malicious payload in Android malware using a model checking based approach were explored also by these papers [40,41]. Being able to locate the payload code is essential for automatizing and extend the scope of formal verification tools, because allow formalising more and more specific and efficient Temporal Logic formulas by manual analysing the code.

The paper by Jasiul B. et al. [42] presents an approach to modelling a malware with Colored Petri nets, and discusses its possible applications. The paper is mainly theoretical and it does not provide experimental results to compare the approaches.

Beaucamps P. et al. [43] presents the theoretical base for a framework able to produce abstracted program traces from malware, independently by the software language implementation. Their approach was applied to information leak detection task. They obtained a CFG from static analysis and collect the program set of traces. They also represent some malicious behaviour with First-Order Linear Temporal Logic, and then check if some of these behaviours were detected in the program traces. There are many similarities in our approaches, they also took into account the feasibility of the detection task in the search space, trying to address the problem of path explosion by applying techniques to restrict the set of traces. We also studied this problem and adopt the use of two graph data structures to reduce the search space and detection tasks to different granularities. The main difference resides in the practical application of the approaches proposed. While the work of Beaucamps P. et al. presents a formal framework mainly at a theoretical level, we provide a functional methodology for a well-defined problem in Android environment and present a robust experimental analysis to support our approach.

In addition, the work by Bai G. et al. [44] presents an interesting dynamic approach that takes into account, with static analysis, the problem of search space explosion. The authors have built a framework named DroidPF, based on Java PathFinder, which use software model checking to verify Android apps. The framework runs internally the application and explores the concrete state spaces, including simulated user interaction and environmental input. The state space is reduced using static analysis techniques and then the software model checking is applied to verify security properties and, eventually, pinpoint the actual violations and data leakage. Although this approach mainly uses dynamic tests on the apps, it adopts static analysis to reduce the problem of search space explosion. Therefore, the main theoretical difference in our approaches resides in the adoption of the dynamic explorations to detect the actual violations, while we use static analysis and CG representation to detect malicious behaviour. Indeed, the dynamic analysis approaches need time to collect data and build a dataset; this time-consuming task, usually, leads to test the approach on smaller datasets than similar but static approaches.

Table 6 reports a comparison between some of the related works discussed in this section. The overall performances were calculated, when applicable, using the data reports in the papers. The comparison shows that our approach improves the performance in malware detection and also takes into account malicious code localization and path explosion mitigation, which were only partially discussed by previous works.

Table 6. Comparison with related works; the ticks in the bracket for the "Malware Families" and "Localisation malicious code" columns refer, respectively, to the few numbers of families evaluated and to the localization to a higher level than method code (for instance, to class level).

Publication	**Evaluation**		**Overall Performances**					**Localisation Malicious Code**	**Path Explosion Mitigation**
	Trusted & Malware	*Malware Families*	*# Dataset*	*Prec.*	*Rec.*	*F-Meas.*	*Acc.*		
[44]	✓	✗	131	n.a.	n.a.	n.a.	n.a.	✗	✓
[43]	✓	✗	-	-	-	-	-	✗	✓
[28]	✓	✓	400	0.933	0.946	0.928	0.983	(✓)	✗
[39]	✓	✗	1331	n.a.	n.a.	n.a.	n.a.	✗	✗
[38]	✗	(✓)	400	0.928	0.805	0.858	0.878	(✓)	✗
[8]	✓	(✓)	6055	0.96	0.99	0.97	0.99	✓	✗
Our Work	✓	✓	3052	0.986	0.995	0.986	0.988	✓	✓

6. Conclusions and Future Work

Considering the need of novel methods to the real-world mobile malware detection, in this paper we design an approach exploiting model checking aimed to detect the belonging malware family of Android malware samples.

We represent an Android application in terms of Control Flow and Call Graph automata: the first automaton (i.e., the one based on Control Flow Graph) aimed to detect whether an application can potentially exhibit malicious behaviour, while the second automaton, exploiting Call Graph, performs a finer grain analysis by analysing all the paths in the application where the malicious behaviour detected by the CFG-based automata is called.

Static analysis approaches may suffer from the path explosion problem when they face an exponential increase of the search space on which apply the analysis, and mitigations should be taken into account by all static analysis approaches. Our approach implements a two step analysis with different classification granularity to reduce the search space and apply more specific detection formulae. By doing so, we are able to reduce the number of paths to analyse in the CGs, and, in case of malware, localize the malicious code on a method code level. Compared to similar works presented in the literature, our methodology is the only one who takes into account both localization of the malicious code and mitigation of path explosion problem. Moreover, we achieved also good results in the evaluation, outperforming many similar works, as reported in Section 5.

The real-world experiment reaches an accuracy from 0.97 to 1 in the evaluation of 21 Android malicious families, taking into account more than 3000 Android samples (including 2552 malicious samples). With regard to the future research directions, we will focus our attention to investigate if the proposed approach can effectively identify malware on the iOS platform. Moreover, we are working on the formulation of an algorithm aimed to automatically extract the malicious behaviour property in order to make the proposed approach fully automatic.

Author Contributions: Conceptualization, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; methodology, G.I., F.M. (Francesco Mercaldo) and A.S.; software, G.I. and F.M. (Francesco Mercaldo); validation, G.I. and F.M. (Francesco Mercaldo); formal analysis, G.I., F.M. (Francesco Mercaldo) and A.S.; investigation, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; resources, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; data curation, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; writing–original draft preparation, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; writing–review and editing, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; visualization, G.I., F.M. (Fabio Martinelli), F.M. (Francesco Mercaldo) and A.S.; supervision, F.M. (Fabio Martinelli) and A.S.; project administration, F.M. (Fabio Martinelli) and A.S.; funding acquisition, F.M. (Fabio Martinelli). All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially supported by MIUR—SecureOpenNets, EU SPARTA, CyberSANE and E-CORRIDOR projects.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bhat, H.J. Investigate the Implication of "Self-service Business Intelligence (SSBI)"—A Big Data Trend in Today's Business World. *Curr. Trends Inf. Technol.* **2020**, *10*, 17–22.
2. Taylor-Quiring, D.P.; Wiebe, N.J. Cognitive and Predictive Analytics on Big Open Data. In Proceedings of the Cognitive Computing—ICCC 2020: 4th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, 18–20 September 2020; Springer Nature: London, UK, 2020; Volume 12408, p. 88.
3. Gautam, A.; Chatterjee, I. Big Data and Cloud Computing: A Critical Review. *Int. J. Oper. Res. Inf. Syst. (IJORIS)* **2020**, *11*, 19–38. [CrossRef]
4. Chatfield, A.T.; Reddick, C.G. Cybersecurity innovation in government: A case study of US pentagon's vulnerability reward program. In Proceedings of the 18th Annual International Conference on Digital Government Research, Staten Island, NY, USA, 7–9 June 2017; pp. 64–73.
5. Langner, R. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Secur. Priv.* **2011**, *9*, 49–51. [CrossRef]
6. Liagkou, V.; Kavvadas, V.; Chronopoulos, S.K.; Tafiadis, D.; Christofilakis, V.; Peppas, K.P. Attack detection for healthcare monitoring systems using mechanical learning in virtual private networks over optical transport layer architecture. *Computation* **2019**, *7*, 24. [CrossRef]
7. Cimino, M.G.; De Francesco, N.; Mercaldo, F.; Santone, A.; Vaglini, G. Model checking for malicious family detection and phylogenetic analysis in mobile environment. *Comput. Secur.* **2020**, *90*, 101691. [CrossRef]
8. Cimitile, A.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Talos: no more ransomware victims with formal methods. *Int. J. Inf. Secur.* **2018**, *17*, 719–738. [CrossRef]
9. Statista.com. Development of New Android Malware Worldwide from June 2016 to March 2020. Available online: https://www.statista.com/statistics/680705/global-android-malware-volume/ (accessed on 20 September 2020).
10. Fan, M.; Liu, T.; Liu, J.; Luo, X.; Yu, L.; Guan, X. Android malware detection: A survey. *Sci. Sin. Inf.* **2020**, *50*, 1148–1177.
11. Casolare, R.; Martinelli, F.; Mercaldo, F.; Santone, A. Android Collusion: Detecting Malicious Applications Inter-Communication through SharedPreferences. *Information* **2020**, *11*, 304. [CrossRef]
12. Bayazit, E.C.; Sahingoz, O.K.; Dogan, B. Malware Detection in Android Systems with Traditional Machine Learning Models: A Survey. In Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), Ankara, Turkey, 26–28 June 2020; pp. 1–8.
13. Pedreschi, D.; Giannotti, F.; Guidotti, R.; Monreale, A.; Ruggieri, S.; Turini, F. Meaningful explanations of Black Box AI decision systems. In Proceedings of the AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; Volume 33, pp. 9780–9784.
14. Milner, R. *Communication and Concurrency*; PHI Series in Computer Science; Prentice Hall: Upper Saddle River, NJ, USA, 1989.
15. Camilleri, J.; Winskel, G. CCS with priority choice. *Inf. Comput.* **1995**, *116*, 26–37. [CrossRef]
16. Milner, R.; Parrow, J.; Walker, D. A calculus of mobile processes, I. *Inf. Comput.* **1992**, *100*, 1–40. [CrossRef]
17. Stirling, C. An Introduction to Modal and Temporal Logics for CCS. In *Concurrency: Theory, Language, And Architecture*; Springer: Berlin/Heidelberg, 1989; pp. 2–20.
18. Emerson, E.A. Temporal and modal logic. In *Formal Models and Semantics*; Elsevier: Amsterdam, The Netherlands, 1990; pp. 995–1072.
19. Gabbay, D.M.; Hodkinson, I.; Reynolds, M.A.; Finger, M. *Temporal Logic: Mathematical Foundations and Computational Aspects*; Clarendon Press Oxford: Oxford, UK, 1994; Volume 1.
20. Ben-Ari, M.; Pnueli, A.; Manna, Z. The temporal logic of branching time. *Acta Inform.* **1983**, *20*, 207–226. [CrossRef]
21. Francesco, N.D.; Lettieri, G.; Santone, A.; Vaglini, G. Heuristic search for equivalence checking. *Softw. Syst. Model.* **2016**, *15*, 513–530. [CrossRef]

22. Orailoglu, A.; Gajski, D.D. Flow graph representation. In Proceedings of the 23rd ACM/IEEE Design Automation Conference, Las Vegas, NV, USA, 29 June–2 July 1986; pp. 503–509.
23. Allen, F.E. Control flow analysis. *ACM Sigplan Not.* **1970**, *5*, 1–19. [CrossRef]
24. Cesare, S.; Xiang, Y. Classification of malware using structured control flow. In Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing, Darlinghurst, Australia, 3–7 February 2010; Volume 107, pp. 61–70.
25. Bruschi, D.; Martignoni, L.; Monga, M. Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 129–143.
26. LaToza, T.D.; Myers, B.A. Visualizing call graphs. In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Pittsburgh, PA, USA, 18–22 September 2011; pp. 117–124.
27. Kinable, J.; Kostakis, O. Malware classification based on call graph clustering. *J. Comput. Virol.* **2011**, *7*, 233–245. [CrossRef]
28. Canfora, G.; Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Leila: formal tool for identifying mobile malicious behaviour. *IEEE Trans. Softw. Eng.* **2018**, *45*, 1230–1252. [CrossRef]
29. VirusTotal. VirusTotal Developer Hub. Available online: https://developers.virustotal.com/reference (accessed on 28 September 2020).
30. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. *NDSS* **2014**, *14*, 23–26.
31. Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Vaglini, G. Model Checking and Machine Learning techniques for HummingBad Mobile Malware detection and mitigation. *Simul. Model. Pract. Theory* **2020**; Volume 105, 102169. [CrossRef]
32. Google Play Unofficial Python API. Available online: https://github.com/egirault/googleplay-api (accessed on 28 September 2020).
33. dex2jar—Tools to Work with Android.dex and java.class Files. Available online: https://github.com/pxb1988/dex2jar (accessed on 20 September 2020).
34. Vallée-Rai, R.; Gagnon, E.; Hendren, L.; Lam, P.; Pominville, P.; Sundaresan, V. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 18–34.
35. Soot Java Optimization Framework. Available online: https://github.com/soot-oss/soot (accessed on 20 September 2020).
36. Iadarola, G. graph4apk—Analysis for Generating Graphs from Apk File. Available online: https://github.com/Djack1010/graph4apk (accessed on 20 September 2020).
37. Iadarola, G.; Martinelli, F.; Mercaldo, F.; Santone, A. Formal Methods for Android Banking Malware Analysis and Detection. In Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 22–25 October 2019; pp. 331–336.
38. Battista, P.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Identification of Android Malware Families with Model Checking. In Proceedings of the ICISSP, Rome, Italy, 19–21 February 2016; pp. 542–547.
39. Song, F.; Touili, T. Model-checking for android malware detection. In *Asian Symposium on Programming Languages and Systems*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 216–235.
40. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Ransomware Steals Your Phone. Formal Methods Rescue It. In Proceedings of the Formal Techniques for Distributed Objects, Components, and Systems—36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, 6–9 June 2016; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9688, pp. 212–221.
41. 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, FormaliSE@ICSE 2016, Austin, TX, USA, 15 May 2016; ACM: New York, NY, USA, 2016; pp. 22–28.
42. Jasiul, B.; Szpyrka, M.; Śliwa, J. Formal specification of malware models in the form of colored Petri nets. In *Computer Science and its Applications*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 475–482.

43. Beaucamps, P.; Gnaedig, I.; Marion, J.Y. Abstraction-based malware analysis using rewriting and model checking. In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 806–823.
44. Bai, G.; Ye, Q.; Wu, Y.; van der Merwe, H.; Sun, J.; Liu, Y.; Dong, J.S.; Visser, W. Towards Model Checking Android Applications. *IEEE Trans. Softw. Eng.* **2017**, *44*, 595–612. [CrossRef]