"Mirror, Mirror on the Wall, Who is the Fairest One of All?" *Machine Learning versus Model Checking: A Comparison between Two Static Techniques for Malware Family Identification*

Vittoria Nardone and Corrado Aaron Visaggio
Department of Engineering, University of Sannio, Benevento, Italy

Keywords: Malware, Android, Security, Model Checking, Testing.

Abstract: Malware targeting Android platforms is growing in number and complexity. Huge volumes of new variants emerge every month and this creates the need of being able to recognize timely the specific variants when encountered. Several approaches have been developed for malware detection. Recently the research community is developing approaches able to detect malware variants. Among all, two approaches demonstrated high performances in detecting malware and assigning the family it belongs to: one based on machine learning and one on formal methods. In this paper we compare the results achieved by two methods in terms of Precision, Recall and Accuracy. We highlight points of strength and weakness of two methods.

1 INTRODUCTION

Mobile devices are spreading at an impressive pace, and as reported by the Worldwide Quarterly Mobile Phone Tracker, in the second quarter of 2016 Android kept the greatest market share of mobile OS[1].

This record has boosted the community of malware writers to devote efforts towards mobile platforms. According to *Internet Security Threat Report*[2], the number of Android malware families added in 2015 grew by 6 percent,compared with the 20 percent growth in 2014. The volume of Android malware variants increased by 40 percent in 2015, compared with 29 percent growth in the previous year, while there were more than three times as many Android apps classified as containing malware in 2015 than in 2014, an increase of 230 percent.

Smartphones are also largely used for building botnet, as a recent DDOS attack has demonstrated where 1 TB per second of traffic has been conveyed by just using infected smartphones remotely controlled[3]. These facts suggest that it is urgent to find techniques of detection that are able to detect malware targeting mobile platforms, contrasting the evasion techniques whose current malware makes large use of. Different and diverse methods have been proposed to detect mobile malware and classify variants. Machine learning base classification is one of the most investigated. The main limit stands in the fact that the effectiveness of the classifier depends on the kind of malware (and the number) that is included in the training set. A malware that is not represented by the training set will not be detected. This limit is not trivial, if we consider the huge number of variants and new kinds of malware that are released in the wild each month. Formal methods have the capability to assess with a very high precision whether a rule is verified by a piece of code. The immediate advantage of this technique is that if the specific behaviors represented by the rules are shown by the malware, they will be surely recognized. The second advantage is that if a categorized behavior is implemented in the program, the formal methods are able to locate it in the code. In this paper we compare the two approaches, the one based on machine learning (Canfora et al., 2016) and the one based on formal method (Mercaldo et al., 2016a; Mercaldo et al., 2016c), in order to characterize the points of strength and weakness of the techniques. The paper proceeds as follows: section 2 discusses the related literature, section 3 provides the background for the compared approaches, while section 4 describes the approaches in detail. Section 5 presents the experimentation and the obtained results. Finally, section 6 draws the conclu-

[1]http://www.idc.com/prodserv/smartphone-os-market-share.jsp
[2]https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf
[3]http://thehackernews.com/2016/09/ddos-attack-iot.html

Nardone, V. and Visaggio, C.
"Mirror, Mirror on the Wall, Who is the Fairest One of All?" - Machine Learning versus Model Checking: A Comparison between Two Static Techniques for Malware Family Identification.
DOI: 10.5220/0006287506630672
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 663-672
ISBN: 978-989-758-209-7

sion of the work.

2 RELATED WORK

Authors in (Alam et al., 2016) use clone detection for recognizing malware variants for Android. Authors applied the method to a smaller and older data-set than ours (166 Andorid malware). The main limitation of this technique is that if a variant is not a clone of the representative family members it is not recognized. (Faruki et al., 2015) explores the homogeneity of bytes distribution for capturing similarity among programs' variants. This technique can be sensitive to obfuscation. (Zhang et al., 2014) proposes an approach that classifies Android malware via dependency graph. The authors build programs semantics with contextual API.They are able to detect correctly the 93% malware instances.

The authors in (Suarez-Tangil et al., 2014) present `Dendroid`, a text mining based approach. Suarez-Tangil et al. base their approach on the code structures and they use a real data-set of Android malware families[4]. They measure similarity between malware samples, and than they use this similarity to automatically classify the malware into families. Their approach uses a data-set of 1260 malware collected in 2010 and it has a smaller number of samples for each family if it is compared with the data-set used by (Mercaldo et al., 2016a; Mercaldo et al., 2016c; Canfora et al., 2016). The researchers in (Feng et al.,) present a semantics-based approach (called Apposcopy) to identify Android malware. Apposcopy specifies semantic characteristics of malware families using signatures. The signature matching algorithm of Apposcopy uses a combination of static taint analysis and a new form of program representation called Inter-Component Call Graph to efficiently detect Android applications that have certain control- and data-flow properties. Apposcopy in evaluated on the Malgenoma data-set, as Dendroid (Suarez-Tangil et al., 2014). The authors in (Battista et al., 2016; Mercaldo et al., 2016b), using a model checking based approach, identify the malicious payload in repackaged Android applications. The logic rules define the malicious payload. As preliminary evaluation the authors only investigate DroidKungFu, Opfake and FakeInstaller families. Another behavioural based approach is described in (Bose et al., 2008). Bose and his colleagues specify common malware behavior using temporal logic formulas. This approach is partially dynamic and it uses mobile viruses and worms targeting the Symbian OS. The frequencies of ngrams of opcodes to identify Android malware family is used in (Canfora et al., 2015). The authors use a data-set composed of 5560 malware belonging to several different families. The results show on the average an accuracy equals to 97%.

[4]Android Malware Genome Project available at http://www.malgenomeproject.org

3 PRELIMINARIES

In this section we introduce some preliminary concepts related to the two techniques compared in this work. In particular the first one is a classification realized with a Machine Learning engine (Canfora et al., 2016), while the second one is a formal technique that uses the model checking for verifying the presence of certain malicious behaviors in the code (Mercaldo et al., 2016a; Mercaldo et al., 2016c). Both the methods are static.

3.1 Machine Learning

The classification based on machine learning consists of identifying some features to be extracted from a source code that allow the distinction between malware and goodware. This process is made of two main steps:

1. *Training*: in this phase the classifier is built, by applying algorithms of data mining. The engine evaluates which are the features that better distinguish the two classes of objects, in this case malware and goodware. The learning could be supervised or not supervised. The learning is supervised if the training data-set is labeled with the name of the belonging class. The methodology under analysis is based on the supervised learning, because it's known in advance whether the application is malicious or not.
2. *Prediction*: this is the phase of evaluation. The aim of this step is to evaluate the effectiveness of the classifier constructed in the previous phase. At this stage the capability of the classifier to predict the class a data-set's member belongs to is assessed. When detecting malware this stage evaluates whether the classifier can discriminate correctly a malware from a goodware.

It should be underlined that a good classification is performed only with a selection of an appropriate set of features. One of the main limit of a machine learning classifier is that the performance depends on how much the training set is representative of the two class examples.

3.2 Model Checking

Model Checking is a type of formal technique. Formal Methods are usually used to specify and verify complex systems. Model checking technique requires three steps: (i) to define the systems with a precise notation; (ii) to specify the properties with a precise notation; (iii) to verify the properties on the system with a model checker tool.

Define the System

The system behavior is represented as an automaton. There are a set of labeled 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). The edges are labeled. An edge means that the system can evolve from a state s to a state s' performing an action a (the label of the edge). This transition is indicated as follows: $s \xrightarrow{a} s'$. The initial state of the system is the root of the automaton. It is often convenient to algebraically represent the automaton in the form of processes. Usually the process algebras have been used as precise notation to describe complex computer systems. The methodology under analysis uses Milner's Calculus of Communicating Systems (CCS) (Milner, 1989) as process algebra. In CCS process algebra the systems are represented through processes and actions, which respectively correspond to states and transitions. For more details on CCS the reader can refer to (Bruns, 1997; Milner, 1989).

Specify the Properties

A property that a system should satisfy can be defined using a temporal logic. In temporal logics there are constructs allowing to verify in a formal way that a particular event will eventually happen or that a property is verified in every state. The methodology under analysis uses the logic named *mu-calculus* (Stirling, 1989).

Model Checker Tool

Finally, to verify the properties defined in the temporal logic, the Model Checker is applied to the system (modeled as transition system). This is a tool that takes two inputs: the system model and the property. The output of the Model Checker is binary. It returns `true` whether the property is verified on the model or `false` otherwise. The check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. The methodology under analysis uses as formal verification environment the *Concurrency Workbench of New Century (CWB-NC)* (Cleaveland and Sims, 1996). While model checking was originally developed to verify the correctness of systems, recently it has been also proposed in other fields such as clone detection (Santone, 2011), biology (De Ruvo et al., 2015), secure information flow (De Francesco et al., 2003), and mobile computing (Anastasi et al., 2001). In the last years, model checking has been successfully applied also in the security field, as explained in the following sections.

4 THE TWO APPROACHES APPLIED

The paper compares the performance of two approaches, one based on Machine Learning, and the other one based on Model Checking which are now detailed. The Machine Learning based approach is presented in (Canfora et al., 2016). The Model Checking based approach is presented in (Mercaldo et al., 2016a; Mercaldo et al., 2016c).

4.1 Machine Learning based Methodology (*ML*)

The methodology based on machine learning (Canfora et al., 2016) uses two different techniques to detect Android malware families: the Hidden Markov Model (HMM) (Rabiner, 1989; Annachhatre et al., 2015) and the Structural Entropy (Baysa et al., 2013). HMM models certain sequences of opcodes belonging to the app under analysis that could characterize the eventual malicious behavior. The structural entropy evaluates the distribution of bytes in the physical file for characterizing it in terms of malware (or goodware). These two techniques are used to extract four features (f_1, f_2, f_3, f_4). The first three capture the HMM while the last one evaluates the Structural Entropy. In detail the extracted features are: (i)f_1 is a HMM score with 3 hidden states; (ii)f_2 is a HMM score with 4 hidden states; (iii)f_3 is a HMM score with 5 hidden states; (iv)f_4 is a measure of the structural entropy.

The classification process aim is to establish whether the features correctly classify the malware family. In this approach six classification algorithms are used: J48, LabTree, NBTree, RandomForest, RandomTree and RepTree.

The HMM-based features are composed starting from the sequence of instructions of the application. In particular the authors consider the sequence of opcodes in the smali[5] code of the application. Starting from the entry point of each application (i.e., the Main Activity), the authors reconstruct the sequence

[5]http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

Table 1: An example of logic rule.

$$\begin{aligned}
\varphi_1 &= \mu X = \langle pushCOMMANDS \rangle\, \varphi_{1_1} \vee \langle -pushCOMMANDS \rangle\, X \\
\varphi_{1_1} &= \mu X = \langle pushCommands \rangle\, \varphi_{1_2} \vee \langle -pushCommands \rangle\, X \\
\varphi_{1_2} &= \mu X = \langle pushTgZzeroLHwIICkoa \rangle\, \varphi_{1_3} \vee \langle -pushTgZzeroLHwIICkoa \rangle\, X \\
\varphi_{1_3} &= \mu X = \langle pushACTIVATION \rangle\, \varphi_{1_4} \vee \langle -pushACTIVATION \rangle\, X \\
\varphi_{1_4} &= \mu X = \langle pushActivation \rangle\, \varphi_{1_5} \vee \langle -pushActivation \rangle\, X \\
\varphi_{1_5} &= \mu X = \langle pushTgOottoHBgYfBVoM \rangle\, \mathtt{tt} \vee \langle -pushTgOottoHBgYfBVoM \rangle\, X
\end{aligned}$$

of opcodes of every called method, jumping to the instructions of the called method whenever there is an `invoke` instruction. This reconstruction ends whether there is a class belonging to the Android framework or when the recursion level is equal to 4. On these sequences the HMM detector is trained. The authors used a number N of hidden states equal to 3, 4 and 5, according with the features f_1, f_2, f_3.

Regarding the Structural Entropy method, the authors estimate the structural entropy of the Android executable (*.dex* file). Starting from blocks of different size, belonging to the *.dex* file, the method computes the Shannon entropy for each block. The wavelet transform is used to represent the segments of the file. Finally a similarity score, based on Levenshtein distance, is computed. The authors compare the segments of two files to compute this score. At the end of this process the feature f_4 is computed.

4.2 Model Checking based Methodology (*MC*)

The methodology based on model checking (Mercaldo et al., 2016a; Mercaldo et al., 2016c) is composed of two main steps:

- to build the model through a translation of Bytecode instructions in form of process;
- to specify the properties related to malicious behaviours.

The formal model is written in CCS (Calculus of Communicating Systems of Milner (Milner, 1989)). The authors use a transformation function that translates every Bytecode instruction of the Android application into CCS process. In particular, starting from the apk file of an application through a reverse engineering process the authors obtain the `.class` files. Afterwards the authors use the Apache Commons Bytecode Engineering Library (BCEL)[6] to parse the Bytecode in order to translate every instruction in a CCS process. This is an automatic process. At the end of the first step the formal model is built.

According to the model checking technique to formal verification, the authors specify some properties. The aim of step two is to investigate whether an application is a malware and belongs to a particular Android family. In order to achieve this goal, the specified formulae catch a specific malicious behavior, which is a typical behavior allowing the family characterization. The mu-calculus logic, (Stirling, 1989) as a branching temporal logic, is used to describe a determined behavior of the app. Thus, after this second step, for every malware family there is a set of formulae able to catch a specific malicious behaviour. These are temporal logic rules and are obtained through a manual inspection process of malware samples. Also the specification of the property is not automatic.

Table 1 shows the logic rule related to a malicious behaviour exposed by Plankton sample. The formula catches some commands of the Plankton botnet. In this formula is used the last fixpoint ($\mu Z.\phi$) of the recursive recursive equation $Z = \phi$. μZ *binds* free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ.

In this approach the Concurrency Workbench of New Century (CWB-NC) (Cleaveland and Sims, 1996) is used as formal verification environment. The CWB-NC model checker takes as input the formal CCS model (built in the first step) and the temporal logic rules written in mu-calculus (specified in the second step). The output of the model checker is binary: `true`, whether the property is verified on the model and `false` otherwise. The authors assume that a sample belongs to a particular family whether the properties related to that particular family are verified on the model.

It is well-known that a model checking technique typically suffers of the state explosion problem. In fact, it is mainly applicable to small-scale applications, but do not scale up well. However, the state explosion problem is not a real problem when verifying Android applications, since the produced CCS specifications do not have a large number of states and transitions.

[6]http://commons.apache.org/bcel/

Table 2: Number of samples used by two methodologies.

Family	(Canfora et al., 2016)	(Mercaldo et al., 2016a)	(Mercaldo et al., 2016c)
FakeInstaller	925	40	60
DroidKungFu	667	40	60
Plankton	**625**	**625**	60
Opfake	613	40	60
GinMaster	339	40	60
BaseBridge	**330**	**330**	60
Kmin	147	40	60
Geimini	92	0	60
Adrd	91	0	60
DroidDream	81	0	60
AnserverBot	0	187	0
DroidKungFu Update	0	1	0
Ransomware	**672**	0	**683**

5 THE COMPARISON

The experimentation aims at comparing the performances of the two different methodologies. The performances of classification are measured with the metrics recall, precision and accuracy that evaluate the ability of the methods to correctly detect the family a malware belongs to. The experimentation is carried out on a real world data-set of Android applications.

5.1 Metrics

The performances of the methodology are evaluated with the following metrics:

$$PR = \frac{TP}{TP+FP};\ RC = \frac{TP}{TP+FN}; \quad Acc = \frac{TP+TN}{TP+FN+FP+TN}; \tag{1}$$

which are respectively the Precision (*PR*), the Recall (*RC*) and the Accuracy (*Acc*) formulae. The first two formulae indicate the measures of exactness and correctness since the Precision tests the *quality* and the recall tests the *quantity* of the detection. The Accuracy evaluates the percentage of correct classifications with respect of the total number of examined samples. The variables in the Equations 1 are the following: TP (True Positives) indicates the number of malware programs that are correctly associated to the right family, FP (False Positives) indicates the number of malware programs that are erroneously associated to a family, FN (False Negatives) indicates the number of malware programs that are not associated to the belonging family, and TN (True Negatives) indicates the number of malware programs that do not belong to the considered families, and the classification does not associate them with any family.

It should be underlined that the Precision value strictly depends on the number of samples that is incorrectly identified. A sample is not correctly identified when the prediction of its family is wrong. The Precision depends on the number of False Positives: increasing the number of samples belonging to different families could increase also the number of FP.

Even if the Accuracy's formula includes the number of FP, it evaluates the number of correct classifications on the overall data-set. This makes the accuracy a measure more comparable between the two data-sets with different size.

5.2 Data-set

The two methodologies compared in this work use in their experimentation the following two data-sets: Drebin (Arp et al., 2014; Spreitzenbarth et al., 2013) and a collection of freely available Ransomware samples (672[7] and 11[8]). In particular the machine learning based methodology in its experimentation (Canfora et al., 2016) uses the ten most numerous families of Drebin data-set and the collection of ransomware samples. The model checking based methodology in (Mercaldo et al., 2016a) uses for its experimentation the Android malware samples that implement the update attack and malware from other Drebin families. Plankton, AnserverBot, BaseBridge and DroidKungFu-Update are the families that implement the update attack. In (Mercaldo et al., 2016c) the authors use the ransomware samples and samples belonging to the ten most numerous families of Drebin data-set. Table 2 shows the number of samples used by the two methodologies in their experimentation. As shown in Table 2, in the cases of Plankton, BaseBridge and Ransomware families the two approaches

[7]http://ransom.mobi/

[8]http://contagiominidump.blogspot.it/

use the same data-set. This is the reason why we compare the results achieved recognizing these three families. We compare their results in terms of correctness and in this case the comparison is perfect. In terms of Precision and Accuracy the comparison is a bit different since in these metrics are involved the number of FPs. As mentioned above, False Positive is a sample not correctly identified and when the number of samples belonging to different families increases, also the number of FP could increase. This difference of the data-set is considered in our comparison.

The results used in this comparison are the results achieved by two methodologies recognizing Plankton, BaseBridge anf Ransomware samples. The families considered present the following malicious behaviours:

Plankton family: the samples belonging to this family send sensitive data of the infected smartphone to a remote server, like IMEI and browser history. They use the class loading (a native functionality) to perform the malicious actions. Furthermore, Plankton downloads unwanted advertisements and changes the browser homepage or adds unwanted bookmarks to it.

BaseBridge family: the samples of this family run an embedded payload located in an external folder. They are able to receive premium numbers from remote C&C servers and dial calls or send out SMS messages to them, incurring fees for users.

Ransomware family: The main malicious aim of the samples belonging to the Ransomware family is to steal all personal data stored in the phone by encrypting all the files residing in the smartphone. Alternatively the malware could lock the phone: in both the cases the user is not able to access the smartphone, so the ransomware asks for money in order to unlock the phone.

5.3 Results

Table 3 shows the values of Recall obtained with the two methodologies, where:

- **Family** column indicates the malware family the classification results refer to.
- **Machine Learning based Approach** column contains the values of Recall reached by the methodology based on Machine Learning technique. In particular it is composed of four sub-columns:
  - **Algorithm** sub-column shows the algorithms used for the classification.
  - f_1 sub-column shows the Recall achieved by the feature one with the six classification algorithms. The feature f_1 captures the HMM (Hidden Markov Models) with 3 hidden states.
  - f_2 sub-column shows the Recall results achieved by the feature f_2 that captures the HMM with 4 hidden states.
  - f_3 sub-column shows the Recall results obtained by the feature f_3 that captures the HMM with 5 hidden states.
  - f_4 sub-column shows the Recall results pursued with the feature f_4 that measures the Structural Entropy of the bytes distribution.
- **Model Checking based Approach** column contains the values of Recall reached by the methodology that applies the Model Checking technique.

In our comparison, the results show that the methodology based on Model Checking outperforms all the other techniques. This means that the Model Checking based approach reaches the best values of correctness in the classification. Regarding the methodology based on Machine Learning, the best performances are produced by the feature f_4. As a matter of fact this feature shows also the smallest variability of Recall among the six used classification algorithms. This means that the feature f_4 is able to correctly identify the right family the malware belongs to. The other three features are very sensitive to the used algorithm, as a matter of fact there are values of Recall that widely vary.

The histogram in Figure 1 shows the results of Precision achieved by the two methodologies, grouped for malware family. We reported in the graph only the maximum values of Precision obtained by Machine Learning based approach for all the considered features(f_1, f_2, f_3 and f_4), as this value represents the best performance that can be reached with a specific pair (feature, classification algorithm).

The results highlight that the Precision value reached by the Model Checking based approach is grater than the values achieved by the other approach. Unfortunately, here the comparison between the two method's precision is just an indication of the real differences in exactness, as previously discussed.

With regards to Accuracy in Figure 1, the Model Checking outperforms the Machine Learning for all the data-sets, with the only exception of BaseBridge family where performances are equals.

5.4 Discussion

Hereafter we will refer to the two approaches by using the acronyms *ML* and *MC*,standing respectively for Machine Learning approach and Model Checking approach. The experimentation allowed us to characterize the pros and cons of the two approaches.

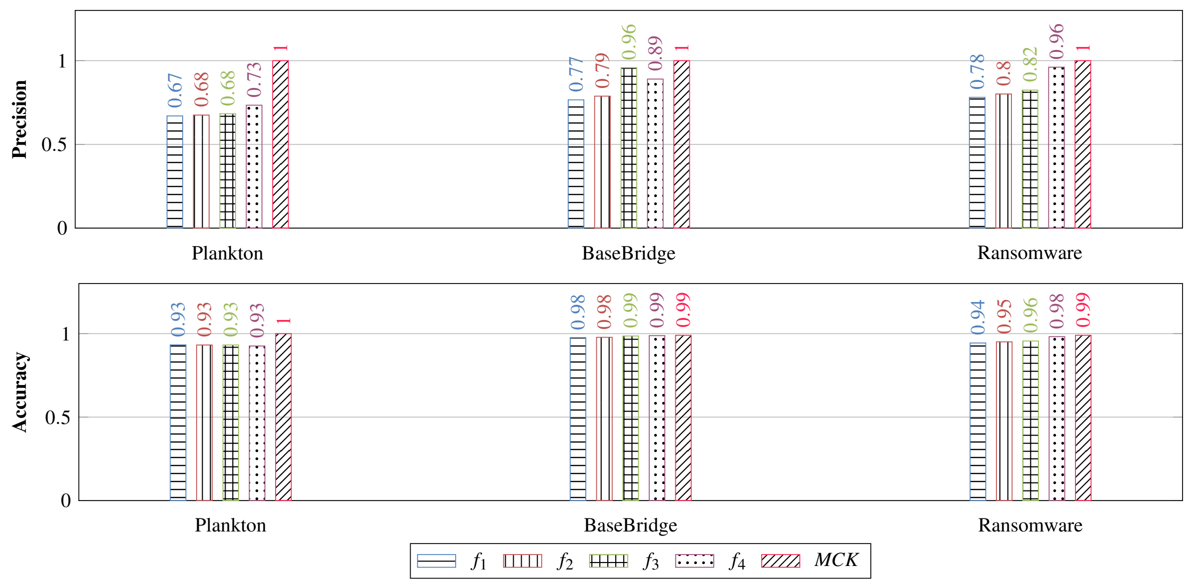

Figure 1: Precision and Accuracy values Comparison. The values of f_1, f_2, f_3, and f_4 (Precision Histogram) are the maximum Precision values achieved by the Machine Learning based approach. The maximum is selected between the six different classification algorithms. MCK (Model Checking) indicates the value of Precision and Accuracy reached by the methodology based on Model Checking.

Strengths and Advantages

The *ML* is a completely automatic approach. It reaches a good Recall in the malware family identification, especially using the feature f_4 (as shown in Table 3). The authors in (Canfora et al., 2016) perform a very large experimentation in order to validate their approach. The *ML* has a very low execution time, especially when the samples are analyzed using the feature f_4 (Structural Entropy). In fact, the average CPU time required is equal to 3.85 sec on a personal computer with the following computational profile: Intel Core i5 desktop with 4 gigabyte RAM, equipped with Linux Mint 15. The other three features f_1, f_2, and f_3 are effective for discriminating malware from goodware, but they do not perform well in recognizing the family a malware belongs to. In fact, the Precision and the Recall in malware identification are always greater than the 93% (as shown in (Canfora et al., 2016)). In the light of all above, *ML* obtains good result in malware family classification when using the feature f_4 with a low execution time, while *ML* achieves a very good correctness and exactness in malware detection using the other three features.

Conversely, the *MC* reaches high levels of correctness (i.e. Recall) in malware families identification (as shown in Table 3). Futhermore, since it is based on a formal method, it is a very rigorous approach and it is able to identify the exact location of the malicious payload in the malware code. This is made possible by the specified formulae that describe the malicious behaviour to be found within the malware. In particular, *MC* points out the method where the payload is located. *MC* does not require a training set, but for each family a set of samples must be manually inspected to extract the formulae representing the malicious behavior. In the worst case, the largest set counted 20 samples, which is small if compared with the average size of the training sets used in *ML*. Another advantage of *MC* is to work also whether the malware is obfuscated, as shown in (Mercaldo et al., 2016a; Mercaldo et al., 2016c). It is possible since *MC* is behavioural based, and trivial transformations of the code do not change the normal behaviour of the code. For example, when an attacker inserts in the code some unconditional jumps (code reordering) it changes only the form of the code but the normal execution flow of the code is preserved. *MC* is not pattern matching, it looks for the malicious behaviour in the form of malicious actions performed. Thus, *MC* is resilient to code obfuscation. Nothing we can say about *ML* and its robustness to code obfuscation since the authors in (Canfora et al., 2016) do not provide any example.

Weaknesses and Disadvantages

The *ML* approach produced values of correctness that are smaller than those obtained with the *MC*, even if the f_4 feature showed performances that are close to those of *MC*. A further weakness of *ML* is the

Table 3: Comparison of the Recall Results.

Family	**Machine Learning Based Approach**					**Model Checking Based Approach**
	Algorithm	f_1	f_2	f_3	f_4	
Plankton	J48	0.608	0.698	0.696	0.694	1
	LadTree	0.608	0.698	0.696	0.694	
	NBTree	0.202	0.178	0.211	0.694	
	RandomForest	0.667	0.675	0.674	0.694	
	RandomTree	0.683	0.681	0.68	0.694	
	RepTree	0.606	0.604	0.609	0.694	
BaseBridge	J48	0.727	0.73	0.741	0.799	0.98
	LadTree	0.024	0.018	0.015	0.841	
	NBTree	0.211	0.214	0.224	0.59	
	RandomForest	0.769	0.775	0.793	0.841	
	RandomTree	0.771	0.775	0.783	0.841	
	RepTree	0.629	0.637	0.628	0.778	
Ransomware	J48	0.766	0.704	0.72	0.896	0.99
	LadTree	0.545	0.655	0.654	0.879	
	NBTree	0.602	0.589	0.702	0.89	
	RandomForest	0.654	0.608	0.714	0.902	
	RandomTree	0.712	0.711	0.743	0.935	
	RepTree	0.612	0.672	0.637	0.872	

Table 4: The Two Methodologies in comparison.

	ML	*MC*
Advantages & Strengths	Completely Automatic	High Correctness
	Low Execution Time	Payload Localization
	Exhaustive Experimentation	Very Small Training Set
Disadvantages & Weaknesses	Not High Correctness	High Execution Time
	Big Training Set	Analyst Involvement
	No Payload Localization	Small Experimentation

required large cardinality of the training set which forces the malware analyst to have a relevant volume of samples to hand out to the machine learning engine. In fact in the *ML* experimentation the authors used a training set that contains the 80% of the collected samples. Furthermore the *ML* does not provide any information about the payload and its localization. The *ML* classifies only a malware in its family. The execution time to analyze a new sample using the features f_1, f_2 and f_3 is in average greater than 10 minutes, which cannot be considered convenient. Finally, these features achieve a very good values of Precision and Recall only in malware detection, while in family identification their average values of exactness and correctness never exceed the 75%, as shown in Figure 1. The *MC* is not completely automatic, since the involvement of an analyst is necessary to specify the formulae. The execution time of *MC* is high, which is in average equal to 60 seconds to check an application; this time is computed on a personal computer with the following computational profile: Intel Core i7 with 2 gigabyte RAM, equipped with Linux Ubuntu 15. This execution time is strictly dependent on the number of furmulae used, the time to build the automaton and the verification time. The time to build the automaton depends on the number of the states and the number of transitions. These two numbers are determined by the complexity of the application's code, i.e. the number of bytecode instructions, the number of `if` statements and cycles. The number of formulae used is proportional to the number of different malicious behaviours that must be caught in the code. Finally, it is worth considering that the experimentation performed by the authors in (Mercaldo et al., 2016a; Mercaldo et al., 2016c) is run on a data-set that is much larger than the one used for the validation of the *MC*. This hinders the comparison of the precision of the two approaches, but allows to have only an indication on how different the performance of the two approaches is. For this reason we computed also the Accuracy of the two approaches, that provides a more reliable comparison. The accuracy histogram in Figure 1 shows the Accuracy values achieved by *ML* and *MC*. It should be underlined that FPs are involved also in the formula of Accuracy. Table 4 shows and summarizes the advantages/strengths and disadvantages/weaknesses of two methodologies *ML* and *MC*. To conclude, the two approaches exhibit several advantages and disadvantages. The malware analyst can choose the right trade-off with agreement to the demands. If the analyst wants to exactly locate the payload within the malware code or wishes a high value of correctness in the family identification, *MC* should be used. However, this approach requires a greater computational time than *ML*. Instead, if the analyst is interested in achieving a high correctness in family identification, is not looking for the payload location, and the efficiency has a priority higher than the effectiveness, the choice should fall on the *ML* with f_4 feature. Finally, if the analyst wants to achieve a high correctness in malware detection, the *ML* should be employed, by using the f_1, f_2 or f_3 features. Unfortunately this will require a longer execution time.

6 CONCLUSIONS

Recognizing malware families (Zhou and Jiang, 2012) is a primary goal of malware analyst and several approaches have been developed to face this issue. In this work we have compared two static approaches. The first one is a Machine Learning based approach, differently the second one is a Model Checking based approach. We have investigated strengths and weaknesses of the two approaches. As future work, we want to compare them with dynamic techniques in order to have a clearer and wider picture.

REFERENCES

Alam, S., Riley, R., Sogukpinar, I., and Carkaci, N. (2016). Droidclone: Detecting android malware variants by exposing code clones. In *2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 79–84.

Anastasi, G., Bartoli, A., De Francesco, N., and Santone, A. (2001). Efficient verification of a multicast protocol for mobile computing. *Computer Journal*, 44(1):21–30. cited By 12.

Annachhatre, C., Austin, T. H., and Stamp, M. (2015). Hidden markov models for malware classification. *J. Computer Virology and Hacking Techniques*, 11(2):59–73.

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

Battista, P., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Identification of android malware families with model checking. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*, pages 542–547.

Baysa, D., Low, R. M., and Stamp, M. (2013). Structural entropy and metamorphic malware. *Journal of Computer Virology and Hacking Techniques*, 9(4):179–192.

Bose, A., Hu, X., Shin, K. G., and Park, T. (2008). Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 225–238, New York, NY, USA. ACM.

Bruns, G. (1997). *Distributed Systems Analysis with CCS*. Prentice-Hall.

Canfora, G., Lorenzo, A. D., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015). Effectiveness of opcode ngrams for detection of multi family android malware. In *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security*, ARES '15, pages 333–340, Washington, DC, USA. IEEE Computer Society.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2016). An hmm and structural entropy based detector for android malware. *Comput. Secur.*, 61(C):1–18.

Cleaveland, R. and Sims, S. (1996). The ncsu concurrency workbench. In *CAV*. Springer.

De Francesco, N., Santone, A., and Tesei, L. (2003). Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundamenta Informaticae*, 54(2-3):195–211. cited By 12.

De Ruvo, G., Nardone, V., Santone, A., Ceccarelli, M., and Cerulo, L. (2015). Infer gene regulatory networks from time series data with probabilistic model checking. pages 26–32. cited By 7.

Faruki, P., Laxmi, V., Bharmal, A., Gaur, M., and Ganmoor, V. (2015). Androsimilar: Robust signature for detecting variants of android malware. *Journal of Information Security and Applications*, 22:66 – 80. Special Issue on Security of Information and Networks.

Feng, Y., Anand, S., Dillig, I., and Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016a). 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 '16, pages 22–28, New York, NY, USA. ACM.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016b). Hey malware, i can find you! In *2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 261–262.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016c). *Ransomware Steals Your Phone. Formal Methods Rescue It*, pages 212–221. Springer International Publishing, Cham.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Santone, A. (2011). Clone detection through process algebras and java bytecode. pages 73–74. cited By 10.

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

Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In *Concurrency: Theory, Language, And Architecture*, pages 2–20.

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., and Blasco, J. (2014). Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Syst. Appl.*, 41(4):1104–1117.

Zhang, M., Duan, Y., Yin, H., and Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1105–1116, New York, NY, USA. ACM.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109.