Model Checking and Machine Learning techniques for *HummingBad* Mobile Malware detection and mitigation

Fabio Martinelli[a], Francesco Mercaldo[a,c], Vittoria Nardone[b], Antonella Santone[c], Gigliola Vaglini[d]

[a] *Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy*
[b] *Department of Engineering, University of Sannio, Benevento, Italy*
[c] *Department of Medicine and Health Sciences "Vincenzo Tiberio", University of Molise, Campobasso, Italy*
[d] *Department of Information Engineering, University of Pisa, Pisa, Italy*

Abstract

Android currently represents the most widespread operating system focused on mobile devices. It is not surprising that the majority of malware is created to perpetrate attacks targeting mobile devices equipped with this operating systems. In the mobile malware landscape, there exists a plethora of malware families exhibiting different malicious behaviors. One of the recent threat in this landscape is represented by the *HummingBad* malware, able to perpetrate multiple attacks for obtain root credentials and to silently install applications on the infected device. From these considerations, in this paper we discuss two different methodologies aimed to detect malicious samples targeting Android environment. In detail the first approach is based on machine learning technique, while the second one is a model checking based approach. Moreover, the model checking approach is able to localize the malicious behaviour of the application under analysis code, in terms of package, class and method. We evaluate the effectiveness of both the designed methods on real-world samples belonging to the *HummingBad* malware family, one of the most recent and aggressive behaviour embed into malicious Android applications.

Keywords: Model Checking, Formal Methods, Machine Learning, Malware, Android, Security

---

1. Introduction and Background

Malware targeting mobile devices (i.e., smartphones and tables) are really widespread. As a matter of fact, our devices are really of interest for malicious writers, considering the plethora of sensible and private information that are stored in these devices [1].

As a matter of fact, McAfee security analysts highlight a dramatic increase in not only the number of new malware, but the sophistication and complexity of Android malware[1]: during the second half of 2016, the increase in smartphone infections was 83% following on the heels of a 96% increase during the first half of the same year[2].

In this landscape, a new malware family called *HummingBad* has infected a plethora of devices [2].

HummingBad family was discovered by Check Point analysts in February 2016[3]. This malware is characterised for the ability to silently install a rootkit on Android devices[3, 4]. Moreover, its malicious payload is able to obtain advertisement revenue by silently installing external fraudulent applications [5]. Security analysts estimated to be generating $300,000 per month in fraudulent advertisement revenue. Moreover they state that considering the great number of *HummingBad* infected devices, it is possible to generate a botnet and carry out targeted attacks on businesses or government agencies.

In a nutshell, the malicious aim of this family is to gather root privileges to execute drive-by-download attacks [6].

HummingBad samples basically exploit two different attack vectors: the first one aimed to exploit root access, the second one is initialised whether the first attack fails and its malicious goal is the same of the previous one. This double attack is repeated until it is able to obtain root privileges. Once the root access is finally obtained, the *HummingBad* payload is able to communicate with the attacker (C&C) server (i.e., Command and Control) with the intent to obtain a list of malware applications. Once obtained this list, the *HummingBad* malicious payload will start to silently install several malicious applications obtained from the downloaded list in the infected device.

In current literature, researchers developed several methods for detecting Android malware exploiting static [7, 8] or dynamic analysis [9]. We focus on the

[1] https://www.mcafee.com/us/resources/reports/rp-m

[2] https://pages.nokia.com/8859.Threat.Intelligence.Report.html

[3] https://blog.checkpoint.com/2016/02/04/HummingBad-a-persistent-mobile-chain-attack/

first strategy (i.e., the static one), that is the one involved in the proposed method.

The analysis of the bytecode targeting the Android Dalvik Virtual Machine [8]) is considered in [7]. This paper focuses on the op-code analysis: the occurrences of op-code n-grams are used, by means of supervised machine learning, to classify apps as benign or malicious.

Differently, researchers in [10] analyse sets of required permissions for the classification of malicious application. In [11], behaviors symptomatic of malware as, for instance, sending *SMS* messages without confirmation or accessing unique phone identifiers like the *IMEI* are identified for malware detection. The main issues of these methods are related to the adoption of the permissions as feature [12]: this is reflecting in the high false positive rates obtained from these methods [13, 14].

The cited methods basically relies in the generation of models by exploiting machine learning supervised classification. Recently, the possibility to identify the malicious payload in Android malware using a model checking based approach has been explored in [15, 16, 17, 18]. Starting from the payload behavior definition, the authors formulate logic rules and then test them by using a real-world dataset. The main difference between these works and the one we propose is represented by the focus on the *HummingBad* malicious payload.

In this paper we discuss two different approaches for malware detection in mobile environment based on static analysis: the first approach exploits machine learning techniques [19], while the second one considers the model checking technique [2] for the detection and the localization the malicious payload.

This paper represents an extension of a preliminary work entitled: "Model Checking to Detect the HummingBad Malware" [2] appeared in the "International Symposium on Intelligent and Distributed Computing" (IDC 2019). The differences with respect to the work in [2] are the following:

- we evaluate an extended dataset of real-world applications (i.e., 1000), while in reference [2] the proposed method based on model checking was preliminary evaluated (on 250 applications);
- we discuss and we experiment a second method, based on several machine learning classifiers, with the aim to compare the performances obtained by the model checking based approach;
- we evaluate mobile applications obfuscated with three different morphing engines (while in reference [2] only one morphing engine was considered);

- we provide an example of malicious payload localization and we propose a way to sanitise malicious applications.

The paper continues with Section 2, introducing the machine learning based approach, in Section 3 the methodology based on formal methods is described, experimental analysis is presented in Section 4. Finally, conclusions and future works are drawn in Section 5.

2. The Machine Learning Approach

The idea behind the machine learning based approach we discuss is to classify malware by considering a set of features counting the occurrences of a specific group of op-codes extracted from the smali code of the application under analysis (AUA in the remaining of the paper). Smali is a language that represents disassembled code for the Dalvik Virtual Machine [4], a virtual machine optimized for the hardware of mobile devices.

The designed machine learning based method consists in producing histograms from of a set of op-codes belonging to the AUA: each histogram dimension represents the number in which the op-code corresponding to that dimension appears in the code.

We resort to op-codes as feature considering that they represent static features (i.e., that do not require the AUA execution) largely exploited in the current state-of-art-literature regarding malware analysis. [20, 21, 22] . As a matter of fact, the rationale behind the choice of these op-codes is guided from the assumption that legitimate mobile application exhibit a greater complexity if compared to malicious malware, as demonstrated in [23, 24] .

Following op-codes are take into account in this study:

- *move*: aimed to move the content of one a first register in a second register;
- *jump*: aimed to deviate the control flow to a new instruction;
- $packed-switch$: representing a switch statement by using an index table;
- $sparse-switch$: representing a switch statement with sparse case table;
- *invoke*: considered for method invocation;

[4] `http://pallergabor.uw.hu/androidblog/dalvik\_opcodes.html`

- *if*: basically a Jump instruction considered for the verification of a truth predicate.

For the feature computing following steps are considered. The first one is aimed to preprocess the AUA for histograms generation [19, 13].

The output of this step is represented by a set of histograms. In detail one histogram for each class is obtained; each histogram is composed by six dimensions, where a dimension is related to one of the six op-codes we previously described. The second step is aimed to compute two additional features, represented by two different Minkowski distances.

To obtain op-code representation of the AUA we exploit APKTool[5], a software for Android application reverse engineering able to generate Dalvik source code files.

Figure 1 shows the process we consider for histogram generation.

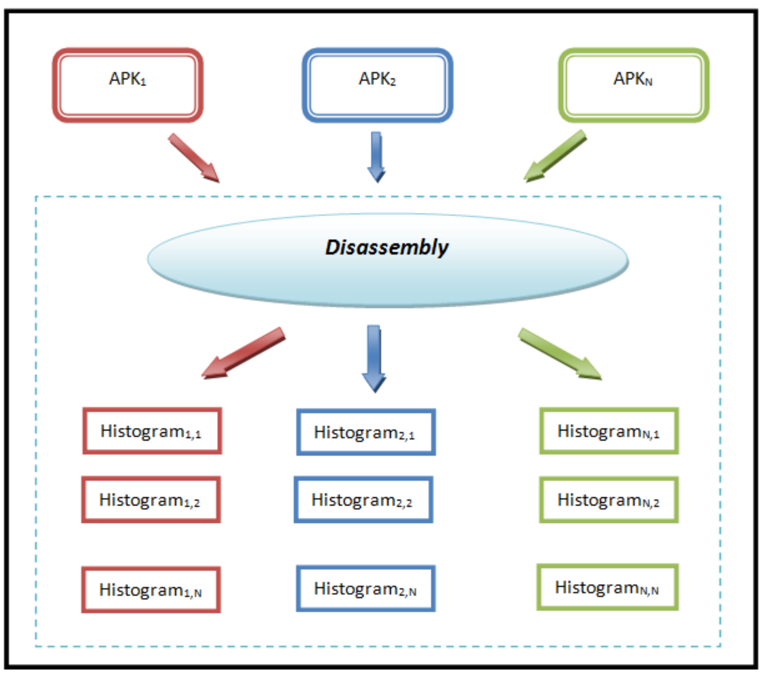

Figure 1: Histograms generation.

In Fig. 2 we show an example related to a class histogram.

[5] `https://code.google.com/p/android-apktool/`

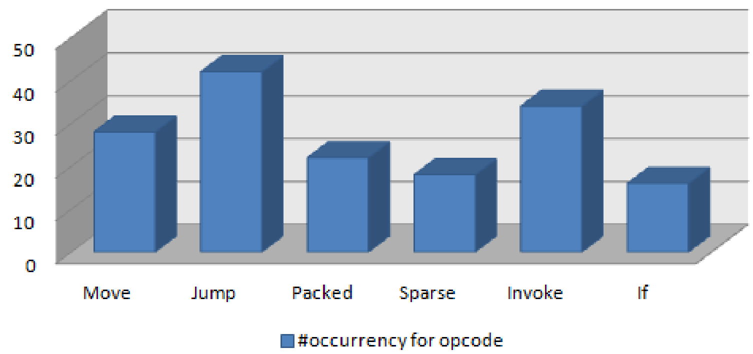

Figure 2: Histogram generated from the n-th class of the j-th AUA.

The first six features are computed as follows; let X be one of the following values:

- M_i: 'move' occurrences in the i-th class;
- J_i: 'jump' occurrences in the i-th class;
- P_i: 'packed-switch' occurrences in the i-th class;
- S_i: 'sparse-switch' occurrences in the i-th class;
- K_i: 'invoke' occurrences in the i-th class;
- I_i: 'if' occurrences in the i-th class.

Then:

$$\#X = \frac{\sum_{k=1}^{N} X_i}{\sum_{k=1}^{N}(M_i+J_i+P_i+S_i+K_i+I_i)}$$

where X is the occurrence of one of the six op-codes extracted and N is the total number of the classes forming the AUA.

The next step is related to the computation of the Minkowski distances between the various histograms obtained with the step 1. In the follow we explain these two features but for clarity it is useful to briefly recall the Minkowski distance.

Let's consider two vectors of size n, $X = (x_i, x_2, \ldots, x_n)$ and $Y = (y_i, y_2, \ldots, y_n)$, then the Minkowski distance between two vectors X and Y is:

$d^r_{X,Y} = \sum_{k=1}^{N} |x_i - y_i|^r$

One of the most popular histogram distance measurements is the Euclidean distance. It is a Minkowski distance with r = 2:

$d^E_{X,Y} = \sqrt{\sum_{k=1}^{N} (x_i - y_i)^2}$

Another popular distance is represented by the Manhattan distance. It is a form of the Minkowski distance, but in this case r = 1:

$d^M_{X,Y} = \sum_{k=1}^{N} |x_i - y_i|$

The last two features are the Manhattan and Euclidean distance, computed with a process of three steps. Given an AUA containing N classes, the AUA will have N histograms, one for each class, where each histograms H_i will be a vector of six values, each one corresponding to an op-code of the model ('move', 'jump', 'packed-switch', 'sparse-switch', 'invoke', 'if').

As an example, we will show an application of the model to a simplified case in which the model has only three classes and two op-codes. Let's assume that the AUA's histograms are H_1= {4,2}, H_2={2,1}, H_3={5,9}.

- *Step1*: the Minkowski distance is computed among each pair H_i, H_j with i≠j and 1≤i,j≤N. In the example we will have $d_{1,2}$=3; $d_{1,3}$=2; $d_{2,3}$=11.We do not compute $d_{2,1}$, $d_{3,1}$ and $d_{3,2}$ because Minkowski distance is symmetric, i.e. $d_{i,j}$ = $d_{j,i}$ for 1≤i,j≤N. For simplicity we consider only the Manhattan distance in the example;

- *Step 2*: the vector with all the distances is computed for each AUA, D= {$d_{i,j}$ — i≠j and 1≤i≤ N, 2≤j≤ N}. Each dimension of the vector corresponds to a class of the AUA. In the example D ={3, 2, 11}.

- *Step 3*: the max element in the vector is extracted, which is M_{AUA} = MAX (D[i]). In the example M_{AUA} is 11.

Finally the last two features are the values M_{AUA} computed, respectively, with Manhattan and Euclidean distance. Thus, M_{AUA} is a measure of dissimilarity among the classes of the AUA.

These features represents the input for building several models by exploiting following supervised classification algorithms: J48, Random Forest, Hoeffding

Tree and Neural Network, really widespread for classification problems [4, 25]. In detail we set the supervised classification algorithms with following parameters:

- with regard to the J48 algorithm we consider the minimum number of instance for leaf equal to 2 and 100 for the preferred number of instances to process for batch prediction;
- with regard to the Random Forest algorithm we set the number of iteration to perform equal to 100 and 100 for the preferred number of instances to process for batch prediction (similarly to the J48 algorithm);
- with regard to the Hoeffding Tree algorithm we also exploit the batch prediction instances equal to 100 and 200 as number of instances a leaf should observe between split attempts;
- with regard to the Neural Network algorithm we set the number of epoch equal to 10 and one hidden layer formed by 100 units.

3. The Formal Methods Approach

In this section the second method i.e., the model checking based approach for Android malware families detection is described. In according with the model checking technique [26, 27, 28], we need: a formal model of the system, a set of behavioural properties and a model checker tool able to verify the property on the model. Since the model and the properties require a precise notation to be defined, we use the Calculus of Communicating Systems of Milner (CCS) [29] and the mu-calculus logic [30], respectively to define them. The CAAL (Concurrency Workbench, Aalborg Edition) [31] is exploited in this work as formal verification environment. CAAL supports several different specification languages, among which CCS. In the CAAL environment the verification of temporal logic formulae is based on model checking.

Below we describe the step for the modeling an AUA in terms of labelled transition system. To achieve this goal we use the code of the AUA to build the formal model. We retrieve the application code, i.e., Java Bytecode, through a reverse engineering process and we perform the following steps:

- we use dex2jar[6] tool to convert the the Dalvik Executable file (`dex`) into Java Archive file (`jar`);
- we extract the Java classes using the command: `jar -xvf` provided by the Java Development Kit;
- we parse the classes file using the Bytecode Engineering Library (Apache Commons BCEL)[7].

Finally, every Java Bytecode instruction is translated in a CCS process through a Java Bytecode-to-CCS transformation function defined by the authors. We translate every Java Bytecode instruction in a CCS process through an our Java Bytecode-to-CCS transformation function. Since in the CCS process algebra the systems are represented through processes and actions, which correspond to states and transitions, respectively, our model of the system is represented as an automaton. This representation allows to simulate the normal flow of the instructions. The automaton of an application has a set of labelled edges and a set of nodes. The nodes are the system states while an edge represents a transition from a state to another state (precisely the next state). An edge means that the system can evolve from a state s to a state s' performing an instruction a (the label of the edge). For example, the `if` statement is translated as a non-deterministic choice: the system can evolve from a state s to two different states s' and s'', corresponding to the two alternative paths (true/false) of the classical `if` statement.

In detail a CCS model for each method of the AUA is generated. This is obtained by translating each java byte-code instruction in a CCS process. The definition of the translation can be found in [32, 33, 34]. In the follow we recall the main concepts to better understand the proposed method for generating automata from Android applications.

With regard to the sequential Java byte-code instructions, the translation is the following:

$$proc\ x_{current} = op - code.x_{next}$$

where, $x_{current}$ is the current instruction under analysis, while x_{next} represent the process related to the next instruction and finally, $op - code$ represents the name of the Java byte-code instruction. An example of CCS translation from translation of sequential op-code instructions is shown in Listings 1 and 2.

[6] `https://sourceforge.net/projects/dex2jar/`
[7] `https://commons.apache.org/proper/commons-bcel/`

Listing 1: CCS process for Listing 1

```
proc M1 = aload.M2
proc M2 = getfield.M3
proc M3 = return.nil
```

Listing 2: CCS process for Listing 2

```
proc M1 = goto.M2
proc M2 = goto.M3
proc M3 = goto.M4
proc M4 = aload.M5
proc M5 = getfield.M6
proc M6 = goto.M7
proc M7 = return.nil
```

Branch instructions are used to change the sequence of the instruction execution. We consider the $+$ operator to manage the choice [29].

A CSS process is built for each method of the application under analysis. Let be *aua* an application under analysis. Supposing that the *aua* has n methods, i.e., $F_1, \ldots, F_n$, the *aua* CCS representation has n $M_1, \ldots, M_n$ CCS processes.

In order to identify the malicious behaviour, we specify temporal logic formulae written in mu-calculus logic. The specified formulae encode a specific malicious behaviour, which is a typical behaviour characterizing the family. These are temporal logic rules and are obtained through a manual inspection process of few malware samples and examining malware technical reports. Finally, we use CAAL tool which takes as input the formal CCS model (built as described above) and the temporal logic rules written in mu-calculus logic. The output of the model checker is binary: `true`, whether the property is verified on the model and `false` otherwise. We assume that a sample belongs to a particular family whether the properties related to that particular family are verified on the model.

Figure 3 outlines the above described work-flow of our approach underlying the second approach based on formal methods.

4. Experimental Analysis

In this section we present the results we obtained from the evaluation of the machine learning and model checking approaches in the detection of the *HummingBad* malware samples.

4.1. Machine Learning Approach Evaluation

The evaluation of the machine learning approach consists of building several classifiers and evaluating the reached accuracy for each classifier.

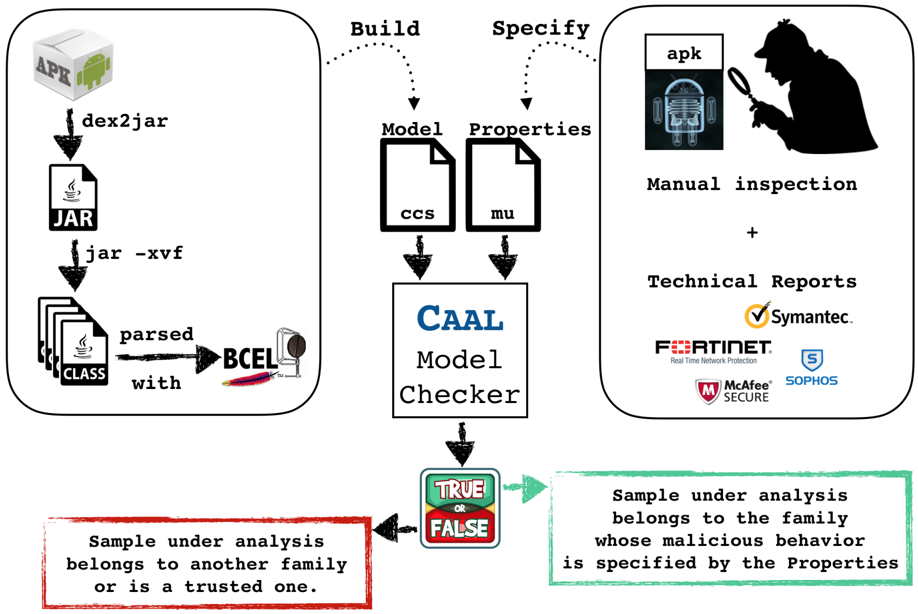

Figure 3: The work-flow of the model checking approach

For model training , we defined T as a set of labelled mobile applications *(AUA, l)*, where each AUA is associated to a label $l \in \{$*not HummingBad, HummingBad*$\}$. For each *AUA* we built a feature vector F $\in R_y$, where y represents the feature number ($1 \leq y \leq 8$).

For the learning a k-fold cross-validation is considered with the aim to better generalise the proposed model.

Following procedure is adopted to evaluate the proposed supervised machine learning model:

1. build a training set $T \subset D$;
2. build a testing set T' = D÷T;
3. run the training phase on T;
4. apply the learned classifier to each element of *T'*.

A 5-fold cross validation is considered i.e, the procedure is repeated for five times varying the composition of T (and, as consequence, of *T'*).

```java
var3 = new Intent("com.android.vending.INSTALL_REFERRER");
if (var2.contains("referrer")) {
   String var6 = Uri.parse(var2).getQueryParameter("referrer");
   Intent var4 = new Intent("com.android.vending.INSTALL_REFERRER");
   var2 = var6;
   if (!TextUtils.isEmpty(var6)) {
      var2 = var6;
      if (!var6.contains("android_id=")) {
         var2 = "&android_id=" + AppInfoUtils.getAndroidId(this.val$context);
         var2 = var6 + var2;
      }

      var4.putExtra("referrer", var2);
      Log.e("HDJ", "å¹¿å'Šé"¾æŽ¥ sendReferrerã€□æ˜¯ã€'ï¼š" + var2);
   }
```

Figure 4: Code snippet for the Humming malware identified by the *0a4c8b5d54d860b3f97b476fd8668207a78d6179b0680d04fac87c59f5559e6c* hash.

4.2. Formal Methods Approach Evaluation

In the follow we describe the temporal logic formula for the *HummingBad* malicious payload detection.

In Figure 4 we show a real-world Java code snippet of a typical malicious behaviour exhibited by the *HummingBad* malware.

We highlight in the code snippet the behaviour shown by the *com.android.vending.INSTALL_REFERRER* intent: it is sent in broadcast when an app is installed from the official Android market [8]. In this way the *HummingBad* malware is able to listen for that Intent, passing the install referrer data for Mobile Apps and Google Analytics.

Humming malware is able to send referrer requests to generate a Google Play advertisement revenue. For this reason, the *HummingBad* malware obtains a list of packages and referrer ids from the C&C server and subsequently scans the applications running on the infected device. Once the *HummingBad* malicious payload collected these information, it sends the *com.android.vending.INSTALL_REFERRER* intents with the corresponding referrer ID, with the to obtain revenue.

The temporal logic property able to catch this behaviour is the following: the AUA is labelled as malware belonging to the *HummingBad* family if in the AUA there is at least one invocation of the *com.android.vending.INSTALL_REFERRER*

[8] `https://developers.google.com/android/reference/com/google/android/gms/tagmanager/InstallReferrerReceiver`

$$\varphi = \mu X. \langle pushcomandroidvendingINSTALLREFERRER \rangle \, \texttt{tt} \vee \langle pushcomandroidvendingINSTALLREFERRER \rangle \, X$$

Table 1: Temporal logic formula for the *HummingBad* malicious behaviour detection.

intent, as shown in Table 1.

4.3. The Overall Evaluation

In the evaluation of both the designed approaches we consider the following dataset: 550 samples belonging to the *HummingBad* family[9], 300 samples randomly selected from the 10 most populous families of the Drebin dataset [35] and 150 legitimate samples downloaded from Google Play[10], the Android official market. The full dataset is composed by 1000 Android samples.

It worth to note that our dataset is composed of only real word samples. Drebin dataset is a well known collection of malware used in many scientific works, which includes the most diffused Android families. We consider in the 10 most populous families, shown in Table 2. The family label is related to the malicious payload that a particular family exposes. Thus, every sample is labelled and categorized starting from its malicious behaviour.

Table 2 shows in descending order the top 10 Drebin families, from the most populous to the minus one. In our evaluation we randomly selected 25 samples from each one of them. We want to demonstrate if our tool is able to correctly categorize and distinguish the samples belonging to the *HummingBad* family from the other ones (i.e., legitimate applications and malware belonging to other families).

In order to evaluate the completeness and correctness of our methodology we have computed the following metrics: Precision (PR), Recall (RC), F-measure and Accuracy.

$$Precision = \frac{TP}{TP+FP}; \; Recall = \frac{TP}{TP+FN};$$

[9] http://contagiominidump.blogspot.com/2016/07/hummingbad-android-fraudulent-ad.html

[10] https://play.google.com

Table 2: Top 10 most populous families belonging to Drebin dataset

Family	Total number of samples	Number of samples randomly selected for our evaluation
FakeInstaller	925	30
DroidKungFu	667	30
Plankton	625	30
Opfake	613	30
GinMaster	339	30
BaseBridge	330	30
Kmin	147	30
Geinimi	92	30
Adrd	91	30
DroidDream	81	30
TOTAL	-	**300**

$$F-measure = \frac{2PR\,RC}{PR+RC};\;\; Accuracy = \frac{TP+TN}{TP+FN+FP+TN}$$

In the above formulae are involved also the values of True Positives (TP), False Positives (FP), False Negatives (FN) and True Negatives (TN). In our evaluation these values assume the following meaning: a sample results as a TP if our tool correctly identifies it in the *HummingBad* family; a sample results as a TN if our tool correctly identifies it as not belonging to the *HummingBad* family; when our tool classifies a samples in the wrong family, it is considered as an FP; when our tool not classifies a sample in the *HummingBad* family, it is considered as an FN.

Table 3 shows the results achieved from the two methodologies.

Table 3: Performance Evaluation

Method	Precision	Recall	F-measure	Accuracy
J48	**0.918**	**0.915**	**0.919**	**0.921**
Random Forest	**0.945**	**0.943**	**0.945**	**0.943**
Hoeffding Tree	**0.926**	**0.928**	**0.932**	**0.937**
Neural Network	**0.972**	**0.978**	**9.974**	**0.981**
Formal Methods	**1**	**1**	**1**	**1**

Both the designed approaches obtain interesting results. In fact with regard to the machine learning classifiers (i.e., J48, Random Forest, Hoeffding Tree and Neural Network), they obtain an accuracy ranging from 0.921 to 0.981, symptomatic that the models are able to discriminate between generic malware, legitimate samples and *HummingBad* mobile applications. In detail the model obtaining the best performances is the one built by exploiting the Neural Network algorithm with a precision of 0.972 and a recall equal to 0.978. Also formal methods obtain interesting performances, by overcoming the machine learning approach: in fact this second approach is able to correctly recognize the *HummingBad* samples without any negative result.

With the aim to demonstrate that the proposed approaches are able to overcome the performances of the current anti-malware technologies, we report the results obtained by analysing the *HummingBad* malicious samples with several diffused anti-malware software by submitting the *HummingBad* samples: the results of this analysis are shown in Table 4.

Table 4: Comparison between our methodology and anti-malware (in terms of samples detected)

AVG	Ad Aware	Avast	Arcabit	Alibaba	ESET NOD32	McAfee
0	0	0	0	11	0	0

Only the *Alibaba* anti-malware is able to detect 11 (on 500) *HummingBad* samples as belonging to the *HummingBad* family.

Furthermore, to show the effectiveness of the approach obtaining the best performances i.e., the approach based on model checking, we consider a set of well-known code transformations techniques [36, 37, 38] applied to the *HummingBad* applications. These techniques are used by malware writers to evade the signature-based detection approaches adopted by current anti-malware [39].

In particular, we applied following transformation techniques:

1. **Disassembling & Reassembling.** The compiled Dalvik Bytecode in *classes.dex* of the application package may be disassembled and reassembled through *apktool*. This allows various items in a *.dex* file to be represented in another manner. In this way, signatures relying on the order of different items in the *.dex* file are likely to be ineffective with this transformation.

2. **Repacking.** Every Android application has a developer signature key that will be lost after disassembling and reassembling the application. Using the *signapk*[11] tool, it is possible to embed a new default signature key in the

[11] `https://code.google.com/p/signapk/`

reassembled application in order to avoid detection signatures that match the developer keys.

3. **Changing package name.** Each application is identified by a unique package name. This transformation renames the application package name in both the Android Manifest file and all the application classes.

4. **Identifier renaming.** This transformation renames each package name and class name by using a random string generator, in both the Android Manifest file and *smali* classes, handling renamed classes invocations.

5. **Data Encoding.** Strings could be used to create detection signatures to identify malware. To elude such signatures, this transformation encodes strings with a *Caesar cipher*. The original string is restored during application execution with a call to a *smali* method that knows the *Caesar key*.

6. **Call indirections.** This transformation mutates the original call graph of the application by modifying every method invocation in the code with a call to a new method which simply invokes the original method.

7. **Code Reordering.** This transformation is aimed at modifying the instructions order in the application methods. A random reordering of instructions has been accomplished by inserting *goto* instructions with the aim of preserving the original run-time execution trace.

8. **Defunct Methods.** This transformation adds new methods that perform defunct functions, clearly the logic of the original source code remains unchanged.

9. **Junk Code Insertion.** These transformations introduce code sequences that have no effect on the function of the code. Detection algorithms relying on instructions sequences may be defeated by this transformation. This transformations provides insertion of *nop* instructions into each method, unconditional jumps into each method, and allocation of three additional registers performing garbage operations.

10. **Encrypting Payloads and Native Exploits.** In Android, native code is usually made available as libraries accessed via Java Native Interface (JNI). However, some malware, such as DroidDream, also pack native code exploits meant to run from a command line in non-standard locations in the

application package. All such files may be stored encrypted in the application package and be decrypted at run-time. Certain malware such as DroidDream also carry payload applications that are installed once the system has been compromised. These payloads may also be stored encrypted. These are easily implemented and have been observed in the wild (e.g., DroidKungFu malicious family uses encrypted exploit [6]).

11. **Function Outlining and Inlining.** In function outlining, a function is broken down into several smaller functions. Function inlining involves replacing a function call with the entire function body. These are typical compiler optimization techniques. However, outlining and inlining can also be used for call graph obfuscation.

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

We apply the full transformation set to the *HummingBad* samples with the Droidchameleon [37], the ADAM [38] and the Carnival[12] tools. Table 5 shows the obfuscation techniques implemented by the three tools.

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

Table 6 reports the achieved results with the obfuscated samples showing that the performances keep pretty unchanged.

A previous work [37] demonstrated that current anti-malware solutions fail to recognize the malware after these transformations. We applied our method to the morphed dataset in order to verify if the proposed model checking based method is able to detect *HummingBad* malicious payload even the malware has been obfuscated.

The analysis confirms that the proposed method is resilient to the common code obfuscation techniques.

[12] `https://github.com/faber03/AndroidMalwareEvaluatingTools`

Table 5: The transformation techniques provided by considered obfuscators.

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

Table 6: Resilience to the Obfuscation Techniques

dataset	**Original**		**Morphed**	
	# Samples	**TP**	**# Samples**	**TP**
HummingBad	500	**500**	500	**500**

Considering the ability of the model checking based approach to detect but also to localise the package, the class and the method of malicious payload, it is possible to sanitise the malicious application. Considering the snippet in Figure 4 (which relative CCS model is labeled as TRUE when the temporal logic property in Table 1 is evaluated), to perform a sanitisation process it is necessary to remove the method labelled and their invocation and rebuild the AUA. Once rebuilt the AUA, to verify whether the *HummingBad* malicious behaviour is effectively removed, the formula in Figure 7 can be evaluated.

We formulate the property aimed to detect this behaviour whether there is no invocation of the *com.android.vending.INSTALL_REFERRER* intent, as shown in Table 1 by the ψ formula.

Whether the formula shown in Table 7 is resulting TRUE the AUA is not affected by the *HummingBad* malicious payload and the sanitisation process was effectively performed.

$$\psi = \nu X.[pushcomandroidvendingINSTALLREFERRER]\, \texttt{ff} \wedge [pushcomandroidvendingINSTALLREFERRER]X$$

Table 7: Temporal logic formula for the *HummingBad* sanitisation verification.

5. Conclusion and Future Work

In last years mobile malware has widely spread, thankful to the great diffusion of mobile devices currently employed in a plethora of contexts of our everyday life for instance, from banking account management to social network activities. For this reason in our devices are stored an increasing number of private and sensitive information and they are so appealing from the malicious writers point of view.

We proposed in this paper the design and the implementation of two different approaches for the malicious behaviour detection related to Android environment: the first approach is based on supervised machine learning while the second one considers the model checking technique. Both the approaches are evaluated by analyzing the real-world *HummingBad* malicious family, one of most aggressive threat recently discovered in the Android malware landscape. Both the machine learning and the model checking based approaches obtained interesting performances, but our outcomes demonstrate that model checking obtains better performances from the malicious payload detection point of view. As a matter of fact, an accuracy equal to 1 is obtained by the model checking based method by evaluating 1000 (malicious and legitimate) real-world Android applications. Moreover we evaluate also the model checking technique resilience to widespread obfuscation techniques currently employed by malicious writers: the experiment confirms that the model checking method is able to detect *HummingBad* malware even it is obfuscated.

The proposed method can be easily applied for the detection and the sanitisation of other widespread families. In fact, once the malware analysts formulated the logic temporal property (starting, from instance, from the manual inspection of a couple of malicious samples), the proposed model is immediately applicable for the detection of all kind of malicious payloads (once the analysts formulated the logic temporal property). As shown from the experiment focused on the *HummingBad* malware, once found the property for the detection of the malicious behaviour, the sanitisation property is immediately found.

For these reasons, as future works, we plan to extend the experiments to other widespread malware threats with the aim to enforce the methodology proposed in this work. Moreover authors plan to evaluate the proposed method for the detection and the sanitisation of malicious payloads in iOS samples.

Acknowledgments

This work has been partially supported by MIUR - SecureOpenNets and EU SPARTA contract 830892, CyberSANE projects, and the EU project CyberSure 734815.

[1] A. De Lorenzo, F. Martinelli, E. Medvet, F. Mercaldo, A. Santone, Visualizing the outcome of dynamic analysis of android malware with vizmal, Journal of Information Security and Applications 50 (2020) 102423.

[2] F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, G. Vaglini, Model checking to detect the hummingbad malware, in: International Symposium on Intelligent and Distributed Computing, Springer, 2019, pp. 485–494.

[3] F. Mercaldo, V. Nardone, A. Santone, C. Visaggio, Hey malware, i can find you!, 2016, pp. 261–262. doi:10.1109/WETICE.2016.67, cited By 10.

[4] A. Cimitile, F. Martinelli, F. Mercaldo, Machine learning meets ios malware: Identifying malicious applications on apple environment., in: ICISSP, 2017, pp. 487–492.

[5] M. G. Cimino, N. De Francesco, F. Mercaldo, A. Santone, G. Vaglini, Model checking for malicious family detection and phylogenetic analysis in mobile environment, Computers & Security 90 (2020) 101691.

[6] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012), 2012.

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

[8] H.-S. Oh, B.-J. Kim, H.-K. Choi, S.-M. Moon, Evaluation of android dalvik virtual machine, in: Proceedings of the 10th International Workshop on

Java Technologies for Real-time and Embedded Systems, ACM, 2012, pp. 115–124.

[9] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, A. K. Sangaiah, Android malware detection based on system call sequences and lstm, Multimedia Tools and Applications 78 (2019) 3979–3999.

[10] W. Enck, M. Ongtang, P. McDaniel, On lightweight mobile phone application certification, in: Proceedings of the 16th ACM conference on Computer and communications security, ACM, 2009, pp. 235–245.

[11] A. P. Felt, M. Finifter, E. Chin, S. Hanna, D. Wagner, A survey of mobile malware in the wild, in: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2011, pp. 3–14.

[12] A. P. Felt, K. Greenwood, D. Wagner, The effectiveness of application permissions, in: Proceedings of the 2nd USENIX conference on Web application development, 2011, pp. 7–7.

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

[14] G. Canfora, E. Medvet, F. Mercaldo, C. A. Visaggio, Detecting android malware using sequences of system calls, in: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, ACM, 2015, pp. 13–20.

[15] F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Ransomware steals your phone. formal methods rescue it, in: International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Springer, 2016, pp. 212–221.

[16] F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Download malware? no, thanks: how formal methods can block update attacks, in: Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, ACM, 2016, pp. 22–28.

[17] F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Hey malware, i can find you!, in: Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2016 IEEE 25th International Conference on, IEEE, 2016, pp. 261–262.

[18] F. Mercaldo, V. Nardone, A. Santone, Ransomware inside out, in: Availability, Reliability and Security (ARES), 2016 11th International Conference on, IEEE, 2016, pp. 628–637.

[19] G. Canfora, F. Mercaldo, C. A. Visaggio, Evaluating op-code frequency histograms in malware and third-party mobile applications, in: International Conference on E-Business and Telecommunications, Springer, 2015, pp. 201–222.

[20] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, M. Rajarajan, Android security: a survey of issues, malware penetration, and defenses, IEEE communications surveys & tutorials 17 (2014) 998–1022.

[21] M. Egele, T. Scholte, E. Kirda, C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, ACM computing surveys (CSUR) 44 (2008) 1–42.

[22] E. Gandotra, D. Bansal, S. Sofat, Malware analysis and classification: A survey, Journal of Information Security 2014 (2014).

[23] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, C. A. Visaggio, Effectiveness of opcode ngrams for detection of multi family android malware, in: Availability, Reliability and Security (ARES), 2015 10th International Conference on, IEEE, 2015, pp. 333–340.

[24] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, R. E. Bryant, Semantics-aware malware detection, in: 2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA, IEEE Computer Society, 2005, pp. 32–46.

[25] G. Canfora, F. Mercaldo, C. A. Visaggio, An hmm and structural entropy based detector for android malware: An empirical study, Computers & Security 61 (2016) 1–18.

[26] E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT Press, 2001.

[27] A. Santone, G. Vaglini, Abstract reduction in directed model checking ccs processes, Acta Informatica 49 (2012) 313–341.

[28] M. Ceccarelli, L. Cerulo, A. Santone, De novo reconstruction of gene regulatory networks from time series data, an approach based on formal methods, Methods 69 (2014) 298–305. doi:10.1016/j.ymeth.2014.06.005.

[29] R. Milner, Communication and concurrency, PHI Series in computer science, Prentice Hall, 1989.

[30] C. Stirling, An introduction to modal and temporal logics for ccs, in: A. Yonezawa, T. Ito (Eds.), Concurrency: Theory, Language, And Architecture, volume 491 of *LNCS*, Springer, 1989, pp. 2–20.

[31] J. R. Andersen, N. Andersen, S. Enevoldsen, M. M. Hansen, K. G. Larsen, S. R. Olesen, J. Srba, J. K. Wortmann, CAAL: concurrency workbench, aalborg edition, in: Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings, volume 9399 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 573–582.

[32] G. Canfora, F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Leila: formal tool for identifying mobile malicious behaviour, IEEE Transactions on Software Engineering 45 (2018) 1230–1252.

[33] G. Crincoli, T. Marinaro, F. Martinelli, F. Mercaldo, A. Santone, Code reordering obfuscation technique detection by means of weak bisimulation, in: International Conference on Advanced Information Networking and Applications, Springer, 2020, pp. 1368–1382.

[34] A. Cimitile, F. Mercaldo, V. Nardone, A. Santone, C. A. Visaggio, Talos: no more ransomware victims with formal methods, International Journal of Information Security 17 (2018) 719–738.

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

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

[37] V. Rastogi, Y. Chen, X. Jiang, Droidchameleon: evaluating android anti-malware against transformation attacks, in: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM, 2013, pp. 329–334.

[38] M. Zheng, P. P. Lee, J. C. Lui, Adam: an automatic and extensible platform to stress test android anti-virus systems, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2012, pp. 82–101.

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