Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

An HMM and structural entropy based detector for Android malware: An empirical study

*Gerardo Canfora, Francesco Mercaldo *, Corrado Aaron Visaggio*

Department of Engineering, University of Sannio, Benevento, Italy

ARTICLE INFO

Article history:
Received 30 July 2015
Received in revised form 8 February 2016
Accepted 29 April 2016
Available online 6 May 2016

Keywords:
Malware
Mobile
HMM
Entropy
Android

ABSTRACT

Smartphones are becoming more and more popular and, as a consequence, malware writers are increasingly engaged to develop new threats and propagate them through official and third-party markets. In addition to the propagation vectors, malware is also evolving quickly the techniques adopted for infecting victims and hiding their malicious nature to antimalware scanning. From SMS Trojans to legitimate applications repacked with malicious payload, from AES encrypted root exploits to the dynamic loading of a payload retrieved from a remote server: malicious code is becoming more and more hard to detect.

In this paper we experimentally evaluate two techniques for detecting Android malware: the first one is based on Hidden Markov Model, while the second one exploits structural entropy. These two techniques have been successfully applied to detect PCs viruses in previous works, and only one work in literature analyzes the application of HMM to the detection of Android malware. We demonstrate that these methods, which reveal effective for PCs virus, are also successful for detecting and classifying mobile malware.

Our results are promising: we obtain a precision of 0.96 to discriminate a malware application, and a precision of 0.978 to identify the malware family.

1. Introduction

With the growth of smartphones capabilities, malicious software targeting mobile devices is rapidly spreading, and it is getting more and more successful in evading the detection.

In 2013, the growth rate of mobile malware was far greater than the growth rate of new malware targeting PCs (Alcatel Lucent, 2013), for the first time in malware history.

New kinds of malware spread out continuously at a very fast pace, and malware writers refine both the evasion techniques and the techniques for obtaining tangible return from the attacks, in terms of money or damage to the victim (F-Secure, 2015). Unfortunately, current solutions to protect users from new threats are still inadequate (Fraunhofer AISEC, 2013; Visaggio and Mercaldo, 2015). For example, a malware that is plaguing a huge number of devices while the authors are writing this paper is the ransomware (InfoWorld, 2013), which encrypts data stored on the device and holds it for ransom. The information will be released only after the victim pays the required amount, often in bitcoin.

In addition to this, there exist several techniques to allow the mobile malware to evade signature detection (Ramachandran et al., 2012; Rastogi et al., 2013), which makes detection harder.

In the meantime, simple forms of polymorphic attacks targeting Android platform have been seen in the wild (Bayer et al., 2006): the main effect of polymorphism (and metamorphism) is that signature-based detection becomes ineffective.

* Corresponding author. Department of Engineering, University of Sannio, 82100 Benevento, Italy. Tel.: +390824305529.
E-mail address: fmercaldo@unisannio.it (F. Mercaldo).

http://dx.doi.org/10.1016/j.cose.2016.04.009
0167-4048/

That considered, it urges to develop new techniques to detect malware targeting mobile devices.

Recent papers (Attaluri et al., 2008; Baysa et al., 2013) have used the structural entropy to detect metamorphic virus and Hidden Markov Models (HMM) to classify them. We observed that the way Android malware evolves makes it similar to metamorphic malware, in certain regard. As a matter of fact, writers of malware for Android use to modify some existing malware, by adding new behaviors or merging together parts of different existing malware's codes. This explains also why Android malware is usually grouped in families: in fact, given this way of generating Android malware, the malware belonging to the same family shares common parts of code and behaviors.

Considered these similarities, and considered that Structural Entropy and HMM were able to successfully detect metamorphic viruses for personal computer (Attaluri et al., 2008; Baysa et al., 2013), we investigate with this paper whether these two techniques can be effective in recognizing Android malware and the malware families. The fact that these techniques are effective with personal computers' malware does not entail that they are effective also for Android malware.

As a matter of fact, Android presents program's structures and features that make an Android malware different from a PC malware, since these features are leveraged by malware writers for developing techniques of infection, evasion and payload activation that are not used in PC's malware. Examples are the dynamic loading that permits to dynamically add malicious code to an app, the intent based programming that allows techniques of attacks like service or activity hijacking (Chin et al., 2011), and the system of permissions that limits the range of actions a malware can do, but allows attacks like the update attack (Poeplau et al., 2014), which is a very effective and widespread anti-detection technique. These peculiarities of Android lead us to wonder whether structural entropy and HMM hold their effectiveness in detecting malware also when applied to malicious software written for Android. Moreover, at the best knowledge of the authors, only one paper explores the effectiveness of HMM for detecting Android malware (Chen et al., 2014), but the authors obtained lower performances, used a smaller dataset than the one we used in the experiments, and applied HMM on different features from the ones our method relies on.

The experiments we carried out to demonstrate that HMM is effective in recognizing malware, i.e. with a precision of 0.96, while the structural entropy successfully identifies the family a malware belongs to, with a precision of 0.98.

Identifying the family that a malware belongs to is of primary importance as it helps to discover new malware families (Khoo and Lio, 2011; Ma et al., 2006), creates models of provenance and lineage (Dumitras and Neamtiu, 2011), and generates phylogeny models (Karim et al., 2005).

The paper proceeds as follows: the next section provides background notions about HMM and structural entropy and discusses the related work; Section 3 discusses the adoption of HMM and structural entropy methods to detect mobile malware; Section 4 discusses the experimental evaluation; Section 5 illustrates the results of experiments; Section 6 discusses the detection performance of HMM and structural entropy methods; Section 7 explains the threats to validity and, finally, conclusions are drawn in Section 8.

2. Background and related work

Before discussing the state of the art of malware detection using HMM and structural entropy, we recall the essential background about HMM and structural entropy.

2.1. HMM

The Hidden Markov Model (HMM) is a statistical pattern analysis algorithm. HMM uses the following notations:

T = length of the observed sequence;
N = number of states in the model;
M = number of distinct observation symbols;
O = observation sequence $\{O_0, O_1, \dots, O_{T-1}\}$;
A = state transition probability matrix;
π = initial state distribution matrix.

Fig. 1 shows the generic scheme of a Hidden Markov Model, which represents the states and the observations at time t, respectively with X_t and O_t, and the probabilities of transitions among the states, a_{ij} which is the probability of the transition from the state X_i and the state X_j.

The Markov process, which is hidden behind the dashed line, includes an initial state X_0 and the A matrix, i.e. the set of the probabilities of all the transitions among the states.

The only observable part of the process is represented with the O_i; the matrix B contains the probabilities that an observation O_i be related to a state X_i.

Common applications of HMMs are protein modeling (Plotz and Fink, 2005) and speech recognition applications (Kinjo and Funaki, 2006), i.e. identifying whether a protein can be attributed to a known protein structure, or if a speech fragment can be associated to a known speech pattern.

As a machine learner, HMM works in this way: the first step consists of creating a training model that represents the input data (training data).

The training model includes a chain of unique symbols observed within the input data along with their positions in the input sequence.

This model will be used by the HMM to determine if a given new input sequence shows a pattern similar to that found in the model.

In a recent paper (Attaluri et al., 2008), HMM machine learners have been applied to detect metamorphic virus.

Although metamorphic engines change the form of viral copies by employing different code obfuscation techniques, some similar patterns can be found within the same family of virus.

An HMM-based detector gathers the input data from a sample of known virus and builds the training model (one for each family virus) with this input dataset.

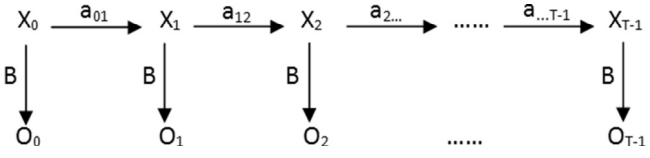

Fig. 1 – The scheme of a Hidden Markov Model.

Subsequently, any file can be tested against these models to determine if it can be considered as belonging to one among the learned models. If an input file belongs to a model, then it is classified as a member of the virus family that the model represents.

For space's reasons we do not provide mathematical details of the HMM, which can be found in the literature (Rabiner, 1989).

2.2. *Structural entropy*

The similarity method considered here was originally introduced in Baysa (Baysa et al., 2013).

The technique performs a static analysis of files using the *structural entropy* (Sorokin, 2011) to evaluate the similarity among Android applications.

Unlikely the HMM method, which retrieves hidden states from opcode sequences, the similarity score is computed directly on the executable file (i.e. the .dex file in the case of Android applications); thus the disassembly step is not needed, and the technique can be applied independently of the specific executable file format, because it ignores header-specific information.

The method of *structural entropy* compares two given files and produces a similarity measure, i.e. evaluates to which extent the two files can be considered similar.

The entropy measure provides a sort of signature of a file, by computing the distribution of bytes within the file. We do not provide the details of the computation, that can be found in (Baysa et al., 2013).

The assumption is that different malware samples of the same family have a similar order of code and data areas; as a matter of fact each area may be characterized not only by its length, but also by its homogeneity. We try with this method to characterize a mobile malware by the complexity of its data areas. Authors in (Sorokin, 2011) identify as *structural entropy* this characteristic of an application; we extend this concept to mobile environment.

The approach consists of using discrete wavelet transform (DWT) for the segmentation of files into segments of different entropy levels and using edit distance between sequence segments to determine the similarity of the files.

The method comprises two steps: file segmentation and sequence comparison.

The first step splits each file into segments of varying entropy levels using wavelet analysis (Baysa et al., 2013) applied to raw entropy measurements.

Wavelet analysis (Addision, 2002) is aimed at transforming a signal (a collection of observation points, in our cases the dataset) into a form that provides greater opportunities of analysis.

A wavelet is in fact a wavelike function that can be used to analyze data in different locations and at different scales.

Scaling a wavelet allows to reduce the high frequency of information present in the original dataset without reducing the information content and the possibilities of analysis.

Once the files have been segmented, for each given couple of files to compare, a similarity score is produced by computing the edit distance between all the corresponding couples of segments.

Wavelet analysis is useful to locate those areas in a file where significant changes in entropy occur.

A sliding window is applied to extract the corresponding series of entropy levels using Shannon formula (Borda, 2011).

After this, we are able to evaluate the similarity between the two files, using the extracted segments. As in (Sorokin, 2011), we use the Levenshtein distance, or the edit distance (Baysa et al., 2013), to determine the scores between two sequences, that in our case are represented by two files.

Using edit distance we evaluate the minimum number of edit operations required to transform one sequence into the other.

The possible edit operations are substitution, insertion, and deletion. The substitution operation consists of replacing an element from the first sequence with an element in the second, while the insertion consists of inserting a new element into the second sequence. Finally, a deletion operation eliminates an element from the second sequence.

Greater details can be found in (Baysa et al., 2013).

2.3. *Related work*

Although HMM has been explored as technique for detecting malware for personal computers, at the best knwoledge of the authors only one paper investigated HMM for the detection of malware for smartphones (Xie et al., 2010) and only one concerns specifically Android platform (Chen et al., 2014).

Chen et al. (Chen et al., 2014) examine Android Intent messages at run-time, thus they build a hidden Markov model (HMM) in order to detect apps runtime malicious behavior. The main difference with our work is that we apply static analysis, i.e. based on opcodes sequences, while Chen et al. (Chen et al., 2014) use a dynamic analysis, which has the limit of requiring the execution of the app for establishing whether it is a malware or not. On the contrary static analysis can be applyed directly on the app. As stated by the authors, their method did not obtain high performances: 0.7 of precision, while our method is able to reach 0.96 of precision. Additionally, our malware dataset was much larger (6,192) than the one used in reference (Chen et al., 2014).

Xie et al. (Xie et al., 2010) propose a behavior-based malware detection system named pBMDS, which employs a statistical approach to learn within cellphones the behavioral difference between user initiated applications and malware compromised ones. The novelty of this approach stands in the fact that it focuses on recognizing non-human behavior instead of relying on known signatures to identify malware. They propose a Hidden Markov Model (HMM) based malware detection engine which takes only a limited number of observations (user inputs) in input and associates process states (hidden states that cannot be directly observed) and their transitions with these observations. For the experimentation they use Linux-based smartphones, thus they do not apply the technique to Android platform and this is the first difference with our work. Furthermore they implemented three ad-hoc malicious apps for validating the method, while we used 6,192 malware taken from the real world. Even if they obtain a good detection rate and a low false positive and negative rate, they did not test their method with goodware (in order to understand how many trusted applications are recognized as

malware), while we validated our method also on trusted applications.

For completeness, we provide here an overview of the literature about Android malware detection, while later we discuss the related literature about HMM, structural entropy, and machine learning applied to malware detection.

Approaches aiming at identifying the malicious behavior of the app through sequences of system or API calls have been proposed by many authors, as in (Canfora et al., 2015; Ki et al., 2015; Wei et al., 2013). One of the main limits of these approaches is that the malware writer can alter the original sequence of system or API calls leaving effective the app, but making ineffective the detection. SherlockDroid (Apvrille and Apvrille, 2015) is a classification engine which uses a large set of features and it is able to recognize specifically unknown malware, while our aim was also to classify the family a malware belongs to.

Yerima et al. (Yerima et al., 2015) apply the ensemble machine learning for detecting Android zero-day attacks, which leverages an extensive feature-based approach. Even if the method produced a very high precision (99%), it does not allow the classification of malware families.

Detecting zero-day attacks on Android platforms is also the goal of Sayfullina et colleagues (Sayfullina et al., 2015). They explored several techniques for tackling independence assumptions in Naive Bayes and proposed Normalized Bernoulli Naive Bayes classifier that resulted in an improved class separation and higher accuracy. They conducted a set of experiments on an up-to-date large dataset of APKs provided by F-Secure (an anti-malware producer) and achieved 0.1% false positive rate with overall accuracy of 91%, which is smaller than the precision reached by our method.

Munoz et al. (Munoz et al., 2015) show that modern Machine Learning techniques applied to collected metadata from Google Play can provide a first approach towards the detection of malware applications, and they further identify which features have the highest predictive power among the total. Of course, the technique can be evaded easily by altering properly the metadata.

2.3.1. *HMM-related methods*

Xin (Xin et al., 2012) and Qin (Qin et al., 2011) propose a mobile malware detector based on an HMM model using system call traces. In (Xin et al., 2012) authors monitor the keys pressed and the system function call sequences, where the pressed keys represent the hidden states while the system call sequences represent the observations. This proposed solution is evaluated on a single Symbian application, with a specific focus on the sms sending process, while our solution is aimed at monitoring the overall malware behaviour. In (Qin et al., 2011) researchers propose a prototype of HMM-based detection system but they do not evaluate it.

Other approaches use dynamic analysis to build the model, i.e. require the execution of the mobile applications. These approaches differ from the one proposed in this paper, as we train the machine learner with models obtained by static analysis.

Authors in (Attaluri et al., 2008) propose HMM to train a metamorphic malware detector. They evaluate their solution building a dataset with three different virus construction kits (VCL32, PS-MPC and NGVCK) to generate multiple variants for each family, for a total of 240 virus variants and 70 trusted samples between DLLs and applications. Experiments show a 100% detection rate for VLC32 and PS-MPC, but regarding NGVCK they do not obtain an useful result.

HMM revealed to be effective (Attaluri et al., 2008) in classifying virus, but it is ineffective in recognizing a virus when a quota higher of 35% of dead code (taken from trustful programs) is added to the virus code.

2.3.2. *Entropy-related methods*

The similarity method presented in (Baysa et al., 2013) is applied to binary files, and does not require a pre-processing phase like disassembling.

In the evaluation the authors consider three metamorphic families: 50 viruses generated by G2 virus construction kit, 50 by NGVCK virus construction kit and a worm family, developed by authors, able to evade statistical opcode-based detection techniques (MWOR) (Chouchane et al., 2013).

Authors demonstrate that structural entropy is useful to classify metamorphic malware of G2 and MWOR family, but at high percentage of trusted code injected the MWOR family cannot be detected; results for NGVCK metamorphic family depends on the number and the length of segments.

Longer files will tend to produce more segments, so the score is sensitive to file length. Since the NGVCK files differ significantly in length, authors conclude that this may be the cause of lack of success with this family.

Structural entropy showed to be effective (Baysa et al., 2013) in detecting the families of metamorphic virus, but it revealed to be ineffective with certain virus families: the weakest point of this technique was that virus may successfully evade it by morphing the code.

As explained before, the structural entropy score depends heavily on the segment length and the number of segments selected, and consequently even the success of this technique depends on tuning properly these properties.

Ugarte-Pedrero (Ugarte-Pedrero et al., 2012) and colleagues propose a method to measure entropy for ciphered data. Their solution is evaluated on a dataset formed with Zeus family samples, a real malware for PC. They obtain the best results for a region of 128 bytes with an accuracy of 0.952.

Lyda and Hamrock (Lyda and Hamrock, 2007) discuss the entropy approach adoption to discovery packed and encrypted malware, proposing a set of metrics that analysts can use to distinguish the packed or encrypted executable from non-packed or unencrypted ones.

They develop Bintropy, a prototype tool that computes the entropy score of blocks, the average and the highest entropy scores from binary files to estimate the likelihood that a binary file contains compressed or encrypted bytes. Their experimentation was aimed at determining the entropy metrics relying on the computed difference intervals for the average and the highest entropy.

At the best of the authors knowledge, the mentioned papers represent the only work in literature that applies HMMs and Structural Entropy to malware detection.

2.3.3. *ML-related methods*

Machine learning is largely applied to detect malware. MDoctor (Lagerspetz et al., 2014) determines the "health" of a device

based on several indicators: the authors use application market trust, and developer key trust, as parameters for determining the correlation with known malware. The authors do not discuss the evaluation of the proposed solution.

Canfora et al. (Canfora et al., 2013) propose a method for detecting Android malware based on three metrics: the occurrences of a specific set of system calls, a weighted sum of a subset of permissions, and a set of combinations of permissions. They evaluate the proposed solution with a dataset of 400 applications (200 malware and 200 trusted) obtaining a precision of 0.74. The main differences with this work is the application of HMM and structural entropy, combined with the use of static analysis for extracting the features in place of the dynamic analysis; furthermore, the dataset is significantly enlarged.

Droid Detective (Liang and Du, 2014) classifies an Android application by using a technique based on permissions combination. The evaluation with a dataset of 1,260 malware and 741 benign produced a detection rate respectively of 96% and 88% for malware and benign recognition.

Lui et al. (Liu and Liu, 2014) propose another permission-based approach: they extract requested and used permissions and make combinations of them to build a J48 classifier to test their dataset containing 28,548 benign and 1,563 malicious applications. Their evaluation obtains a 0,898 precision.

Arp et al. (Arp et al., 2014) propose a method to perform a static analysis of Android applications based on features extracted from the manifest file and from the disassembled code (suspicious API calls, network addresses and other). Their approach uses Support Vector Machines to produce a detection model, obtaining a precision equal to 0.94 by using a dataset formed by 123,453 trusted and 5,560 malware applications.

MAST (Chakradeo et al., 2013) extracts attributes from mobile applications using the correlation between multiple data (permissions, intents, native code information and questionnaire); the method is tested with 15,000 trusted and 732 malicious applications.

Yerima et al. (Yerima et al., 2013) present Bayesian classification models obtained from static analysis. They extract 20 features from 2,000 application (1,000 malware and 1,000 trusted) to build the models, obtaining a precision rate equal to 0.944.

DroidLegacy (Deshotels et al., 2014) classifies Android malware extracting families signatures with a precision rate of 87% from their dataset formed by 1,052 malicious applications and 48 benign ones.

Apposcopy (Feng et al., 2014) groups Android malware by using a semantic-based approach, a static taint analysis and a call graph inter components; authors evaluate their solution with 1,027 malware obtaining an accuracy of 90%.

DroidDolphin (Wu and Hung, 2014) performs a static and a dynamic analysis in order to extract features from network access, api calls, achieving a prediction accuracy of 86.1% with a balances dataset composed by 32,000 trusted and 32,000 malicious applications using an SVM classifier. The api calls trace requires the app instrumentation.

AndroSimilar (Faruki et al., 2013) aims to find regions of statistical similarity starting from the .dex file. Authors obtain an accuracy of 72.27% using a dataset of 101 malicious applications. Among the methods appying ML, this is the closest one to the method we applied in this paper.

Our work enriches the existing literature with a twofold contribution: (i) the study of HMM and structural entropy for recognizing malware and the family the malware belongs to; (ii) the validation of the method is carried out on a very large dataset, including 6,192 malware and 5,560 trusted samples, recently collected from the real world.

3. Adopting HMM and structural entropy malware detection for android

In order to use HMM and structural entropy to detect Android malware, their original application to the PC's metamorphic malware detection was modified, as described in this section.

3.1. *HMM as malware detection tool*

HMM-based malware detection requires a training dataset to produce a model.

The goal is to train several HMMs to represent the statistical properties of the full malware dataset and of malware families.

Trained HMMs can then be used to determine if an application is similar to malware (families) contained in the training set.

A model is produced for each family of malware by collecting only the malware belonging to a single family, because a previous study (Attaluri et al., 2008) demonstrates the efficacy of this choice in metamorphic malware detection.

Alternatively, the HMM detector could be trained by using the overall malware dataset, without distinguishing among the families.

Our purpose is to extract the sequence of instructions from each app of the training dataset which better represents a possible execution of the app itself.

Once obtained such a sequence, we need to convert it in a corresponding sequence of symbols. For us, the symbols will be represented by the opcodes of the instructions.

For obtaining the opcodes, a malware app is first converted into the correspondent smali (Paller, 2015) code.

To do this, we use smali/baksmali (Anon., 2015), the state of art for apk disassembler.

Smali opcodes found in the applications will constitute the HMM symbols.

We concatenate the opcode sequences (obtained by all the malware apps of the dataset) into a unique observation sequence.

The opcode sequence is obtained with the following process: we search the entry point of each application in the correspondent Manifest file, we extract all the opcodes of all the called methods, in sequence, starting from the entry point.

When we find an "invoke" instruction, we jump to the invoked method, and collect all the opcodes of all the instructions forming that method.

The process stops when we reach a class of the Android framework, or when we reach the maximum recursion level, fixed to 4. This threshold was established for convenience

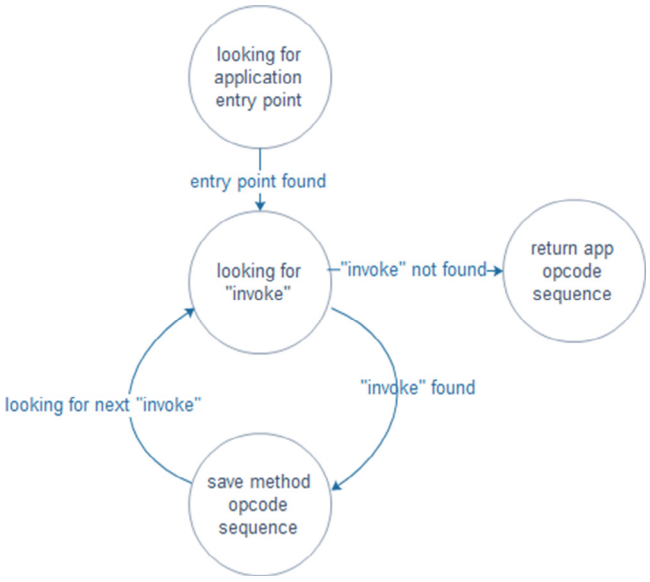

Fig. 2 – State diagram of opcode sequence extraction.

reason, as a tradeoff between efficacy (as complete sequence of collected instructions as possible) and the cost of computation. The complete process is represented in Fig. 2.

Once extracted the opcode sequence from each application, the HMM detector must be trained: all the opcode sequences obtained by all the apps are merged together to form one file, i.e. a single sequence of opcodes.

In order to train the HMM detector, we need to fix the number of the hidden states (of the generated HMMs).

Thus we apply the Baum-Welch algorithm (Attaluri et al., 2008) for finding the values of the unknown parameters of the model. Each time the number of hidden states changes, the algorithm is run again.

After this step, we compare a generic opcode sequence (corresponding to the app to classify) with the opcode sequence of the learned model: we use the Forward algorithm (Attaluri et al., 2008) for finding the likelihood of the sequence, i.e. the probability that the (learned) model generates the sequence to classify. If this is true, then the app is considered a malware instance. Of course, the "definition" of a malware depends strictly on the used training dataset.

We trained our HMMs using different numbers of states and examined the resulting probabilities to deduce which features the states represent. The number of hidden states N that we tested are $N = 3, 4, 5$.

3.2. *Entropy-based detection*

The entropy-based method is based on the estimation of structural entropy of an Android executable (.dex file).

The first step is the extraction of an entropy series: once the .dex file has been divided into blocks of fixed size, we compute the Shannon entropy for each block.

This is the first representation of .dex file; in order to obtain the segments of the file, we use the wavelet transform which gives an useful representation of the entropy series, with approximation and detail coefficients.

Once two parameters, minimum and maximum scale of the wavelet transform, have been selected, we use the detail coefficients to detect the discontinuities in the entropy series and extract the segments:

- the length of each segment is represented by the distance between two consecutive discontinuities;
- the entropy value of each segment is the approximation value of the entropy series between the discontinuities.

The output of the segmentation phase is represented by the list of segments that represent the different entropy areas in .dex file.

The second phase of the method is the comparison between the segments of two .dex files to compute a similarity score.

As we mentioned before, the similarity score is based on the Levenshtein distance. This value represents the percentage of similarity between two .dex files based on the corresponding entropy areas.

4. Experimental evaluation: study definition

In this section we discuss the experiments we carried out to evaluate the effectiveness of the HMM and structural entropy in detecting Android malware and correctly classifying the family a malware belongs to.

4.1. *Research questions*

The paper poses four research questions:

- RQ1: is an HMM based detector able to discriminate a malware from a trusted application for smartphones?
- RQ2: is an HMM based detector able to identify the family of a malware application?
- RQ3: is the structural entropy similarity able to discriminate a malware from a trusted application?
- RQ4: is the structural entropy similarity able to identify the family of a malware application?

4.2. *The dataset*

A dataset made of 5560 trusted and 5560 malware Android applications was collected: trusted applications by different categories (call & contacts, education, entertainment, GPS & travel, internet, lifestyle, news & weather, productivity, utilities, business, communication, email & SMS, fun & games, health & fitness, live wallpapers, personalization) were downloaded from Google Play (and then controlled by Google Bouncer, a Google service that checks each app for malicious behaviour before publishing it on the official market (Busticati Productions Presents, 2015)) from July 2014 to September 2014, while malware applications of different nature and malicious intents (premium call & SMS, selling user information, advertisement, SMS spam, stealing user credentials, ransom) were taken from Drebin Dataset (Arp et al., 2014; Spreitzenbarth et al., 2013).

Every family contains samples which have in common several characteristics, like payload installation, the kind of

Table 1 – Number of samples for family in Drebin dataset with details of the installation method (standalone, repackaging, update), the kind of attack (trojan, botnet), the events that trigger the malicious payload and a brief family description.

Family	Inst.	Attack	Activation	Samples	Description
FakeInstaller	s	t,b		925	Server-side polymorphic family
Plankton	s,u	t,b		625	It uses class loading to forward details
DroidKungFu	r	t	boot,batt,sys	667	It installs a backdoor
GinMaster	r	t	boot	339	Malicious service to root devices
BaseBridge	r,u	t	boot,sms,net,batt	330	It sends information to a remote server
Adrd	r	t	net,call	91	It compromises personal data
Kmin	s	t	boot	147	It sends info to premium-rate numbers
Geinimi	r	t	boot,sms	92	First Android botnet
DroidDream	r	b	main	81	Botnet, it gained root access
Opfake	r	t		613	First Android polymorphic malware

attack and events that trigger malicious payload (Canfora et al., 2015; Zhou and Jiang, 2012) (Table 1).

Furthermore we test our methods using a dataset containing 632 real world samples labelled as ransomware, koler, locker, fbilocker and scarepackage (Andronio et al., 2015). The ransomware dataset cointains samples appeared in December 2014–January 2015.

The full malware dataset is composed of 6192 samples.

In order to answer to RQ1 and RQ3 we use the full dataset; for RQ2 and RQ4 we selected the 11 families owning the greatest number of samples (the full malware dataset contains samples from 179 families), including the ransomware samples, in order to obtain more significant and reliable outcomes.

4.3. *The analysis method*

We extracted 3 features (f_1, f_2, f_3) with the HMM method, and one feature (f_4) with the structural-entropy method.

HMM training was accomplished by using cross-validation, specifically the five-fold cross validation. With the five-fold cross validation, we divide the data set into five equalized subsets. Each time we train a model, we choose one of the subsets as the test set and train the model using the dataset formed merging the other four subsets. Because the dataset used in the test set is not used during the training phase, we can use it to evaluate the performance of the model over unseen instances of the same virus. By repeating this process five times and choosing a different subset as the test set each time, we can produce five different models with the same dataset.

Table 2 shows the number of samples used in the training set and in the testing set for each family considered.

We considered the full malware dataset to answer RQ1 and the remaining 10 subsets shown in Table 1 and the ransomare dataset to answer RQ2. The training process can be summarized in this way:

1. given a data set consisting of different malware instances, we pick one subset as the test set and use the remaining four subsets for the training;
2. train HMM for the sequences present in the training set until the log likelihood of the training sequence converges or a maximum number of iterations is reached;
3. compute the score, i.e., the log likelihood of malware in the test set and other files in the comparison set;
4. repeat from (1), choosing a different subset as the test set, until all five subsets have been chosen.

Each training was performed for $N = 3, 4, 5$ hidden states.

When the training process is over, every app in the dataset is associated to a score for the subset considered.

Using HMM with 3, 4 and 5 hidden states, as result we have three scores, i.e. three features for every app (f_1 = HMM score with 3 hidden states, f_2 = HMM score with 4 hidden states, f_3 = HMM score with 5 hidden states).

With regard to the structural entropy, we want to evaluate the similarity of an application X with a population X_p; we first need to compute the structural entropy of X and the structural entropy of X_p. Once we have these two values, we can compute the similarity between them and obtain an evaluation of the similarity between (the structural entropy of) X and (the structural entropy of) X_p. The structural entropy of X_p is the arithmetic mean of the structural entropies of all the applications X_{pi} belonging to the population X_p.

In particular, for answering RQ3 we evaluated:

- the similarity between each malware application (X) and the full malware dataset (X_p); and
- the similarity between each trusted application (X) and the full trusted dataset (X_p).

In order to answer RQ4 we computed:

Table 2 – The number of samples for each family considered in the HMM experiments.

Family	Training set	Test set
FakeInstaller	740	185
Plankton	500	125
DroidKungFu	534	133
GinMaster	272	67
BaseBridge	264	66
Adrd	73	18
Kmin	118	29
Geinimi	74	18
DroidDream	65	16
Opfake	491	122
Ransomware	506	126
Total malware	4954	1238

The test set is approximately the 20% of subset considered.

- the similarity between each malware application (X) and the full set of applications belonging to the same family (X_p).

When the process is over, every app in the dataset is associated to a score for the subset considered (f_4 = structural entropy, as defined previously).

Similarly to HMM, also for the structural entropy we considered the full malware dataset to answer RQ3 and the 11 subsets in Table 1 to answer RQ4: the reason for this choice has been explained previously.

Two kinds of analysis were performed: hypothesis testing and classification.

The test of hypothesis was aimed at understanding whether the features the model consists of are able to produce a statistically significant difference between the two samples: malware and trusted.

The classification was aimed at determining whether the features extracted allowed to associate correctly an application to the malware class or the trusted class, and whether the features allowed to correctly associate each malware to the family it really belongs to.

After extracting the features we tested the following null hypothesis:

H0. Malware and trusted applications have similar values of the proposed features.

The H_0 states that, given the *i-th* feature f_i, if f_{iT} denotes the value of the feature f_i measured on a trusted application, and f_{iM} denoted the value of the same feature measured on a malicious application:

$$\sigma(f_{iT}) = \sigma(f_{iM}) \quad \text{for} \quad i = 1, \ldots, 4$$

being $\sigma(f_i)$ the means of the (control or experimental) sample for the feature f_i.

The null hypothesis was tested with Mann–Whitney (with the p-level fixed to 0.05) and with Kolmogorov–Smirnov Test (with the p-level fixed to 0.05). Two different tests of hypotheses were performed in order to have a stronger internal validity.

The classification analysis was aimed at assessing whether the features where able to correctly classify malicious and trusted applications.

The algorithms were applied to each of the four features.

Six algorithms of classification were used: J48, LadTree, NBTree, RandomForest, RandomTree, RepTree. Similarly to hypothesis testing, different algorithms for classification were used for strengthening the internal validity.

5. Analysis of results

The hypothesis test produced evidence that the considered features have different distributions in the control and experimental sample, as shown in Table 3. As a matter of fact, all the p-values are under 0.001.

Summing up, the null hypothesis can be rejected for the features f_1, f_2, f_3 and f_4. According to the hypothesis tests, both the two methods, HMM and structural entropy are able to distinguish a malware from a trusted app.

Table 3 – Results of the test of the null hypothesis H_0.

Variable	Mann–Whitney	Kolmogorov–Smirnov
f_1	0.000000	$p < .001$
f_2	0.000000	$p < .001$
f_3	0.000000	$p < .001$
f_4	0.000000	$p < .001$

With regard to classification, we define the training set *T*, consisting of a set of labeled applications (*AUA, l*) where the label $l \in \{trusted, malicious\}$. For each AUA, i.e. application under analysis, we built a feature vector $F \varepsilon R^y$, where y is the number of the features used in training phase ($1 \le y \le 4$).

To answer RQ1 we performed three different classifications, each one with a single feature: f_1, f_2 and f_3 ($y = 1$), while for RQ2 we performed ten classifications with f_1m, f_2m and f_3m where m represents the malware family ($0 < m < 11$), in this case the label $m \in \{FakeInstaller, Plankton, DroidKungFu, GinMaster, BaseBridge, Adrd, KMin, Geinimi, DroidDream, Opfake, Ransomware\}$ (each classification is accomplished with a single feature, as in the previous case).

To answer RQ3, we performed the same classification as for RQ1; the only difference is the feature used: in this case we used the structural entropy feature (f_4).

To answer RQ4 we performed eleven classifications, each one with the structural entropy feature (f_4) computed for the selected 10 malware families and ransomware dataset.

We used the k-fold cross-validation: the dataset was randomly partitioned into k subsets of data. A single subset of the dataset was retained as the validation data for testing the model, while the remaining k – 1 subsets were used as training data. We repeated this process k times, each of the k subsets of data was used once as validation data. To obtain a single estimate we computed the average of the k results from the folds.

Specifically, we performed a 10-fold cross validation.

Results are shown in Tables 4 and 5. The rows represent the features, while the columns represent the values of the three metrics used to evaluate the classification results (precision, recall and roc-area) for the recognition of malware and trusted samples. The Recall has been computed as the proportion of examples that were assigned to class X, among all examples that truly belong to the class, i.e. how much part of the class was captured. The recall is defined as:

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

where *tp* indicates the number of true positives and *fn* is the number of false negatives.

The Precision has been computed as the proportion of the examples that truly belong to class X among all those which were assigned to the class, i.e.:

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

where *fp* indicates the number of false positives.

Table 4 – Precision, Recall and RocArea of 3-HMM (f_1), 4-HMM (f_2), 5-HMM (f_3) detector and structural entropy (f_4) based detector for malware classification with the algorithms J48, LadTree, NBTree, RandomForest, RandomTree and RepTree.

Feature	Algorithm	Precision	Recall	RocArea
f_1	J48	0.933	0.997	0.581
	LADTree	0.933	0.997	0.717
	NBTree	0.933	0.997	0.727
	RandomForest	0.948	0.97	0.722
	RandomTree	0.949	0.97	0.683
	RepTree	0.935	0.995	0.688
f_2	J48	0.935	0.996	0.579
	LADTree	0.935	0.997	0.712
	NBTree	0.935	0.996	0.727
	RandomForest	0.948	0.97	0.706
	RandomTree	0.951	0.97	0.674
	RepTree	0.937	0.995	0.735
f_3	J48	0.953	0.996	0.586
	LADTree	0.951	0.998	0.713
	NBTree	0.957	0.996	0.727
	RandomForest	0.96	0.968	0.707
	RandomTree	0.955	0.968	0.671
	RepTree	0.952	0.994	0.739
f_4	J48	0.725	0.525	0.702
	LADTree	0.783	0.41	0.725
	NBTree	0.719	0.515	0.701
	RandomForest	0.772	0.826	0.715
	RandomTree	0.777	0.825	0.723
	RepTree	0.747	0.697	0.712

The Roc Area is the area under the ROC curve (AUC) and is defined as the probability that a randomly chosen positive instance is ranked above randomly chosen negative one.

The classification analysis with the HMMs and structural entropy features suggests several considerations. First of all, HMM outperforms structural entropy in discriminating malware from trusted, both in terms of precision and recall.

The precision obtained with the HMM based classifier ranges from 0.933 to 0.96, while the recall spans between 0.97 to 0.998; on the contrary, the entropy based classifier's precision has values all around 0.7, while the greatest recall is 0.8.

It seems that the performances of the HMM based detector could depend on the number of hidden states, as precision increases with the number of hidden states. The ROC AREA values signal that the accuracy is fair, but it will be desirable to have a value of ROC AREA of 0.9 for having a perfect test.

However ,the differences in performances among 3, 4 and 5 states of the HMM detector are not that significant. This result is consistent with the one obtained with metamorphic malware (Attaluri et al., 2008).

Regarding the detection of the malware's families, whose results are reported in Table 5, the structural entropy reveals to be the best feature to use in the classification, reaching in many cases values of precision greater than 0.9. This is not an unexpected outcome, as the structural entropy is a measure of similarity between executable files, and apps belonging to the same family are supposed to share some parts of the code. Moreover, structural entropy seems to be very effective for some families, like *Opfake*, *Kmin*, *DroidKungFu*, and less for others, but even the smallest values of precision and recall remain enough acceptable, respectively 0.723 and 0.59. It is important to observe that the values of ROC AREA for the structural entropy are high, in many cases over 0.9. So the accuracy of the classification is perfect for the structural entropy, except for *Gemini* and *DroidDream* families.

HMM is not effective for detecting malware families, even if the values of precision and recall in some cases are close to 0.8, which represents not a totally bad performance. It is worth noticing that in this case the ROC AREA is close to 0.9 for many tests which use HMM. This implies that for recognizing malware families, the tests with HMM have a good accuracy. The structural entropy performs better in recognizing the malware families with polymorphic malware: as a matter of fact the best outcomes are obtained with *Opfake*, that is polymorphic. This is consistent with the results of similar studies on PC's viruses (Attaluri et al., 2008; Baysa et al., 2013).

In the following, we respond in detail to the research questions and support answers with descriptive statistics and the results of the classification.

5.1. RQ1: is an HMM based detector able to discriminate a malware from a trusted application for smartphones?

For answering RQ1 it is helpful to examine the scatter plots for HMM likelihood, as illustrated in Figs. 3–5.

From the plotted data it emerges that using 5 hidden states rather than 3 and 4, the HMM detector is more effective, since the distinction between the region of malware (in red) and the region of trusted (in blue) is greater. However, in all the analyzed cases (HMM with 3, 4 and 5 hidden states) the scatter plots show that there is a clear separation between the group of the malware samples and that of the trusted samples.

These results suggest that the best classification will be obtained with a 5 hidden states HMM detector.

The classification analysis, as shown in Table 6, confirms our expectations that HMM can provide very good indicators to discriminate a malware from a trusted Android application.

As a matter of fact we obtained a precision of 0.949 with RandomTree algorithm regarding f_1 (HMM with 3 hidden states), a precision of 0.951 with RandomTree algorithm regarding f_2 (HMM with 4 hidden states) and a precision of 0.96 with RandomTree algorithm using f_3 feature (HMM with 5 hidden states).

RQ1 Summary: the classification analysis shows that the f_3 feature (HMM with 5 hidden states) is the best in class to discern a malware Android application from a trusted one, with a precision of 0.96 obtained with the algorithm RandomForest.

5.2. RQ2: is an HMM based detector able to identify the family of a malware application?

Figs. 6–8 show the box plots for the malware families analyzed.

By looking at the box plots it seems that there are not substantial differences among different families of malware: this result will be reflected into the classification. There is a little increment of likelihood value in the box plot obtained by training an HMM with 5 hidden states, even if this increment is minimal.

Table 5 – Precision, Recall and RocArea obtained by classifying malicious families dataset, with the algorithms J48, LadTree, NBTree, RandomForest, RandomTree and RepTree.

Feature	Algorithm	Precision				Recall				RocArea			
		f_1	f_2	f_3	f_4	f_1	f_2	f_3	f_4	f_1	f_2	f_3	f_4
FakeInstaller	J48	0.706	0.609	0.695	0.854	0.698	0.731	0.704	0.767	0.886	0.881	0.887	0.938
	LADTree	0	0.115	0.119	0.854	0	0.1	0.069	0.767	0	0.603	0.588	0.938
	NBTree	0.729	0.737	0.703	0.854	0.233	0.222	0.229	0.767	0.712	0.707	0.72	0.938
	RandomForest	0.722	0.739	0.741	0.854	0.774	0.777	0.78	0.767	0.886	0.888	0.888	0.938
	RandomTree	0.73	0.735	0.744	0.854	0.776	0.78	0.78	0.767	0.885	0.888	0.887	0.938
	RepTree	0.609	0.625	0.606	0.854	0.588	0.591	0.593	0.767	0.88	0.883	0.885	0.938
Plankton	J48	0.649	0.671	0.645	0.734	0.608	0.698	0.696	0.694	0.895	0.889	0.889	0.821
	LADTree	0	0.091	0	0.734	0.608	0.698	0.696	0.694	0.895	0.889	0.889	0.821
	NBTree	0.649	0.629	0.649	0.734	0.209	0.178	0.211	0.694	0.722	0.721	0.727	0.821
	RandomForest	0.663	0.674	0.677	0.734	0.667	0.675	0.674	0.694	0.897	0.889	0.888	0.821
	RandomTree	0.67	0.675	0.683	0.734	0.683	0.681	0.68	0.694	0.889	0.888	0.887	0.821
	RepTree	0.605	0.632	0.598	0.734	0.606	0.604	0.609	0.694	0.896	0.893	0.891	0.821
DroidKungFu	J48	0.65	0.661	0.646	0.88	0.744	0.762	0.733	0.98	0.888	0.708	0.866	0.928
	LADTree	0.094	0.114	0.099	0.774	0.602	0.629	0.913	0.974	0.566	0.569	0.582	0.805
	NBTree	0.113	0.134	0.119	0.918	0.661	0.666	0.941	0.93	0.7	0.701	0.703	0.891
	RandomForest	0.775	0.772	0.776	0.916	0.778	0.794	0.775	0.93	0.887	0.895	0.887	0.918
	RandomTree	0.772	0.768	0.78	0.918	0.78	0.796	0.783	0.93	0.879	0.885	0.88	0.891
	RepTree	0.506	0.502	0.502	0.876	0.677	0.71	0.679	0.978	0.886	0.889	0.885	0.907
GinMaster	J48	0.685	0.72	0.748	0.834	0.688	0.706	0.699	0.865	0.884	0.883	0.887	0.821
	LADTree	0	0.139	0.089	0.834	0	0.042	0.009	0.865	0	0.62	0.631	0.821
	NBTree	0.701	0.761	0.732	0.834	0.224	0.217	0.214	0.865	0.699	0.705	0.714	0.821
	RandomForest	0.766	0.76	0.777	0.834	0.768	0.76	0.775	0.865	0.883	0.881	0.888	0.821
	RandomTree	0.767	0.775	0.776	0.834	0.771	0.773	0.779	0.865	0.883	0.884	0.887	0.821
	RepTree	0.607	0.646	0.644	0.834	0.597	0.597	0.596	0.865	0.884	0.882	0.887	0.821
BaseBridge	J48	0.627	0.636	0.651	0.89	0.727	0.73	0.741	0.799	0.886	0.889	0.891	0.938
	LADTree	0.344	0.107	0.2	0.864	0.024	0.018	0.015	0.841	0.64	0.643	0.647	0.935
	NBTree	0.714	0.756	0.958	0.887	0.211	0.214	0.224	0.59	0.683	0.682	0.693	0.709
	RandomForest	0.759	0.775	0.786	0.864	0.769	0.775	0.793	0.841	0.883	0.887	0.89	0.935
	RandomTree	0.766	0.788	0.785	0.869	0.771	0.775	0.783	0.841	0.887	0.892	0.896	0.923
	RepTree	0.594	0.608	0.585	0.869	0.629	0.637	0.628	0.778	0.887	0.892	0.896	0.94
Adrd	J48	0.638	0.609	0.643	0.763	0.735	0.731	0.745	0.307	0.89	0.881	0.886	0.863
	LADTree	0	0	0.099	0.875	0	0	0.01	0.782	0	0	0.632	0.779
	NBTree	0.758	0.785	0.739	0.875	0.268	0.267	0.274	0.782	0.703	0.704	0.707	0.779
	RandomForest	0.77	0.772	0.782	0.875	0.775	0.765	0.77	0.782	0.889	0.882	0.885	0.779
	RandomTree	0.778	0.783	0.788	0.875	0.777	0.767	0.77	0.782	0.886	0.882	0.883	0.779
	RepTree	0.581	0.622	0.593	0.875	0.638	0.644	0.637	0.747	0.886	0.885	0.887	0.779
Kmin	J48	0.725	0.732	0.685	0.976	0.68	0.688	0.68	0.752	0.889	0.885	0.885	0.85
	LADTree	0	0.11	0	0.974	0	0.05	0	0.777	0	0.634	0.624	0.991
	NBTree	0.633	0.592	0.538	0.978	0.215	0.167	0.157	0.702	0.715	0.714	0.722	0.817
	RandomForest	0.739	0.742	0.745	0.974	0.771	0.759	0.764	0.777	0.887	0.88	0.883	0.991
	RandomTree	0.738	0.744	0.745	0.778	0.775	0.764	0.768	0.865	0.885	0.887	0.886	0.991
	RepTree	0.614	0.647	0.573	0.946	0.58	0.585	0.605	0.747	0.887	0.887	0.886	0.988
Geinimi	J48	0.608	0.635	0.645	0.723	0.687	0.684	0.694	0.879	0.882	0.881	0.88	0.665
	LADTree	0.255	0.116	0.131	0.725	0.01	0.012	0.035	0.875	0.592	0.593	0.622	0.665
	NBTree	0.611	0.638	0.607	0.723	0.234	0.222	0.229	0.879	0.703	0.706	0.716	0.665
	RandomForest	0.636	0.66	0.645	0.723	0.662	0.76	0.754	0.879	0.88	0.881	0.878	0.665
	RandomTree	0.639	0.641	0.633	0.725	0.634	0.765	0.762	0.875	0.879	0.88	0.878	0.665
	RepTree	0.633	0.611	0.596	0.723	0.577	0.571	0.582	0.879	0.881	0.884	0.884	0.665
DroidDream	J48	0.67	0.67	0.654	0.756	0.708	0.712	0.885	0.64	0.885	0.886	0.886	0.665
	LADTree	0.182	0.67	0.106	0.756	0.011	0.13	0.023	0.64	0.584	0.609	0.587	0.665
	NBTree	0.602	0.649	0.707	0.756	0.22	0.234	0.22	0.64	0.345	0.719	0.729	0.665
	RandomForest	0.664	0.672	0.676	0.756	0.773	0.77	0.775	0.64	0.885	0.884	0.877	0.665
	RandomTree	0.661	0.679	0.7	0.756	0.774	0.772	0.777	0.64	0.884	0.884	0.886	0.665
	RepTree	0.595	0.59	0.575	0.756	0.616	0.607	0.613	0.64	0.885	0.884	0.888	0.665
Opfake	J48	0.55	0.58	0.64	0.941	0.66	0.70	0.70	0.871	0.78	0.78	0.78	0.959
	LADTree	0.62	0.47	0.76	0.941	0.5	0.5	0.51	0.871	0.74	0.74	0.74	0.955
	NBTree	0.602	0.67	0.79	0.934	0.53	0.54	0.54	0.871	0.64	0.64	0.67	0.958
	RandomForest	0.64	0.78	0.80	0.941	0.68	0.68	0.74	0.871	0.84	0.84	0.84	0.955
	RandomTree	0.77	0.79	0.79	0.941	0.7	0.71	0.7	0.871	0.84	0.84	0.85	0.943
	RepTree	0.63	0.66	0.66	0.941	0.6	0.67	0.66	0.871	0.88	0.88	0.882	0.959
Ransomware	J48	0.724	0.801	0.824	0.948	0.766	0.704	0.72	0.896	0.785	0.785	0.785	0.979
	LADTree	0.781	0.754	0.736	0.961	0.545	0.655	0.654	0.879	0.724	0.743	0.754	0.962
	NBTree	0.72	0.69	0.699	0.954	0.602	0.589	0.702	0.89	0.76	0.76	0.79	0.972
	RandomForest	0.634	0.723	0.799	0.931	0.654	0.608	0.714	0.902	0.803	0.839	0.882	0.975
	RandomTree	0.77	0.79	0.796	0.918	0.712	0.711	0.743	0.935	0.812	0.801	0.801	0.967
	RepTree	0.639	0.643	0.646	0.942	0.612	0.627	0.637	0.872	0.799	0.808	0.817	0.932

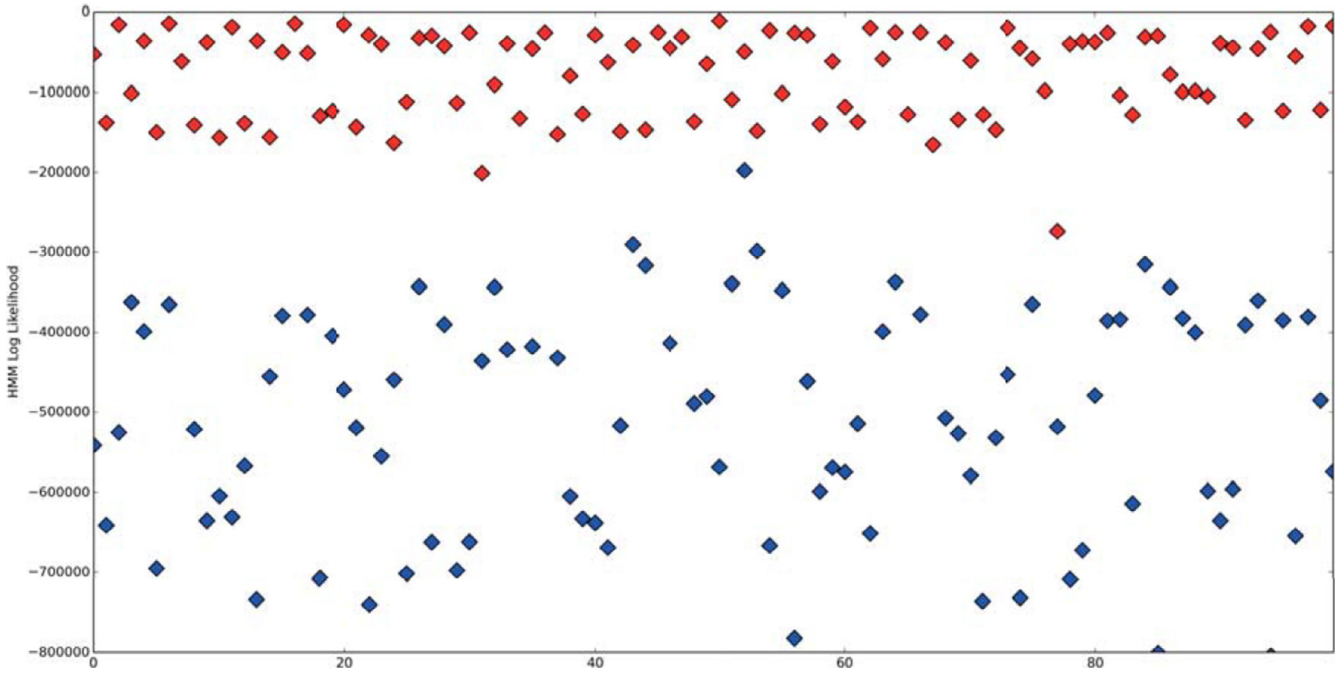

Fig. 3 – Comparison of malware (red) and trusted (blue) datasets classified with 3-HMM.

We, in fact, obtained poor performances in classifying the malware families by using the f_3 feature (HMM with 5 hidden states):

- a precision of 0.744 with RandomTree algorithm to recognize *FakeInstaller* family;
- a precision of 0.683 with RandomTree algorithm to recognize *Plankton* family;
- a precision of 0.776 with RandomForest algorithm to recognize *DroidKungFu* family;
- a precision of 0.777 with RandomForest algorithm to recognize *GinMaster* family;
- a precision of 0.786 with RandomForest algorithm to recognize *BaseBridge* family;
- a precision of 0.788 with RandomTree algorithm to recognize *Adrd* family;

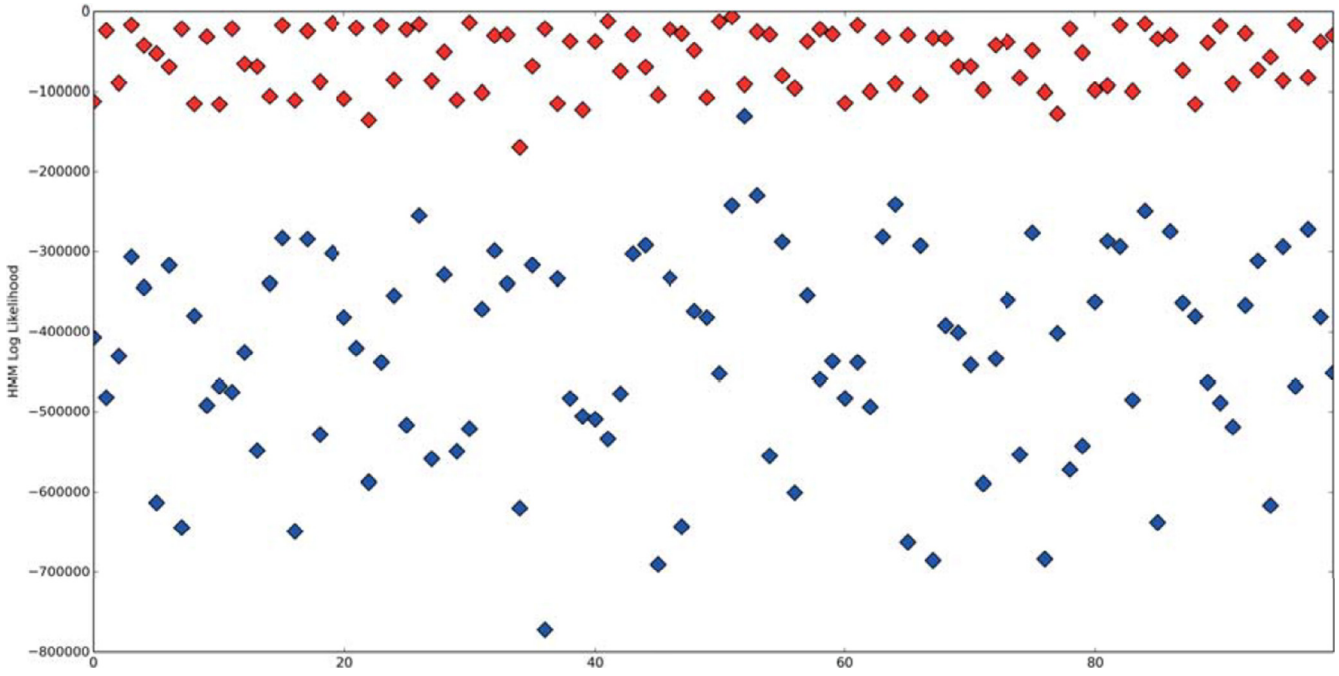

Fig. 4 – Comparison of malware (red) and trusted (blue) datasets classified with 4-HMM.

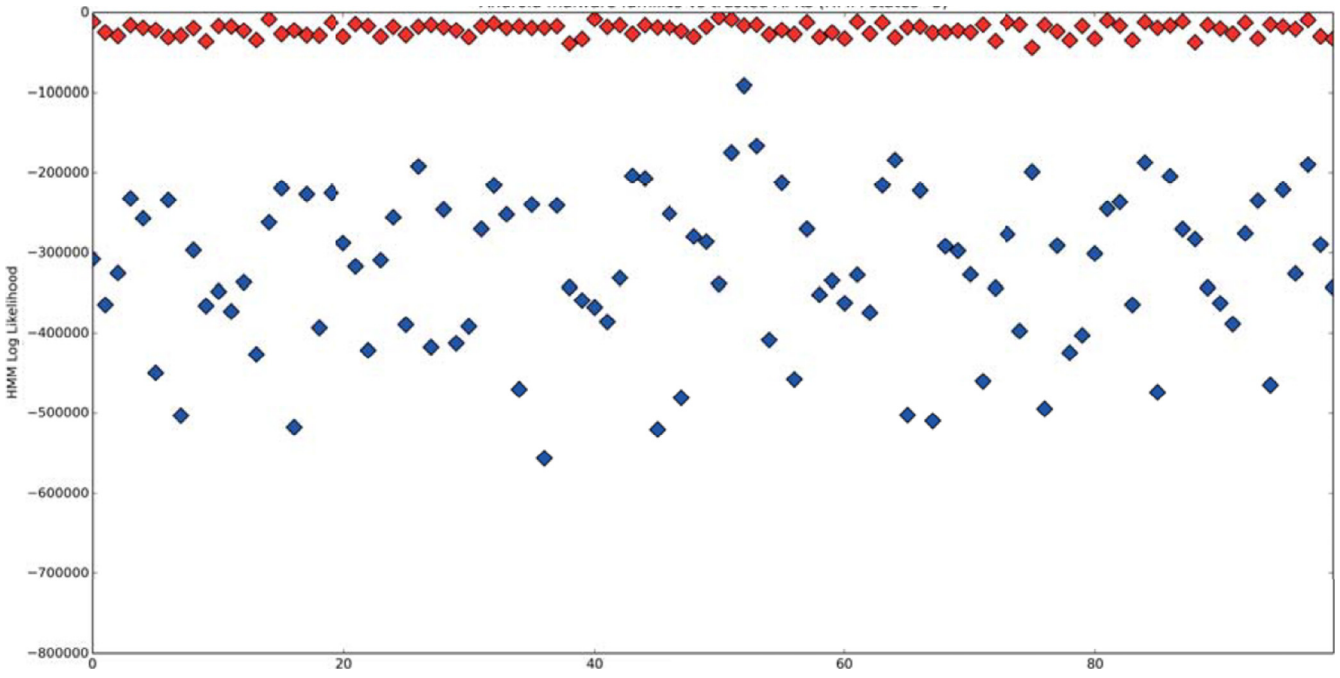

Fig. 5 – Comparison of malware (red) and trusted (blue) datasets classified with 5-HMM.

Table 6 – The performance evaluation of Structural Entropy method.

Contribute	Entropy
te_{score}	3.37891 s
te_{class}	0.47 s
te_{total}	3.84891 s

- a precision of 0.745 with RandomTree and RandomForest algorithms to recognize *KMin* family;
- a precision of 0.645 with J48 and RandomForest algorithms to recognize *Geinimi* family;
- a precision of 0.707 with NBTree algorithm to recognize *DroidDream* family;

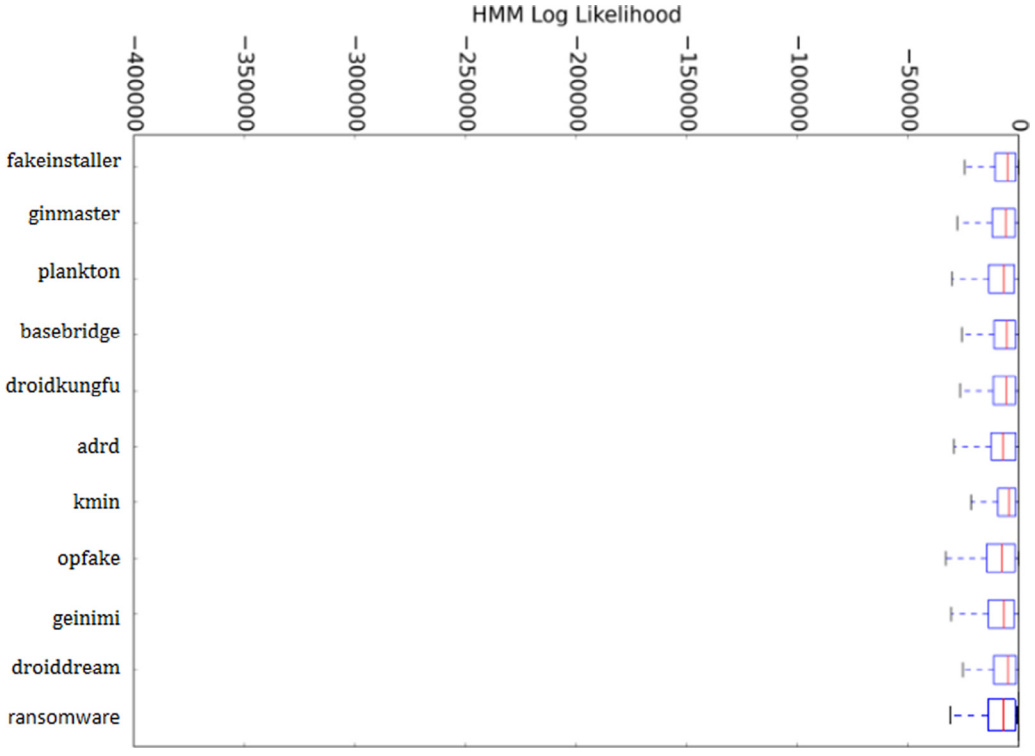

Fig. 6 – Boxplots of 3-HMM values for malware families.

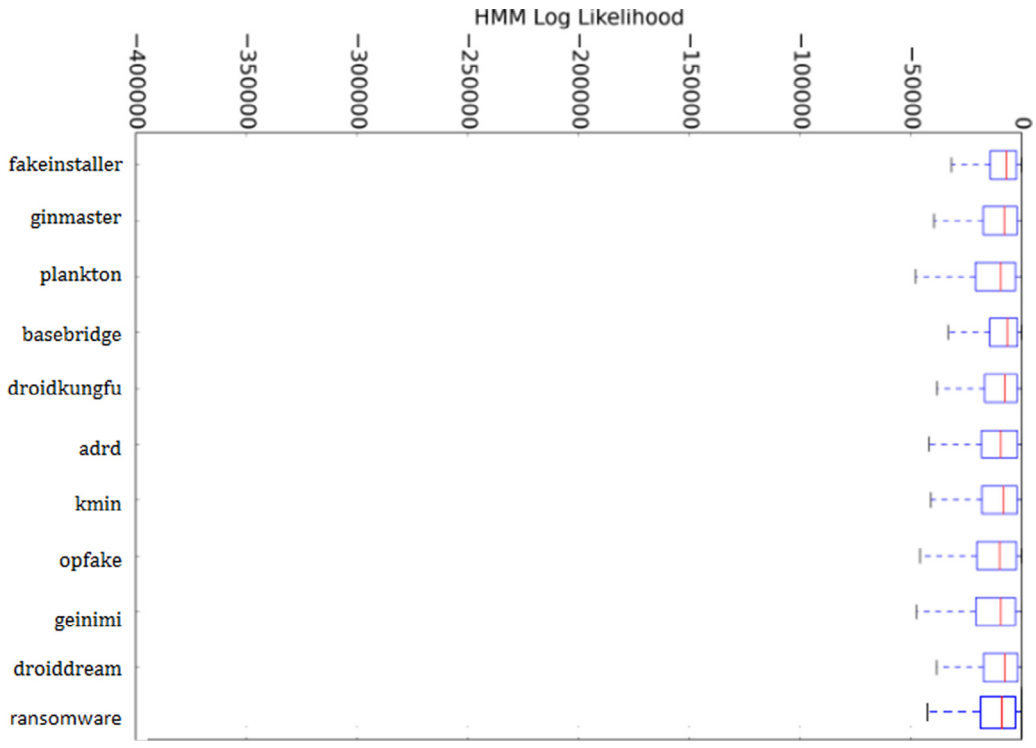

Fig. 7 – Boxplots of 4-HMM values for malware families.

- a precision of 0.8 with RandomForest algorithm to recognize *Opfake* family; and
- a precision of 0.824 with J48 algorithm to recognize *Ransomware* family.

RQ2 Summary: with regard to the classification analysis, we can conclude that similarly to RQ1, the f_3 feature (HMM with 5 hidden states) is the best one among the HMM-based features for classifying the malware families.

Fig. 8 – Boxplots of 5-HMM values for malware families.

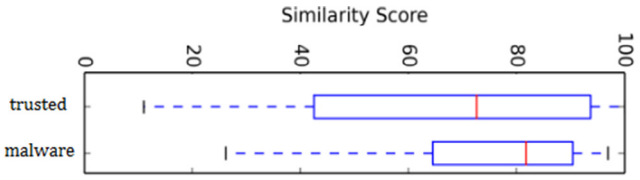

Fig. 9 – Structural entropy boxplots of malware and trusted dataset.

We obtain a range of precision varying from 0.645 to 0.8, which cannot be considered a good result as well.

We conclude that the HMM-based features present a fair precision value to classify mobile malware families.

5.3. *RQ3: is the structural entropy similarity able to discriminate a malware from a trusted application?*

Fig. 9 shows the box plots of the structural entropy values obtained from the malware and trusted dataset. The complete overlap of the malware box plot with a portion of the trusted box plot is a clear symptom that we should not expect a high precision value in the classification.

Classification analysis confirms our expectations: the f_4 (structural entropy) feature obtains a maximum precision value of 0.783 with the classification algorithm LADTree. The result is fair, but it is worse than the HMM-based features.

RQ3 Summary: the f_4 (structural entropy) feature cannot be considered a good indicator to discriminate a mobile malware application from a trusted one, as it presents a low precision value.

5.4. *RQ4: is the structural entropy similarity able to identify the family of a malware application?*

Fig. 10 shows the box plots regarding the structural entropy value of the eleven malware families analyzed.

Unlike HMMs, the comparison among the box plots of the entropy values of the different malware families shows a significant difference among families. This makes us to expect that the structural entropy could be effective to correctly identify the family a malware belongs to. This will emerge, as a matter of fact, with the classification.

We obtain the following values when classifying the malware families by using the f_4 (structural entropy) feature:

- a precision of 0.854 with all the six classification algorithms to recognize *FakeInstaller* family;
- a precision of 0.734 with all the six classification algorithms to recognize *Plankton* family;
- a precision of 0.918 with NBtree and RandomTree algorithms to recognize *DroidKungFu* family;
- a precision of 0.834 with all the six classification algorithms to recognize *GinMaster* family;
- a precision of 0.89 with J48 algorithm to recognize *BaseBridge* family;
- a precision of 0.875 with LADTree, NBTree, RandomForest, RandomTree and RepTree algorithms to recognize *Adrd* family;

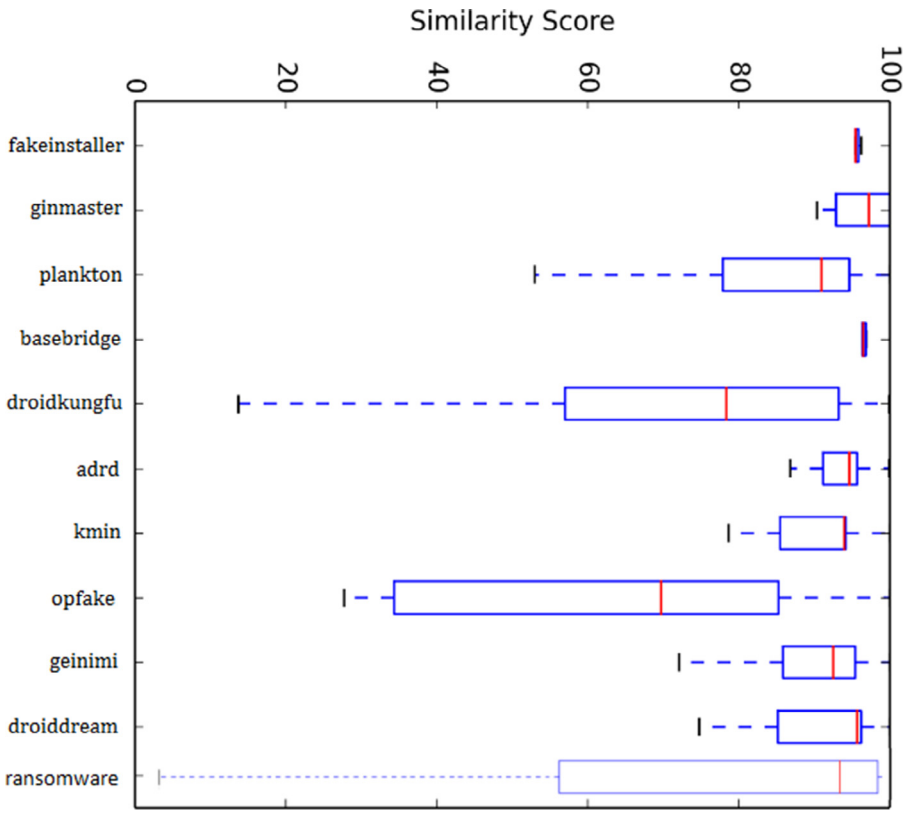

Fig. 10 – Structural entropy boxplots of malware families.

- a precision of 0.978 with NBTree algorithm to recognize *KMin* family;
- a precision of 0.725 with LADTree and RandomTree algorithms to recognize *Geinimi* family;
- a precision of 0.756 with all the six classification algorithms to recognize *DroidDream* family;
- a precision of 0.941 with all the six classification algorithms to recognize *Opfake* family; and
- a precision of 0.961 with LADTree classification algorithm to recognize *Ransomware* family.

RQ4 Summary: the f_4 (structural entropy) feature is the best in class to identify the malware family, with a precision from 0.725 to 0.978, respectively in case of *Geinimi* family and *KMin* family.

6. Performance evaluation

In this section we discuss the performances of the structural entropy and the HMM based detectors.

In order to measure performances of the two methods, we used the *time.clock()* Python function that returns the processor time. The processor time is the percentage of elapsed time that the processor spends to execute a non-idle thread, i.e. the cpu-time measured in seconds that the process requires to perform the computation.

The machine used to run the scripts and to take measurements was an Intel Core i5 desktop with 4 gigabyte RAM, equipped with Linux Mint 15.

We consider the overall time to analyze a sample as the sum of different contributions.

Regarding the structural entropy method we consider two different contributions to the overall time:

1. the average time to extract and compare the segments of two .dex files when computing the similarity score, i.e. te_{score};
2. the time to classify the similarity scores, i.e. the f_4 feature. We refer to this value with te_{class}.

Table 6 shows the cpu-time required to compute te_{score} and te_{class}. We notice that most cpu-time is used for the extraction and comparison. We compute the total time to evaluate a sample with the structural entropy method (te_{total}) as the sum of te_{score} and te_{class}.

Regarding the HMM method we distinguish the following contributions in order to compute the overall time:

1. the average time required to extract the sequence of instructions from each app of the training dataset, i.e. th_{seq};
2. the time required to learn the HMM with 3 hidden states (th_3), with 4 hidden states (th_4) and with 5 hidden states (th_5);
3. the average time required to evaluate the trained HMM, i.e. th_{eval}; and
4. the time required to classify the features extracted (i.e., f_1, f_2 and f_3 features). We refer to this value with th_{class}.

We compute the total cpu time required for HMM methods (th_{total}) as the sum of contributes th_{seq}, th_3 (when measuring HMM with 3-hidden states), th_4 (when measuring HMM with 4-hidden states), th_5 (when measuring HMM with 5-hidden states), and th_{class}.

Table 7 – The performance evaluation of HMM method when using 3-hidden states.

Contribute	HMM3
th_{seq}	3.5607 s
th_3	609.2059 s
th_{eval}	0.0291 s
th_{class}	0.47 s
th_{total}	613, 2657 s

Table 7 shows the HMM methods performance with 3 hidden states.

The most intensive task from the cpu point of view is represented by the time required to learn the HMM, while the evaluation phase requires 0.0291 seconds to test a new sample.

Table 8 shows the HMM methods performance with 5 hidden states.

The most intensive task for the cpu, when testing HMM with 4-hidden states, is represented by the time required to learn the HMM, while the evaluation phase requires 0.0339 seconds to test a new sample.

Table 9 shows the HMM methods performance with 5 hidden states.

Even with 5-hidden states, the most intensive task is represented by the time required to learn the HMM, while the evaluation phase require 0.0349 seconds to test a new sample.

Table 10 reports the comparison in terms of total cpu-time required to test a new sample when using structural entropy and HMM with 3, 4 and 5 hidden states.

We highlight that the less expensive method in terms of cpu time is represented by the structural entropy method that requires 3.84891 seconds in average to classify a new sample, while the HMM method requires more computational time than structural entropy, and when increasing the number of hidden states, the computational time also increases (from 613 seconds to learn an HMM with 3-hidden states to 1111 seconds to learn a HMM with 5-hidden states). The evaluation time for HMM

Table 8 – The performance evaluation of HMM method when using 4-hidden states.

Contribute	HMM4
th_{seq}	3.5607 s
th_3	691.1233 s
th_{eval}	0.0339 s
th_{class}	0.47 s
th_{total}	695, 1879 s

Table 9 – The performance evaluation of HMM method when using 5-hidden states.

Contribute	HMM5
th_{seq}	3.5607 s
th_3	1107.3799 s
th_{eval}	0.0349 s
th_{class}	0.47 s
th_{total}	1111.4455 s

Table 10 – Cpu time required in order to analyze a new sample using respectively structural entropy and HMM methods with 3, 4 and 5 hidden states.

Entropy	HMM3	HMM4	HMM5
3.84891 s	613.2657 s	695.1879 s	1111.4455 s

is quite similar (from 0.0291 seconds using 3-hidden states to 0.0349 seconds when using 5-hidden states).

Furthermore, we observe that the classification time is negligible in both methods; indeed it requires just 0.47 seconds for each method we measured.

7. Threats to validity

This section describes the threats that can affect the validity of our evaluation, known as: construct, internal reliability, and external validity.

7.1. *Construct validity*

Threats to construct validity may be related to imprecisions in measurements.

A construct validity factor in our study is represented by the restricted samples of our dataset; in order to mitigate this factor we used the 10 cross-validation, every classification is repeated for 10 times using testing set formed by different samples in order to evaluate every sample forming the full dataset.

7.2. *Internal validity*

Threats to internal validity regard the extent to which a causal conclusion based on a study is warranted.

Our results are strongly dependent by the machine learning algorithms used: to mitigate this factor we have used six different machine learning algorithms: J48, LADTree, NBTree, RandomForest, RandomTree and RepTree.

7.3. *Reliability validity*

Threats to reliability validity concern the capability of replicating this empirical study and obtaining the same results.

Scripts adopted to run the experiments are available on http://www.gerardocanfora.net/hmm-replication-package/HMM_Entropy.tar.gz. The malware dataset is publicly available for research purpose, at the following URL: http://user.informatik.uni-goettingen.de/darp/drebin/, a detailed instruction to obtain mobile malware samples, while ransomware samples are available at: http://ransom.mobi/. The developed scripts require the Python interpreter.

7.4. *External validity*

Threats concerning the generalization of results may induce the approach to exhibit different performances when applied to other contexts.

First of all, our study on mobile was previously applied to metamorphic malware for PCs, as we explained in related work section. Secondly, we have used a very large dataset (~11,000 applications), which could well represent the real population of malware and trusted applications.

8. Conclusion and future work

In this paper we propose a detector for malicious mobile applications consisting on a classifier which uses as features 3-4-5 states HMM and the structural entropy.

Current malware detection techniques are ineffective, as they usually fail against zero-day attacks, in addition to the fact that existing malware can easily evade the current detectors.

This happens because Android malware is increasingly becoming more and more complex, and it is acquiring characteristics that make it closer to polymorphic and metamorphic malware for PCs; or the Android malware itself uses techniques for morphing code.

The proposed methods has been already experimented on metamorphic viruses for PCs (Attaluri et al., 2008; Baysa et al., 2013): we studied the effectiveness of these methods to detecting Android malware.

Experimentation suggests that HMM method (with 5 hidden states) is the best one to identify malware applications with a precision of 0.96, while the structural entropy can more correctly identify the malware family, with a precision of 0.98.

These two methods could be implemented into a two-phase detector: as the first phase HMM method is applied to discriminate a malware application, while in the second phase structural entropy may identify the malware family.

The accuracy of the performed tests is very high in case of malware families recognition, but it is fair for the malware detection, which could be improved by using a larger number of states.

Considering the evolution of Android malware, the presented methods could be effective also in recognizing unknown malware, at least when it is generated by evolving existing malware.

Future works will be headed to assess the robustness of these two methods against morphing techniques on Android malware and to replicate the experimentation by increasing the number of states in the HMM detector.

REFERENCES

Addision PS. The illustrated wavelet transform handbook: introductory theory and applications in science, engineering, medicine and finance. Taylor & Francis Group; 2002.

Alcatel Lucent. Kindsight security labs malware report – q4. <http://www.tmcnet.com/tmc/whitepapers/documents/whitepapers/2014/9861-kindsight-security-labs-malware-report-q4-2013.pdf>; 2013 [accessed 21.01.15].

Andronio N, Zanero S, Maggi F. Heldroid: dissecting and detecting mobile ransomware. In: Research in attacks, intrusions, and defenses. Springer; 2015. p. 382–404.

Anon. An assembler/disassembler for android's dex format. <https://code.google.com/p/smali/>; 2015 [accessed 26.01.15].

Apvrille L, Apvrille A. Identifying unknown android malware with feature extractions and classification techniques. In: Trustcom/BigDataSE/ISPA, 2015 IEEE, vol. 1. IEEE; 2015. p. 182–9.
Arp D, Spreitzenbarth M, Huebner M, Gascon H, Rieck K. Drebin: efficient and explainable detection of android malware in your pocket. In: Annual network and distributed system security symposium (NDSS). 2014. p. 1–15.
Attaluri S, McGhee S, Stamp M. Profile Hidden Markov Models and metamorphic virus detection. J Comput Virol Hacking Tech 2008;5(2):179–92.
Bayer U, Kruegel C, Kirda E. TTanalyze: a tool for analyzing malware. In: European institute for computer antivirus research annual conference. 2006.
Baysa D, Low RM, Stamp M. Structural entropy and metamorphic malware. J Comput Virol Hacking Tech 2013;9(4):179–92.
Borda M. Fundamentals in information theory and coding. Springer; 2011.
Busticati Productions Presents. Dissecting the android bouncer. <https://jon.oberheide.org/files/summercon12-bouncer.pdf>; 2015 [accessed 30.01.15].
Canfora G, Mercaldo F, Visaggio CA. A classifier of malicious android applications. In: International conference on availability, reliability and security. 2013. p. 607–14.
Canfora G, Medvet E, Mercaldo F, Visaggio CA. Detecting android malware using sequences of system calls. In: Proceedings of the 3rd international workshop on software development lifecycle for mobile. ACM; 2015. p. 13–20.
Canfora G, Lorenzo AD, Medvet E, Mercaldo F, Visaggio CA. Effectiveness of Opcode ngrams for detection of multi family android malware. In: International conference on availability, reliability and security. 2015.
Chakradeo S, Reaves B, Traynor P, Enck W. MAST: triage for market-scale mobile malware analysis. In: ACM conference on Security and privacy in wireless and mobile networks (WiSec). 2013. p. 13–24.
Chen Y, Ghorbanzadeh M, Ma K, Clancy C, McGwier R. A Hidden Markov Model detection of malicious android applications at runtime. In: Wireless and optical communication conference (WOCC). 2014 23rd. IEEE; 2014. p. 1–6.
Chin E, Felt AP, Greenwood K, Wagner D. Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on mobile systems, applications, and services. ACM; 2011. p. 239–52.
Chouchane R, Stakhanova N, Walenstein A, Lakhotia A. Detecting machine-morphed malware variants via engine attribution. J Comput Virol Hacking Tech 2013;9(3):137–57.
Deshotels L, Notani V, Lakhotia A. Droidlegacy: automated familial classification of android malware. In: ACM SIGPLAN on program protection and reverse engineering workshop. 2014.
Dumitras T, Neamtiu I. Experimental challenges in cyber security: a story of provenance and lineage for malware. ACM; 2011.
F-Secure. Mobile threat report. <https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf>; 2015 [accessed 07.02.15].
Faruki P, Ganmoor V, Laxmi V, Gaur MS, Bharmal A. Androsimilar: robust statistical feature signature for android malware detection. In: International conference on security of information and networks. 2013. p. 151–9.
Feng Y, Anand S, Dillig I, Aiken A. Apposcopy: semantics-based detection of android malware through static analysis. In: ACM SIGSOFT international symposium on foundations of software engineering. 2014. p. 576–87.
Fraunhofer AISEC. On the effectiveness of malware protection on android. <http://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Studien_TechReports/deutsch/042013-Technical-Report-Android-Virus-Test.pdf>; 2013 [accessed 21.01.15].
InfoWorld. Update: McAfee: cyber criminals using android malware and ransomware the most. <http://www.infoworld.com/article/2614854/security/update-mcafee-cyber-criminals-using-android-malware-and-ransomware-the-most.html>; 2013 [accessed 21.01.15].
Karim ME, Walenstein A, Lakhotia A, Parida L. Malware phylogeny generation using permutations of code. Springer; 2005.
Khoo W, Lio P. Unity in diversity: phylogenetic-inspired techniques for reverse engineering and detection of malware families. In: SysSec workshop. Springer; 2011.
Ki Y, Kim E, Kim HK. A novel approach to detect malware based on API call sequence analysis. Int J Distrib Sens Netw 2015;2015:4.
Kinjo T, Funaki K. On hmm speech recognition based on complex speech analysis. In: Annual conference on industrial electronics. 2006. p. 3477–80.
Lagerspetz E, Truong HTT, Tarkome S, Asokan N. Mdoctor: a mobile malware prognosis application. In: International conference on distributed computing systems workshops. 2014. p. 201–6.
Liang S, Du X. Permission-combination-based scheme for android mobile malware detection. In: International conference on communications. 2014. p. 2301–6.
Liu X, Liu J. A two-layered permission-based android malware detection scheme. In: International conference on mobile cloud computing, service, and engineering. 2014. p. 142–8.
Lyda R, Hamrock J. Using entropy analysis to find encrypted and packed malware. Secur Priv 2007;5(2):40–5.
Ma J, Dunagan J, Wang HJ, Savage S, Voelker GM. Finding diversity in remote code injection exploits. In: Proceedings of the 6th ACM SIGCOMM conference on internet measurement. ACM; 2006.
Munoz A, Martin I, Guzman A, Hernandez JA. Android malware detection from Google play meta-data: selection of important features. In: 2015 IEEE Conference on Communications and Network Security (CNS). IEEE; 2015. p. 701–2.
Paller G. Dalvik opcodes. <http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html>; 2015 [accessed 26.01.15].
Plotz T, Fink GA. A new approach for hmm based protein sequence family modeling and its application to remote homology classification. In: Workshop on statistical signal processing. 2005. p. 1008–13.
Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G. Execute this! Analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS, vol. 14. 2014. p. 23–6.
Qin Z, Chen N, Zhang Q, Di Y. Mobile phone viruses detection based on hmm. In: International conference on multimedia information networking and security. 2011. p. 516–19.
Rabiner LR. A tutorial on Hidden Markov Models and selected applications in speech recognition. Proc IEEE 1989;77(2):257–86.
Ramachandran R, Oh T, Stackpole W. Android anti-virus analysis. In: Annual symposium on information assurance & secure knowledge management. 2012. p. 35–40.
Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating android anti-malware against transformation attacks. In: ACM symposium on information, computer and communications security. 2013. p. 329–34.
Sayfullina L, Eirola E, Komashinsky D, Palumbo P, Miche Y, Lendasse A, et al. Efficient detection of zero-day android malware using Normalized Bernoulli Naive Bayes. In: Trustcom/BigDataSE/ISPA, 2015 IEEE, vol. 1. IEEE; 2015. p. 198–205.

Sorokin I. Comparing files using structural entropy. J Comput Virol Hacking Tech 2011;7(4):259–65.
Spreitzenbarth M, Echtler F, Schreck T, Freling FC, Hoffmann J. Mobilesandbox: looking deeper into android applications. In: International ACM symposium on applied computing (SAC). 2013. p. 1808–15.
Ugarte-Pedrero X, Santos I, Sanz B, Laorden C, Bringas PG. Countering entropy measure attacks on packed software detection. In: The 9th annual IEEE consumer communications and networking conference – security and content protection. 2012. p. 164–8.
Visaggio CA, Mercaldo F. Evaluating the signature based and research antimalware tools against malware in the wild and third-party markets: a technical report. <https://www.researchgate.net/publication/275334543_Evaluating_the_commercial_and_research_antimalware_tools_against_malware_in_the_wild_and_third-party_markets_A_technical_report>; 2015 [accessed 17.06.15].
Wei J, Juarez E, Garrido MJ, Pescador F. Maximizing the user experience with energy-based fair sharing in battery limited mobile systems. IEEE TransConsum Electron 2013;59(3):690–8.
Wu W-C, Hung S-H. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In: Conference on research in adaptive and convergent systems. 2014. p. 247–52.
Xie L, Zhang X, Seifert J-P, Zhu S. PBMDS: a behavior-based malware detection system for cellphone devices. In: Proceedings of the third ACM conference on wireless network security. ACM; 2010. p. 37–48.
Xin K, Li G, Qin Z, Zhang Q. Malware detection in smartphones using Hidden Markov Model. In: International conference on multimedia information networking and security. 2012. p. 857–60.
Yerima SY, Sezer S, McWilliams G, Muttik I. A new android malware detection approach using Bayesian classification. In: International conference on advanced information networking and applications. 2013. p. 121–8.
Yerima SY, Sezer S, Muttik I. High accuracy android malware detection using ensemble learning, information security. IET 2015;9(6):313–20.
Zhou Y, Jiang X. Dissecting android malware: characterization and evolution. In: IEEE symposium on security and privacy (SP). 2012. p. 95–109.