HAL
open science

On the Use of Artificial Malicious Patterns for Android Malware Detection

Manel Jerbi, Zaineb Chelly Dagdia, Slim Bechikh, Mohamed Makhlouf, Lamjed Ben Said

To cite this version:

Manel Jerbi, Zaineb Chelly Dagdia, Slim Bechikh, Mohamed Makhlouf, Lamjed Ben Said. On the Use of Artificial Malicious Patterns for Android Malware Detection. Computers & Security, 2020, 92, pp.101743. 10.1016/j.cose.2020.101743 . hal-02464180

HAL Id: hal-02464180

https://inria.hal.science/hal-02464180v1

Submitted on 3 Feb 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Use of Artificial Malicious Patterns for Android Malware Detection

Manel Jerbi[1], Zaineb Chelly Dagdia[23], Slim Bechikh[1], Mohamed Makhlouf[4], and Lamjed Ben Said[1]

[1] SMART Lab, University of Tunis, ISG-Campus, Tunisia,
manel.jerbi@gmail.com, slim.bechikh@fsegn.rnu.tn, lamjed.bensaid@isg.rnu.tn
[2] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France.
[3] LARODEC, Institut Supérieur de Gestion de Tunis, Tunis, Tunisia,
zaineb.chelly-dagdia@inria.fr, chelly.zaineb@gmail.com
[4] Kedge Business School, Talence, France.
mohamed.makhlouf@kedgebs.com

Abstract. Malware programs currently represent the most serious threat to computer information systems. Despite the performed efforts of researchers in this field, detection tools still have limitations for one main reason. Actually, malware developers usually use obfuscation techniques consisting in a set of transformations that make the code and/or its execution difficult to analyze by hindering both manual and automated inspections. These techniques allow the malware to escape the detection tools, and hence to be seen as a benign program. To solve the obfuscation issue, many researchers have proposed to extract frequent Application Programming Interface (API) call sequences from previously encountered malware programs using pattern mining techniques and hence, build a base of fraudulent behaviors. Based on this process, it is worth mentioning that the performance of the detection process heavily depends on the base of examples of malware behaviors; also called malware patterns. In order to deal with this shortcoming, a dynamic detection method called Artificial Malware-based Detection (AMD) is proposed in this paper. AMD makes use of not only extracted malware patterns but also artificially generated ones. The artificial malware patterns are generated using an evolutionary (genetic) algorithm. The latter evolves a population of API call sequences with the aim to find new malware behaviors following a set of well-defined evolution rules. The artificial fraudulent behaviors are subsequently inserted into the base of examples in order to enrich it with unseen malware patterns. The main motivation behind the proposed AMD approach is to diversify the base of malware examples in order to maximize the detection rate. AMD has been tested on different Android malware data sets and compared against recent prominent works using commonly employed performance metrics. The performance analysis of the obtained results shows the merits of our AMD novel approach.

Keywords: Malware detection · API call sequences · Articial malicious patterns · Evolutionary algorithm · Android.

1 Introduction

The adoption rate of mobile devices continues to soar, with Android leading the way. This open-source operating system that is led by Google is now found on more than half of all smartphones. In fact, it was found that 85.2%, or 1.276 billion of smartphones that are shipped in 2017 were powered by Android[5]. This massive user base has caught the attention of cybercriminals who have begun to double down on their efforts to illegally obtain personal information from Android owners. For instance, it has been witnessed that many Android applications are found malwares defected in several app stores [58]. These facts and circumstances led to the necessity to call for efficient techniques capable of detecting malwares. In this concern, several malware detection methods were proposed in literature, and these can be categorized into three main heads; namely the static approaches, the dynamic approaches, and the hybrid approaches.

Static detection approaches are the most common ones used by antiviruses. Technically, these techniques examine the binary code of the targeted file, analyze all possible execution paths, and identify the malicious code without any execution. However, analyzing binary codes turns out to be difficult nowadays because modern compilers and runtime libraries have introduced significant complexities to these codes. This has negatively affected the capabilities of binary analysis toolkits to analyze binary codes, and as a consequent inaccurate analysis can be reported. Indeed, static analysis can be easily bypassed by various obfuscation techniques such as polymorphism, encryption or packing as these become more and more sophisticated. In addition, the fact that static analysis relies on a prebuilt signature database makes it hard to detect new unknown malwares until the signatures are updated. Besides, some execution paths can be only explored after execution; a task that static analysis cannot perform.

To cover the mentioned deficiencies, dynamic approaches were introduced in [54] [41]. In contrast to static approaches, dynamic approaches consider the behavior of malware during runtime. Such approaches execute the malicious file and trace its behavior. More precisely, these approaches detect malwares by analyzing the similarity between the behavior of the new and the known ones based on Application Programming Interface (API) calls. Those approaches act almost actively against polymorphic malwares because obfuscation techniques only change the static signature of these and do not affect their behavior. Despite of the advantages of these techniques, they suffer from a greedy preprocessing phase during runtime and monitoring which lowers the system's performance.

Trying to come up with a better class for malware detection techniques, hybrid approaches were proposed in [46]. Those approaches mainly combine both static and dynamic approaches and work on two phases where they use both dynamic and static features.

This paper mainly focuses on dynamic approaches for malware detection. Accordingly, in this work, a novel dynamic analysis approach for malware detection—

[5] `http://www.zdnet.fr/actualites/chiffres-cles-les-os-pour-smartphones-39790245.htm`

called Artificial Malware-based Detection (AMD)— is proposed. AMD is specifically developed to cope with obfuscation techniques. In this study, malware is analyzed based on frequency analysis of API call sequences where this information is collected dynamically. Then, a malicious pattern set and a benign pattern set are built both relying on vectors extracted from the collected data. The idea behind the use of frequent API call sequences is that it represents an inevitable characteristic of malicious computer programs; especially obfuscated ones. Technically, the high frequency of API call sequences can be the result of both conditional iterations and recursive subroutines used by a programmer. Additionally, repetitive actions on data sequences are used by malware writers, especially well-known loops performing infection, encryption, and decryption. Knowing that the frequent API call sequences have a greater potential for having useful information and semantics about computer programs, in comparison to other sequence based substructures, this paper includes experiments on how they can help to detect obfuscated malwares.

In this research, the proposed AMD approach takes the patterns (i.e., API call sequences) found in an executable file to create two behavioral models. The first model reflects the set of malicious applications while the second represents the benign applications. These patterns that consist of groups of frequent API call sequences that use to appear together in the same applications are compared against the list of patterns of a new application, where the most similar behavior will decide its nature. To overcome the fact that the performance of the detection process will heavily depend on the base of examples of malware behaviors, a hybrid detection method that makes use of not only extracted malware patterns but also *artificially generated ones* is proposed. The artificial malware patterns are generated using an evolutionary Genetic Algorithm (GA) [13] which evolves a population of API call sequences. The main idea behind this is to find new malicious behaviors in order to insert them into the base of examples. This will enrich the base of examples with unseen malware patterns. The main motivation behind the proposed AMD approach is to diversify the base of malware examples, and subsequently improve the detection rate. The main contributions of the current research work are the following:

1. The evolutionary generation of artificial malicious patterns, using a GA, and their use in combination with extracted existing patterns for malware detection. This is to guarantee a fairly varied database of malicious behaviours.
2. The demonstration of the benefits of using artificial malicious patterns in maximizing accuracy and minimizing false alarms.
3. The illustration of the outperformance of the proposed AMD approach when compared against several state-of-the-art detection methods.

The rest of this paper is organized as follows: Section 2 presents a detailed description of the related work. Section 3 describes the proposed Artificial Malware-based Detection (AMD) approach. The experimental setup is introduced in Section 4. The results of the performance analysis are given in Section 5, and the conclusion is given in Section 6.

2 Related Work on Evolutionary Malware Detection

In this section, some background information is provided about the different types of evolutionary malware detection techniques. Two major approaches of malware detection using evolutionary approaches are revised; namely the evolutionary approaches for malware classification and the evolutionary approaches for pattern generation. Malware classification is the process of classifying malware into different "known" classes. The main characteristics of these techniques are their efficiencies in terms of classifying malware into different classes. However, due to the existence of various intrusion detection techniques such as code packing, anti-debugging, control-flow and entry point obfuscation, these approaches can be in some cases inefficient [28]. These limitations led to introduce a system that tries to deal with unknown malware classes. Compared to the classification techniques, malware generation techniques try to come up with new variants of malware and this can immunize the analysis process from many obfuscation techniques and even from self-modifying programs. It can also be reported that some other previous works revise three major approaches of malware detection which are proposed in literature; namely the static, the dynamic and the hybrid analysis techniques. These techniques are discussed in Appendix A.

2.1 Evolutionary approaches for malware classification

Malware classification approaches can be categorized into two heads. The first category refers to the non-evolutionary based classification approaches, where works such as [31], [32], [42], [43], [56] can be mentioned, while the second category refers to the branch of the evolutionary based classification approaches like in [1]. In this paper, our main focus relies on the evolutionary based approaches. Among the methods proposed within the second category, eXtended Classifier System (XCS) [55] can be cited and which is a Michigan-style classifier that derives a set of rules based on accuracy. In the XCS's training phase, the input data is used to generate the initial rules. Afterwards, these initial rules are evolved into new rules using a niche genetic algorithm reserved in the population. The fitness of the individual (or rule) is determined based on the accuracy of each rule. The predicted payoff values for all actions are saved in a prediction array. Based on the prediction accuracy of the rules, some of them are also deleted from the population. The evolution of accurate generalizations and accommodation of real values makes XCS suitable to solve various real-world problems. The sUpervised Classifier System (UCS) [6] is another Michigan style classifier and is based on accuracy. UCS inherits the same principle and structure of XCS where an initial population of rules is derived from training data. However, it differs from XCS in the following aspects: (1) it is based on a supervised learning scheme that computes fitness instead of reinforcement, (2) the GA is applied to correct the rule-set for updating its population and (3) it does not use a prediction array. The on-line learning and evolving abilities of UCS make it useful in efficiently solving different real-world problems. In another work,

Bacardit et al. [5] developed a Genetic clASSIfier SysTem (GAssist) which is a Pittsburgh style classifier based on the approach where each individual in the population depicts a complete solution to the classification problem. It uses a GA to evolve the rule sets of the population and exploits a fitness function to get a good balance between complexity and accuracy of the generated rules. GAssist also includes incremental learning with the alternating strata (ILAS) windowing approach, which is used to improve the generalization of the rule sets, as well as the adaptive discretization intervals (ADI) rule representation. In the same category, the work of Gonzblez et al. [19] can be mentioned; which is named Structural Learning Algorithm in Vague Environment (SLAVE). SLAVE is a genetic learning algorithm based on the use of fuzzy logic concepts and the genetic iterative approach. The algorithm elects and gains knowledge via a single fuzzy rule in each iteration. This results in the pruning of the search space for potential solutions. After that, a class is fixed and the best antecedent for this class is selected. Finally, a set of repeated iterations generates a full rule-set for the classification of instances. Here, the fitness of rules is computed by using completeness and consistency of the rules. The algorithm has been effectively used in several real-world applications and shows promising results [38].

2.2 Evolutionary approaches for pattern generation

Different works in literature considered using extracted static features to generate malware. By doing so, several researcher works try to come up with a new idea to detect malware without being tied to a static base of malware signatures. In this context, Aydogan et al. [4] evolved new malwares, specifically variants of known malwares, by using genetic programming (GP) in order to evaluate the performance of existing static analysis tools. Also, Zolkipli et al. [59] proposed a technique that combines a signature-based technique and the evolutionary genetic algorithm. The developed technique has three main modules which are: signature-based detection, genetic algorithm-based detection and signature generator. The signature generator uses signatures which are defined as the string patterns which are unique to identify and characterize the malware. In another work proposed by Edge et al. [15], an artificial immune system genetic algorithm (REALGO) was developed based on the human immune system's use of reverse transcription ribonucleic acid (RNA). The REALGO algorithm combines known information from past viruses with a type of prediction for future viruses. The authors generate antibodies (new virus signatures) from antigens (string of known virus signatures). Also, Noreen et al. [33] proposed a framework based on the notion of evolution in viruses on a well-known virus family, called "Bagle". In [33], features are extracted based on the assembly code and evolved using GAs. The generated virus files are afterwards tested using a commercial antivirus. Kayacık et al. [26] proposed a static anomaly based approach to detect malware. This work focuses on the vulnerability testing of host-based anomaly detectors by generating evasion attacks. In a typical evasion attack, the attacker aims to alter a generic attack template —the core of an attack— so that the evasion attack mimics normal behavior to evade detection. Authors in [26] mainly focused

on generating malware, specifically buffer overflow attacks, and not on detecting them. Already developed detectors were used to evaluate the generated attacks.

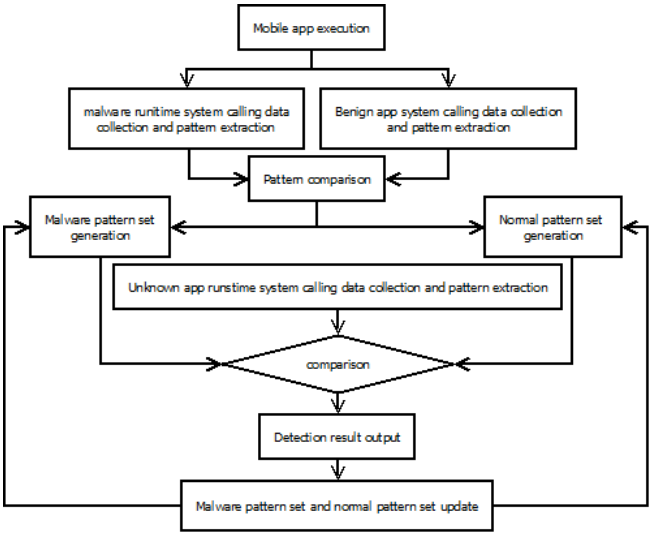
3 Proposed Approach: Artificial Malware-based Detection

In this section, the proposed Artificial Malware-based Detection (AMD) solution is presented. First, a general description of the proposed model is given, followed by a detailed description of the approach by describing its different phases.

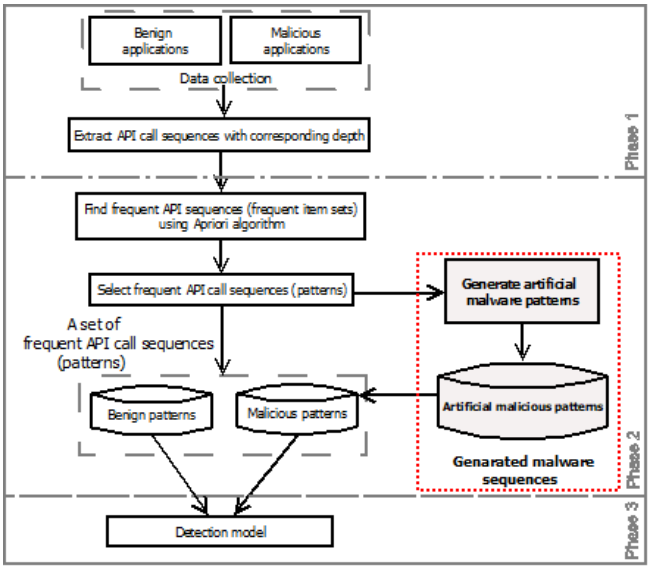
3.1 Overview

Previous works in the academic security research community have studied the malware detection problem extensively. The proposed methods mainly achieve high reported detection performance on predefined data sets. Some of them also report reasonably fast prediction times. However, most of them are less suitable for real-world deployment because requirements for malware detection need to be independent from the provided base of examples of malware behaviors. Several important requirements for deploying malware detection systems in the real-world need to be taken care of. One such requirement is that deployed approaches should be tested against a stream of continuously evolving data. Several state-of-the-art techniques have proposed to extract frequent API call sequences from encountered malware programs using pattern mining techniques. These sequences build a base of *fraudulent behaviors*. Afterwards, API call sequences can be extracted from any program and based on these, the considered program behavior can be judged to be more similar to malware behaviors or benign-ware ones. Figure 1a shows the main steps followed by [47] which is a recent representative work of the state-of-the-art malware detection methods. These steps are also almost the same steps used in previous works in literature which are based on the use of API call sequences to detect malware. The main remark that can be extracted from such detection techniques is that the efficiency of malware detection is *very dependent on the base of examples* of malware behaviors. More precisely, the provided or collected base of examples is static and therefore lacks diversity, which may negatively impact the detection rate.

To deal with this issue, in this paper, an evolutionary based solution—called *Artificial Malware-based Detection (AMD)*— is proposed and that is capable to overcome the problem of lack of diversity where the detection ability becomes less dependent on the base of examples of malware behaviors. Differently to the works presented in [4], [15], [26], [33], AMD does not evolve static signatures of malware. It diversifies the base of examples in an automatic way and this is achieved via the development of an automatic generation technique of malicious patterns; using a Genetic Algorithm (GA). This presents our main contribution in comparison to the state-of-the-art methods. The generated patterns will be injected into the collected malware base. More precisely, after extracting the



(a) Tong et al.'s detection procedure structure [47]



(b) AMD worflow structure

Fig. 1: AMD's added value compared to Tong et al.'s approach [47] (The key component of our AMD approach is highlighted with the red rectangle)

frequent malicious sequences of API calls from the database of collected examples of malware behaviors, these will be fed to a GA that will reproduce a final population of optimized malicious patterns quite similar but different from the provided sequences. Figure 1b shows the main steps of the proposed AMD ap-

proach which allows a comparison to the representative work given in [47]. In Figure 1b, the dotted red rectangle highlights the key part (in phase 2) reflecting the main contribution of this research paper. As presented in Figure 1b AMD is based on three main phases namely: (1) API call sequences extraction, (2) patterns construction and (3) a detection phase where unknown new apps are classified. The first phase (Section 3.2) is responsible mainly for extracting the API call sequences with their corresponding depths from the collection of normal and malicious applications to transmit them afterwards to the next phase. This phase requires a preprocessing of the collected data (Android apps) which is detailed in Appendix B. Through the second phase, the process of the patterns construction is subdivided into three main sub-steps namely the extraction of the frequent API call sequences, the generation of the artificial patterns and the patterns gathering process. In the first sub-step (Section 3.3), the frequent sets of API call sequences, referred to as frequent item sets (also called patterns), are extracted with their corresponding depths using the Apriori algorithm [2], which is one of the most used algorithms for pattern mining. Among these, a selection is performed to keep a set of the unique patterns, i.e., all the common patterns between the benign and the malicious are removed. Meanwhile, in the second sub-step (Section 3.4), a database of artificially generated malware patterns is created using the set of the selected patterns. This is achieved via the use of a GA, aiming to diversify the base of malware examples with artificial malicious patterns in order to maximize the detection rate. The third sub-step (Section 3.5) will mainly gather the results of the two previous sub-steps and thus, has two outputs: in one hand it generates a collection of benign patterns, and on the other hand it offers an enriched collection of malicious patterns, i.e., the selected malware patterns and the artificially generated ones. The third and the last phase (Section 3.6) invokes the detection model. Throughout this phase and by using the collection of the generated patterns, malicious programs will be detected among the new apps.

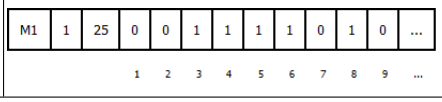
3.2 Extraction of API call sequences

In this section, the followed steps to extract API call sequences from the collection of normal and malicious applications are described. The process is also detailed via the use of a descriptive example.

From a given data collection comprising malware and benign executable samples, two data sources are filled. The first data source includes the samples of benign sequences while the second includes the set of malware sequences. Each data set is in the form of a set of API call sequences described with their identifiers (IDs) followed by their class labels indicating their nature, i.e., either malicious or benign, then their different calling depths and finally a set of binary values indicating if an API call is current or not in the API call sequence. Such sequence format is named an "item vector" which is related to the extracted API call sequences. A succession of these API calls induces an API call sequence characterized by a specific depth referring to the number of API calls within the sequence. To model the API call, the format of LIBSVM [12] is used

generating an identifier (ID) for each API call and its according occurrence in the API call sequence. Afterward, each API call sequence is assigned a specific ID. An example on how to extract an API call sequence generating an item vector is given in Table 1.

Table 1: Process of API call sequence extraction (Inspired by [12])

API call sequence (depth = 25)	generateKey,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass, getDisplayMessageBody,getInstance, getInstance,getInstance,getInstance,close, close,close,close,close,doFinal,doFinal, exit,doFinal
LIBSVM format: Number representation (IDs) Related occurrences	 4 6 6 6 6 6 6 6 6 6 6 5 8 8 8 8 16 16 16 16 3 3 21 3 3:3 4:1 5:1 6:10 8:4 16:4 21:14
Class label	Label: 1 (M: Malware)
Output of phase 1: Extracted API call sequence (item vector) of ID: M1	

Based on its nature, each of these extracted item vectors is stored in its related data set; either to a "database of malicious sequences" (DBM) or to a "database of benign sequences" (DBB).

3.3 Frequent item vectors extraction

In this section, the problem formulation required to apply the Apriori algorithm is given first. Next, the extraction of the frequent API call sequences followed by the selection step are detailed.

Problem formulation using the Apriori Algorithm As previously mentioned, to extract the sets of frequent item vectors (also referred to as patterns) the Apriori algorithm [2] is used. However, the application of the algorithm is not straightforward as some adjustments are needed. Hence, the adopted formalization is as follows:

In the Apriori algorithm, an item set is defined as a set of items which corresponds in the AMD context to the set of the item vectors. Let us recall that every executable is seen as a set of item vectors. Let $B = \{i_1, \ldots, i_m\}$ be a set of items; where m is the number of item vectors. B is called the item base

referring to the set of all the item vectors in all the executables. Any subset $I \subseteq B$ is called an item set. An item set may be any set of item vectors that can appear in the same executable. Let $T = \{t_1, \ldots, t_m\}$ where $\forall k$, $1 \leqslant k \leqslant m$: $t_k \subseteq B$. T is a tuple of transactions over B. This tuple is called the transaction database. The latter can list the sets of item vectors that appear in an executable. Every transaction is an item set. The item base B may not be given explicitly, but only implicitly as:

$$B = \bigcup_{k=1}^{n} t_k \tag{1}$$

Let $I \subseteq B$ be an item set and T a transaction database over B. In such a case, the transaction $t \in T$ covers the item set I, i.e., the item set I is contained in a transaction $t \in T$ if and only if $I \subseteq t$. I is characterized by its support $S_T(I)$ which is defined as the number (or fraction) of transactions that encloses it and is defined as:

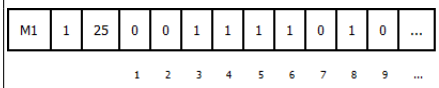
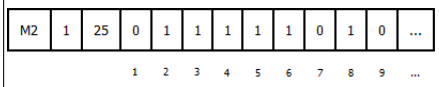
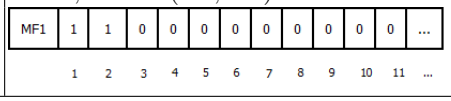
$$S_T(I) = |\{t \in T; I \in t\}|/|T| \tag{2}$$

In this problem formulation, it is required to specify two threshold values, $S_{min} = \{S^b_{min}, S^m_{min}\}$ one for the benign item sets and another one for the malicious item sets. These threshold values will help to judge if an item set is frequent or not. In fact, if $S_T(I) \geq S_{min}$ then the item set I will be considered as frequent. The set of all the obtained frequent item sets is defined as the set $FT(S_{min}) = I \subseteq B | S_T(I) \geq S_{min}$.

The extraction process In this step, the frequent item sets are identified, i.e., the frequent sets of item vectors. Frequent item sets are those that have a support value greater than a threshold value S_{min}. Given a set of item sets that are labeled either as malicious or as benign, frequent item sets are extracted based on the Apriori algorithm formulization. The output is hence two data sets of frequent item sets: a data set containing only malicious frequent sets of item sets (DBMFIV) and a data set containing only benign frequent sets of item sets (DBBFIV). Formally, these generated databases can be defined as follows:

Given a data set DBMFIV or DBBFIV of item sets, sets of item vectors, the support of an item set is the number of occurrence of the item set in the whole set of item sets. An item set is frequent if its support is not less than a given support threshold S_{min}. The generated frequent item set is represented in the form of a set of an identifier followed by a set of binary values where each value takes 1 if the item set occurs in the frequent item set representation otherwise 0. In Table 2, an example of the frequent item set extraction process is given.

Table 2: Example of a frequent item set extraction

API call sequence 1 process	
API call sequence 1 (depth = 25)	generateKey,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass, getDisplayMessageBody,getInstance, getInstance,getInstance,getInstance,close, close,close,close,close,doFinal,doFinal, exit,doFinal
LIBSVM format: Number representation 1 (IDs) Related occurrences	 4 6 6 6 6 6 6 6 6 6 6 5 8 8 8 8 16 16 16 16 3 3 21 3 3:3 4:1 5:1 6:10 8:4 16:4 21:1
Class label	Label: 1(M : Malware)
Item vector 1	
API call sequence 2 process	
API call sequence 2 (depth = 25)	generateKey,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass, startRecording,getDisplayMessageBody, getInstance,getInstance,getInstance, getInstance,close,lose,close, cclose,stop,stop,exit,doFinal
LIBSVM format: Number representation 2 (IDs) Related occurrences	 4 6 6 6 6 6 6 6 6 6 6 7 2 16 16 16 16 19 19 21 3 2:1 3:1 4:1 6:10 7:1 16:4 19:2 21:1
Class label	Label: 1 (M : Malware)
Item vector 2	
Frequent item set generation	
Frequent item set of ID: MF1 Output format:	Let us assume that $S_T(M1, M2) \geq Smin^m$. Hence, the set (M1, M2) is considered as frequent. 

Frequent item sets selection In this step, the frequent item sets that are more discriminative between both of the malicious frequent item sets and the benign ones are selected. To do so, only the frequent item sets that are not

common between the malicious and the benign item sets are kept as indicated in Equation 3 and Equation 4:

$$MPDB = DBMFIV - DBMFIV \cap DBBFIV \tag{3}$$

$$BPDB = DBBFIV - DBMFIV \cap DBBFIV \tag{4}$$

More precisely, the filtered malicious frequent item sets (malicious patterns) will be stored in the Malicious Patterns Database (MPDB) and the Benign Patterns Database (BPDB) will store the filtered benign patterns.

3.4 Artificial malicious patterns generation using GA

Let us recall that the main originality of this research is to generate a set of artificial malicious patterns in order to maintain a fairly varied and renewed database of malicious patterns. To detail this process, first of all, a general overview highlighting the process of artificially generating the malicious patterns is given. Then, a detailed description of the pattern's encoding schema using the genetic algorithm is presented.

General overview To generate an artificial set of malicious patterns, a genetic algorithm is used. More precisely, the process of artificially generating the malicious patterns goes through three main steps. These steps are given in Algorithm 1. In the first step (Algorithm 1, line 1), a base of patterns is produced with compositional characteristics similar to those of the real patterns stored in the MPDB that comprises the malicious patterns. In the second step (Algorithm 1, line 2), each generated pattern is evaluated according to a fitness function using the set of the benign patterns stored in BPDB; this is to only keep the best fitting patterns. The third step described via its sub-steps (Algorithm 1, lines 1-4) consists in replacing the initial set of patterns (i.e., the first generation of patterns from line 1) with those selected as best fitting ones. The third step will be repeated until a stopping criterion is reached. At this level, and as a result of all the previous steps, a set of malicious patterns, called "artificial" set of malicious patterns, is generated. These "artificial" malicious patterns best mimic the behavior of the "real" malicious ones. Once the artificial set of malicious patterns is generated, it will be stored in its related Artificial set of Malicious Patterns DataBase (AMDB).

Pattern encoding using GA A genetic algorithm is a probabilistic search algorithm that iteratively transforms a population of objects (a set of chromosomes), each with an associated fitness value, into a new population of offspring objects using operations such as crossover and mutation. In AMD, GA begins with a set of suitable solutions which refers to the set of selected malicious patterns; namely MPDB. Each solution will be represented by a chromosome-like data structure. Solutions from one population are selected and used to generate a new population. This is motivated by the possibility that the new population

Algorithm 1: Generation of artificial patterns

Input: $MPDB$: set of malicious patterns, $BPDB$: set of benign patterns
Output: Set of artificially generated malicious patterns $AMDB$
1: Generate the initial population of individuals randomly from $MPDB$ (First generation)
2: Evaluate the fitness of each individual in that population using the sets in $BPDB$.
3: Repeat the following generational steps until the termination condition has been reached:
3.1: Select the parent individuals for reproduction.
3.2: Breed parent individuals through crossover and mutation operations to give birth to offspring individuals.
3.3: Evaluate fitness of each offspring individual.
3.4: Execute replacement by a competition between parent individuals and offspring ones.

will be better than the old one. Solutions are selected according to their fitness to generate a new population; more suitable they are more chances they have to reproduce. This is repeated until a specific condition is satisfied, i.e., the fixed number of generations is reached. To achieve the patterns generation task, three factors will have vital impact on the effectiveness of the used genetic algorithm; these are the following: (1) the encoding of individuals, (2) the fitness function and (3) the GA parameters.

The first factor to consider is how to encode the potential solutions to AMD in a form which can be processed by the GA. Each solution may be represented in the form of a chromosome. The different positions in a chromosome, referred to as genes, are changed randomly within a range during the process of evolution. Solutions are encoded as identifier elements as $\{M1, \ldots, MX\}$ where X represents the total number of extracted item vectors. In fact, each chromosome is a sequence within which all the genes are encoded via fixed length item vectors. Let us recall that, each item vector is assigned a specific ID followed by its class label indicating its nature, i.e., either malicious or benign, then its calling depth (length) and finally a set of binary values indicating if an API call is current or not in the vector. A gene and a chromosome may look as shown in Figure 2.

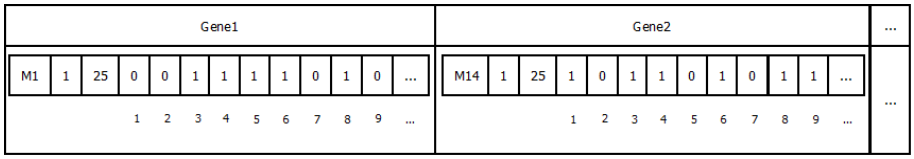


Fig. 2: The GA chromosome representation: A chromosome is a sequence of genes each encoding an item vector corresponding to a particular behavior.

Algorithmic details

Evaluation of the fitness function Each artificially generated pattern is derived from the initial population, i.e., the set of malicious patterns MPDB. Using the fitness function $f_{Qual}()$, defined in Equation 5, each generated malicious pattern is evaluated using the sets of benign patterns in BPDB. The latter is required to ensure that the generated malicious patterns are different from the benign ones in BPDB. Afterwards, those generated malicious patterns have to be inserted into the base of the generated patterns AMDB. The used fitness function guarantees the diversity of the generated population by avoiding the insertion of similar generated malware examples and by falling into an homogeneous population.

$$f_{Qual}(GS_i) = \frac{Sim(MS, GS_i) + Sim(BS, GS_i) + Overlap(GS_i)}{3} \tag{5}$$

where $i \in [1, n]$; n indicates the total number of artificially generated patterns.

Based on $f_{Qual}()$, the quality of a solution which refers to an artificially generated pattern (GS_i) is evaluated using the following three criteria:

1. $Sim(MS, GS_i)$ refers to the similarity between the generated pattern GS_i and the malicious patterns (MS). This measure of similarity needs to be maximized.

$$Sim(MS, GS_i) = \frac{\sum MS_{j \in MS} Sim(GS_i, MS_j)}{|MS|} \tag{6}$$

   where $j \in [1, m]$; m indicates the total number of malicious patterns.
2. $Sim(BS, GS_i)$ refers to the similarity between the generated pattern GS_i and the benign patterns (BS) which has to be the lowest.

$$Sim(BS, GS_i) = \frac{\sum BS_{k \in BS} Sim(GS_i, BS_k)}{|BS|} \tag{7}$$

   where $k \in [1, p]$; p indicates the total number of benign patterns.
3. Overlap(GS_i) is measured as the average value of the individual $Sim(GS_i, GS_l)$ between the generated pattern GS_i and all the other generated patterns GS_l in the generated data set AMDB. l refers to the total number of the generated artificial patterns.

$$Overlap(GS_i) = 1 - \frac{\sum GS_{l, i \neq l} Sim(GS_i, GS_l)}{|GS|} \tag{8}$$

   To calculate the similarity $Sim()$ between two patterns, the Needleman-Wunsch [30] alignment algorithm in adapted to the AMD context. Formally, this algorithm creates a matrix of scores $S_{i,j}$ as presented in Equation 9. When aligning two patterns, called sequences, $(a_1; \ldots; a_n)$ with n representing the total number of items in the sequence A and $(b_1; \ldots; b_m)$ with m representing the total number of items in the sequence B, each position $S_{i,j}$

in the matrix corresponds to the best score of alignment considering the previously aligned elements of the sequences. Meanwhile, the algorithm can introduce gaps represented by "–" to improve the matching of subsequences. In the AMD case, gaps can only appear at the end of a sequence when aligning two sequences with different depths.

$$S_{i,j} = MAX \begin{cases} S_{i-1,j} & + g \text{ insert gap for } b_j \\ S_{i,j-1} & + g \text{ insert gap for } a_i \\ S_{i-1,j-1} & + Sim_{i,j} \text{ match} \end{cases} \tag{9}$$

where $S_{i,0} = g * i$ and $S_{0,j} = g * j$ when aligning two sequences $(a_1; \ldots; a_n)$ and $(b_1; \ldots; b_m)$. At any given point, the algorithm considers two possibilities. First, it considers the case when a gap should be inserted or not. When the algorithm decides that a gap needs to be inserted for either a or b, it applies a penalty of g. Second, the algorithm tries to match sequences. The similarity function $S_{i,j}$ returns the reward or cost of matching a_i with b_j. The final similarity is contained in $S_{n,m}$. The adaptation of the algorithm is straightforward: The gap penalty g and the similarity function to match patterns $Sim()$ are defined, and perfect matches in terms of number of items are sought. A sequence with four items is different from one with six items. A specific function to measure the similarity is defined. As sequences of items are being manipulated, $Sim_{i,j}$ is defined as a matching function, $S_{i,j}$. The latter quantity measures the similarity with respect to the elements of the sequences associated to a_i and b_j. This absolute similarity measure, $S_{n,m}$, is normalized by dividing it by the maximum sequence depth to produce the overall similarity measure $Sim(A, B)$ between the two sequences A and B:

$$Sim(A, B) = \frac{S_{n,m}}{Max(n, m)} \tag{10}$$

where A and B represent the two sequences to compare, n is the number of items in A and m is the number of items in B. Thus, the Sim measure is used to evaluate the quality of the generated sequences with the objective of minimizing the distance between the generated sequences and the referenced ones.

Mating selection The selection of parents for reproduction is done using the binary tournament method [29]. The latter randomly picks, with replacement, two individuals from the population and then selects the best among them. In the case of tie, random selection is performed. This process is repeated until fulfilling the whole mating pool whose size is equal to the half of the population size. This choice is justified by the fact that binary tournament selection is less sensitive to the distribution skew of the fitness function values of population individuals; thereby it allows avoiding premature convergence by introducing enough diversity in the population.

Environmental selection The selection of individuals for the next generation is performed using elitism, which consists in merging both parents and children

and then selecting the best individuals among them to survive for the next population. This strategy allows preserving the good genetic building blocks for the next generation, which encourages convergence. It is worth noting that pure elitism may incur a lack of diversity. To avoid such a problem, a high mutation rate (greater than 0.2) is used to preserve enough diversity in the population.

Mutation The mutation operator is applied to the selected chromosome to maintain genetic diversity from one generation of individuals to the next one. By using the single-point mutation procedure [7], first, a gene is randomly selected from the chromosome. Then, if the selected gene has a certain class (same nature), it is replaced by another gene from the same class. The mutation process is illustrated in Figure 3.

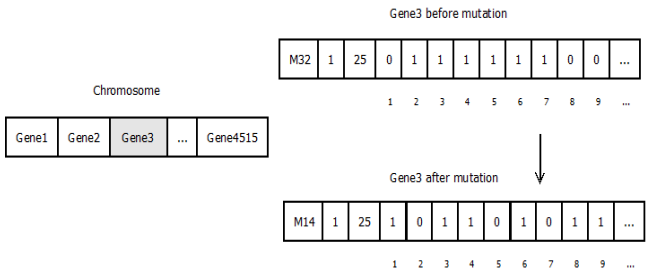


Fig. 3: Example of a mutation operation

Crossover To exploit the best genes of parents, the two-point crossover operator [48] selects two crossover points to create the offspring. It starts with selecting the two parents used for crossover and then randomly selects two crossing points. Two offspring are created by combining the parents at crossover points. An example in Figure 4 is shown to highlight a two-point crossover.

The crossover operator can only be applied on parents that have the same labels. Thus, each child combines information from both parents. In any given generation, a variant will be the parent in at most one crossover operation. For the second population, the crossover operator allows to create two offspring o_1 and o_2 from the two selected parents p_1 and p_2. The crossover process is defined as follows:

- Two random positions $k1$ and $k2$ are selected in the predicate sequences.
- The first $k1$ elements of p_1 become the first $k1$ elements of o_1. Similarly, the first $k1$ elements of p_2 become the first $k1$ elements of o_2.
- The last $k2$ elements of p_1 become the last $k2$ elements of o_1. Similarly, the last $k2$ elements of p_2 become the last $k2$ elements of o_2.
- The remaining elements between $k1$ and $k2$ of, respectively, p_1 and p_2 are added as middle parts of, respectively, o_2 and o_1.

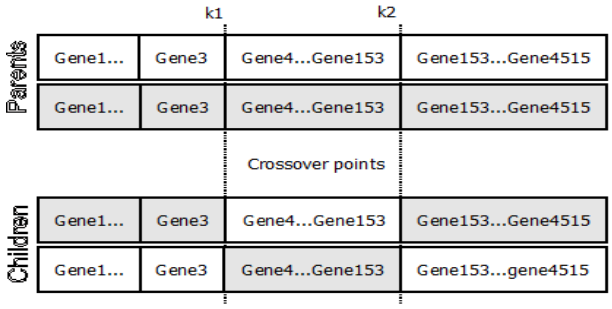


Fig. 4: Example of a crossover operation

3.5 Patterns gathering process

In this step, results generated from the processed steps of the AMD approach will be gathered in one pool. First, let us recall that at the beginning the sets of the frequent extracted API call sequences (the patterns), in both DBM and DBB, were defined. Such information helps in the identification of the executables behavior and hence is important in the patterns gathering process. To extract the patterns, the Apriori algorithm was used. This extraction process is performed twice; first from the database of benign sequences, DBB, and a second time from the extracted API call sequences related to malicious applications; the DBM. The output is hence two data sources namely a database of malicious patterns DBMFIV and a database of benign patterns DBBFIV.

Once achieved, a selection task is performed on the patterns from both DBMFIV and DBBFIV. This task consists in keeping only a filtered set of the malicious and benign patterns by removing the common items between DBMFIV and DBBFIV. As a result, a filtered malicious sets of patterns database (MPDB) and a filtered benign patterns database (BPDB) are generated.

In the current stage, the pattern's gathering process, the obtained MPDB will be enriched by the set of the *artificially generated* malicious patterns that are already stored in their specific database. Let us recall that these generated malicious patterns are obtained after performing an evolutionary algorithm on the set of the filtered malicious patterns from MPDB. The fed new data source, comprising both MPDB and AMDB, will be referred to as the final filtered malicious database (FFMDB).

The outcome of the whole process allows to obtain two sets of patterns. The first pattern set contains the final filtered frequent malicious sets of patterns (FFMDB) while the second patterns set comprises the benign filtered patterns (BPDB). A general workflow description of the patterns gathering process is presented in Figure 5.

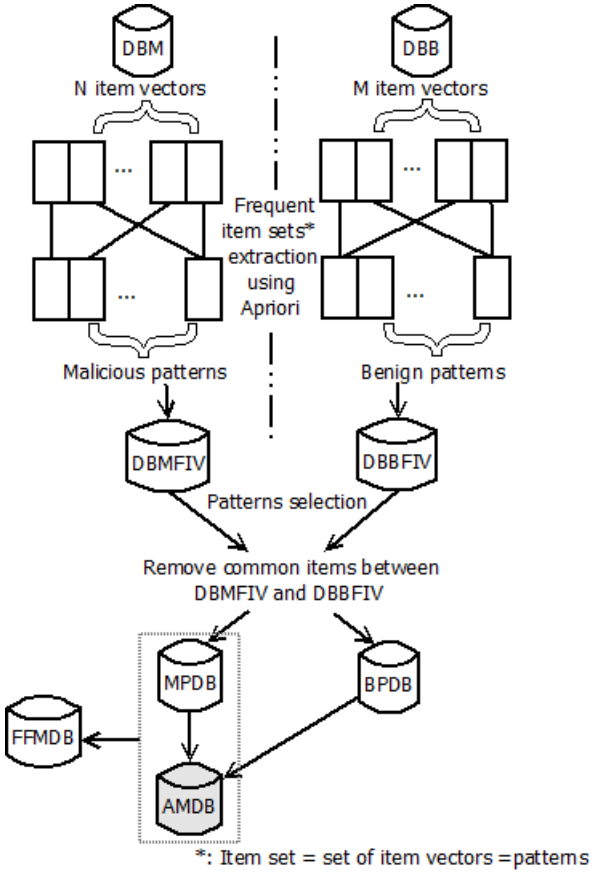


Fig. 5: Workflow of the patterns gathering process using AMDB

3.6 Detection model process using real and artificial malware patterns

Throughout this phase, AMD will perform its classification task where a new app, the executable, will be classified either as a malware or as a benign. This is achieved using the set of patterns presented in both of the FFMDB and BPDB databases. The workflow of this phase is described in Figure 6.

The algorithm for classifying the executable is given in Algorithm 2. Formally, the first step (Algorithm 2, line 1) aims to extract the patterns of the executable. Each pattern will be labeled as benign or as malicious by comparing it to the patterns of the FFMDB and BPDB databases (Algorithm 2, line 2). Then, the support of each labeled pattern is calculated (Algorithm 2, line 3). To represent the set of all the malicious patterns together and all the benign patterns together, a *credence* value is generated for the malicious (Algorithm 2, line 4) and for the benign respectively (Algorithm 2, line 5). If the malicious credence value

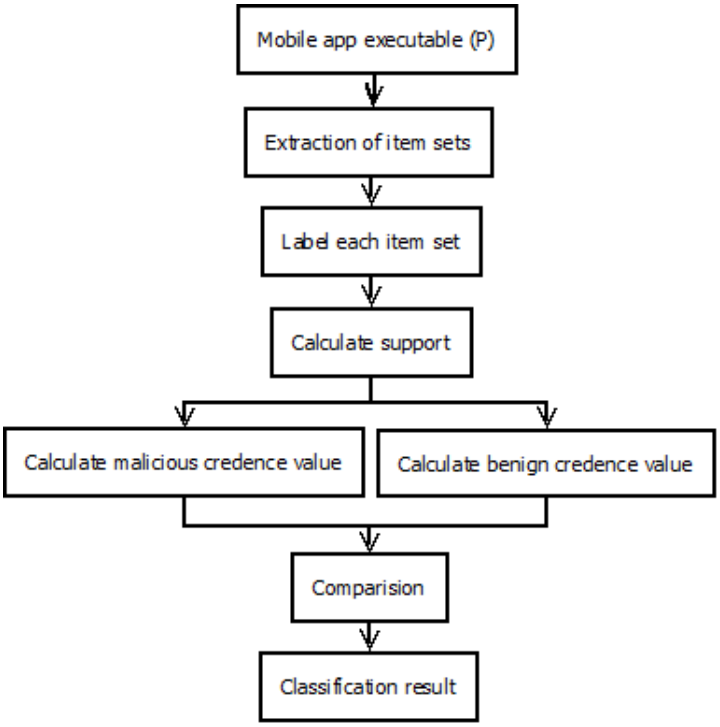


Fig. 6: Detection model process

is greater than the benign credence value, the executable will be classified as a malware (Algorithm 2, lines 6-7); otherwise, it will be classified as benign (Algorithm 2, lines 8-9). The credence refers to how many labeled patterns, MX for malicious patterns or BX for benign patterns, were found among the total number of patterns of an executable P. The credence value of an executable P is calculated twice; once for the malicious patterns (MX) referring to CM and once for the benign patterns (BX) referring to CB. CM and CB are calculated by summing the supports of all their according labeled patterns and by dividing the sum by the total number of patterns in the executable P. CM and CB are highlighted in Equation 11 and Equation 12, respectively, where N indicates the total number of patterns in the executable P, $S_T(MX_i)$ refers to the support of the malicious pattern X_i, and $S_T(BX_i)$ refers to the support of the benign pattern X_i.

$$CM(P) = \frac{\sum_{i=1}^{N} S_T(MX_i))}{N} \tag{11}$$

$$CB(P) = \frac{\sum_{i=1}^{N} S_T(BX_i))}{N} \tag{12}$$

Algorithm 2: The classification task

Inputs: *FFMDB*, *BPDB* : Patterns databases, P: executable
Output: P is malicious or benign

```
Extract the patterns of the executable P
Give a label to each pattern.
Calculate the support for all patterns.
Calculate the malicious credence value of all malicious patterns CM(P)
Calculate the benign credence value of all benign patterns CB(P)
If CM(P) > CB(P) Then
    P is classified as malicious
Else
    P is classified as benign
End if
```

4 Experimental Setup

4.1 Research questions and benchmark data sets

This research study is conducted to quantitatively assess the performance of the proposed AMD malware detection approach when applied in real-world settings. In this concern, a comparative study with a set of well-known state-of-the-art malware detection approaches is performed. More specifically, the aim is to answer the following research questions (RQs):

- RQ1: To what extent can the AMD approach detect malicious apps?
- RQ2: How does the AMD approach perform when compared against the state-of-the-art methods?
- RQ3: What are the benefits of using artificial malicious item sets (patterns)?
- RQ4: What is the difference in terms of performance between using artificial patterns and not using them?

To answer RQ1, the performance of AMD is evaluated using several metrics that are detailed in Section 4.4. To answer RQ2, a comparison is made between the AMD obtained results and those generated by recent prominent state-of-the-art methods as well as to some other antivirus engines. For RQ3, the benefits of using the artificially generated malicious patterns are demonstrated by comparing the AMD approach to the existing approaches that are based on using only existing static sets of malicious apps. We will demonstrate the fact that enriching the malicious data set by artificial malicious patterns will lead to a diversity of the pool of malicious patterns; a necessary task to guarantee a better detection. To answer RQ4, an analysis of the results is made based on the used evaluation metrics (presented in Section 4.4) in case where the artificial malicious patterns are not considered (first step), and then when including these in the malicious pattern database in a second step. Based on these research questions, we will show that the AMD proposed solution outperforms existing malware detection approaches. The details of the used methods for comparison are highlighted in Section 4.5.

To conduct the experiments, 3 000 Android apps were gathered where 2 000 apps are malicious. The gathered apps are obtained from the Android Malware Data set (AMD set) [51], the DROIDCat data set [39] and also from various portable benign tools such as Google play.

For the experimental setup, two tests are performed. For the first test, a balanced data set consisting of 1 000 malicious executables along with 1 000 benign executables is created. In this setting, based on the API calls, a total of 15 822 973 distinct malicious item sets (API call sequences) and a total of 11 302 447 distinct benign item sets were extracted. For the second test, an imbalanced data set is created where all of the collected apps resulting in 2 000 malicious executables and 1 000 benign executables were considered. For this test, a total of 29 483 201 distinct malicious item sets and a total of 11 302 447 distinct benign item sets were extracted. The conducted tests are summarized in Table 3.

Table 3: Number of extracted distinct malicious and benign item sets for each test

Test	Number of apps	Number of obtained distinct item sets
Balanced data set	1 000 benign	11 302 447
	1 000 malicious	15 822 973
Imbalanced data set	1 000 benign	11 302 447
	2 000 malicious	29 483 201

4.2 Parameter settings: Number of obtained frequent item sets according to different values of S_{min}

As mentioned in Section 3.4, dealing with the patterns generation task, the frequent item sets need to be extracted. To achieve this task, two values for the minimum support S_{min} have to be set by the user. The minimum support S^m_{min} is required to extract the malicious item sets while S^b_{min} is needed to extract the benign item sets. For example, for 15 822 973 malicious item sets, a malicious minimum support (S^m_{min}) of 10% means that the frequent item sets (patterns) must be in at least 1 582 298 item sets. If the support is greater than 1 582 298 then the item sets are considered as frequent otherwise the set is not selected. Several S_{min} values were considered to study their effect in the pattern generation process and hence in the evaluation of the overall performance of the AMD approach. The set of the S_{min} considered values are presented in Table 4 where NP refers to the total number of patterns. For instance, the value 1 884 651 refers to the total number of malicious patterns as the value is greater than 1 582 298.

Table 4: Number of extracted malicious and benign frequent item sets for the balanced and imbalanced data sets

Test	Number of distinct item sets		Minimum support S_{min} (%)		NP
Balanced data set	Malicious	15 822 973	S^m_{min}	10	1 884 651
				20	3 355 876
				30	6 145 113
				40	7 531 189
				50	8 023 987
				60	9 495 673
				70	12 076 081
				80	12 891 870
				90	14 775 684
	Benign	11 302 447	S^b_{min}	10	1 205 004
				20	2 260 689
				30	3 390 744
				40	4 521 078
				50	5 651 323
				60	6 781 668
				70	7 911 912
				80	9 051 957
				90	10 172 203
Imbalanced data set	Malicious	29 483 201	S^m_{min}	10	3 275 412
				20	5 896 652
				30	8 845 960
				40	11 793 380
				50	14 741 700
				60	17 689 820
				70	20 838 240
				80	23 587 560
				90	27 534 880
	Benign	11 302 447	S^b_{min}	10	1 205 004
				20	2 260 689
				30	3 390 744
				40	4 521 078
				50	5 651 323
				60	6 781 668
				70	7 911 912
				80	9 051 957
				90	10 172 203

4.3 Genetic algorithm settings and artificial generated sequences

In this section, the aim is to fix the number of artificially generated malicious patterns to maintain a fairly varied and renewed database of malicious patterns.

To perform the patterns generation task, the determination of the GA parameters is essential as it will have a vital impact on the effectiveness of the overall task. The values of various parameters that are the population size, the

individual's size, the number of evolutions or generations, the selection process, the mutation rate and the crossover rate, need all to be set. As experiments are based on the application of the GA, this may provide different results over multiple repeated runs. For each experiment, an initial population is made using the DBFFIV data set. Each chromosome in this data set is represented as the already given description in Section 3.4. The portion of the population, which is not fitting, is removed. Crossover and mutation are applied in the rest of the population which becomes the population of a new generation. The process runs for 1 000 generations. The group of the generated chromosomes will be used as one of the inputs in the next step: malicious pattern generation (Section 3.4). For the conducted experiments, 68 000 malicious patterns were gathered with Eclipse[6] (about a quarter of the number of the generated malicious patterns) with a total number of 4 407 API calls (items). The population size is fixed to 100 and the number of generations to 1 000. In this way, the applied GAm performs 100 000 evaluations. Trial and error method was used to set the population size and the number of generations for the algorithm. This means that several experiments were made using different values for these parameters. Based on these experiments, we concluded that when using a population size of 100 for the GA, the fitness function becomes stabilized around the $1\,000^{th}$ generation. For this reason, the algorithm did not suffer from premature convergence; thereby the comparison is fair not only from the stopping criterion viewpoint but also from the parameter setting one. For the variation operators, a crossover rate of 0.9 and a mutation rate of 0.5 were used.

4.4 Performance indicators

For the evaluation of the AMD approach, proper metrics were used that allow the comparison of AMD to previous works, and to deepen the analysis of the obtained results. As imbalanced data sets are dealt with, and to avoid the accuracy paradox, the performance of AMD is evaluated in terms of precision, recall, specificity, F1 score, Area Under the Receiver Operating Characteristics (ROC) Curve (AUC) and accuracy. Also, to further interpret the obtained results, the ROC curve analysis is used. All the used measures are defined in Appendix C.

All conducted experiments are run on a Lenovo 4.00 Go 2.7 Ghz machine. We note that the number of generations of the GA is 1 000 and the obtained averaged execution time is about 1 hour and 13 minutes.

4.5 Methods and approaches under comparison

To compare the AMD results to other existing works, a set of well-known antivirus engines and two published state-of-the-art approaches that are somehow similar to AMD were selected. These are the Tong et al.'s approach [47] and the Kayacık et al.'s approach [26]. The work proposed in [47] is a hybrid approach for malware detection in Android OS. In [47], a malicious pattern set and a normal

[6] https://www.eclipse.org/

pattern set were built by comparing the patterns of malware and benign apps with each other. To do so, Tong et al. use a dynamic method to collect its system calling data. Then, the collected data is compared to both the malicious and normal pattern sets offline in order to judge the unknown app. The methodology behind Kayacık et al.'s approach [26] is to generate evasion attacks (mimicry attack) in order to detect the vulnerability testing of host-based anomaly detectors. The performance of the proposed AMD approach is compared to Tong et al.'s approach [47] in terms of the sensitivity, the specificity and the accuracy criteria. Additionally to these criteria, AMD is compared to Kayacık et al.'s approach [26] in terms of false positive and false negative rates. Further comparisons are made against a set of different antivirus engines. VirusTotal[7] is used, which is a subsidiary of Google and which is a free online service that analyzes files and URLs by different antivirus engines and website scanners. The used antivirus engines are Cyren[8], Ikarus[9], VIPRE[10], McAfee[11], AVG[12], AVware[13], ESET NOD32[14], CAT QuickHeal[15], AegisLab[16] and NANO[17].

5 Results and Discussion

5.1 Performance analysis

In this section, results obtained using the AMD approach are discussed and thereby we respond to RQ1 and RQ2 highlighted in Section 4.1. For this purpose, two sets of tests (as presented in Table 3) were conducted with several runs: a first test in which 14 775 684 malicious patterns and 10 173 203 benign patterns were analyzed and a second test in which 27 534 880 malicious patterns and 10 173 203 benign patterns were analyzed. In each one of these tests, different values of S^m_{min} and S^b_{min} were set and in each case the resultant precision, F1_score, AUC, recall, specificity and accuracy were calculated and registered as presented in Table 5 and Table 6. Then, the values of S^m_{min} and S^b_{min} which will allow to have better performance results in terms of the evaluated metrics were retained.

[7] https://www.virustotal.com
[8] https://www.cyren.com
[9] https://www.ikarussecurity.com
[10] https://www.vipre.com
[11] https://www.mcafee.com
[12] https://www.avg.com
[13] http://www.avware.com.br/comprar.php
[14] https://www.eset.com
[15] www.quickheal.com
[16] www.aegislab.com
[17] http://www.nanoav.ru

Table 5: The different obtained measures for both balanced and imbalanced data sets in terms of TP, FP, TN, FN, precision, F1_score and AUC

S_{min}		Values for balanced data set							Values for imbalanced data set						
S^m_{min}	S^b_{min}	TP	FP	TN	FN	Pre	FS	AUC	TP	FP	TN	FN	Pre	FS	AUC
10	10	98.00	02.00	98.63	01.37	98.01	98.31	57.69	95.02	04.98	92.30	07.70	99.24	99.39	73.04
	20	98.39	01.61	98.71	01.29	98.39	98.55		98.58	01.42	96.32	03.68	98.54	97.46	
	30	97.77	02.23	98.50	01.50	97.78	98.13		99.23	00.77	98.5	01.50	99.22	98.86	
	40	95.24	04.76	95.32	04.68	96.13	97.15		98.65	01.35	97.03	02.97	98.62	97.84	
	50	97.29	02.71	98.36	01.64	98.34	97.31		92.80	07.20	31.24	68.76	81.26	67.30	
	60	89.52	10.48	95.21	04.79	85.74	90.56		98.81	06.19	90.90	09.10	93.62	92.37	
	70	98.19	01.81	98.71	01.29	98.19	98.49		98.39	01.61	93.25	06.75	98.30	95.88	
	80	98.50	01.50	98.96	01.04	98.95	98.50		99.54	00.46	98.73	01.27	99.53	99.13	
	90	99.78	00.22	99.34	00.66	99.77	99.56		98.41	01.59	95.90	04.10	98.36	97.17	
20	10	98.50	01.50	96.75	03.25	98.47	97.63	74.59	99.78	00.22	99.58	00.42	99.78	99.68	63.69
	20	9.23	00.77	98.63	01.37	99.22	89.93		98.62	01.38	98.81	01.19	98.62	98.71	
	30	98.61	01.39	97.03	03.00	98.58	97.41		90.25	09.75	97.25	02.75	90.89	93.86	
	40	95.00	05.00	90.00	10.00	94.73	92.55		78.92	21.08	96.75	03.25	82.11	88.53	
	50	93.80	06.20	28.61	71.39	82.18	67.16		62.30	37.70	35.04	64.96	48.17	48.56	
	60	93.81	06.19	89.30	10.70	93.51	91.60		95.23	04.77	98.29	01.71	95.37	96.78	
	70	98.39	01.61	96.035	03.79	98.35	97.22		93.81	06.19	96.75	03.25	93.17	95.60	
	80	99.24	00.76	99.78	00.22	99.24	99.51		98.68	01.32	98.55	01.45	98.55	98.55	
	90	99.20	00.80	98.55	01.45	99.19	98.87		99.04	00.96	98.96	01.04	99.03	99.00	
30	10	98.19	01.81	92.89	07.11	94.881	93.67	65.06	78.54	21.46	52.79	47.21	71.10	66.49	64.99
	20	98.50	01.50	96.75	03.25	98.47	97.63		98.62	01.38	98.50	01.50	98.62	98.56	
	30	99.20	00.80	98.55	01.45	99.19	98.87		98.52	01.48	96.32	03.68	96.40	98.49	
	40	98.61	01.39	97.00	03.00	98.58	97.81		96.05	03.95	98.24	01.76	95.24	95.28	
	50	95.00	05.00	90.90	09.10	94.68	92.98		98.54	01.46	96.46	03.54	98.49	97.51	
	60	92.30	07.70	28.41	71.59	92.37	93.62		98.68	01.32	92.18	07.82	90.08	92.43	
	70	98.39	01.61	96.03	03.97	98.35	97.23		98.99	01.01	98.85	01.15	98.89	98.92	
	80	99.24	00.76	98.63	01.37	99.23	98.93		98.61	01.39	98.21	01.79	98.60	98.41	
	90	93.81	06.19	90.00	10.00	93.56	91.93		99.12	00.88	99.85	00.15	98.98	99.92	
40	10	99.24	00.76	99.55	00.45	99.24	99.39	80.41	96.23	03.77	90.12	09.88	95.98	93.26	75.51
	20	98.50	01.51	97.02	02.98	98.47	97.76		98.30	01.70	97.68	02.32	98.29	97.99	
	30	99.20	00.80	98.71	01.29	99.19	98.95		98.61	01.39	98.35	01.65	98.61	98.48	
	40	98.61	01.39	98.06	01.94	98.60	98.33		98.19	01.81	96.75	03.25	98.16	97.47	
	50	95.21	04.79	96.96	03.04	95.29	96.09		98.50	01.70	97.68	02.32	98.49	98.29	
	60	92.30	07.70	29.79	70.21	79.46	66.24		99.20	00.80	99.78	00.22	99.20	99.49	
	70	98.39	01.61	93.25	06.75	93.98	95.30		98.61	01.39	98.63	01.37	98.61	98.62	
	80	99.54	00.77	98.50	01.50	98.67	98.61		95.00	05.00	89.98	10.02	94.74	92.54	
	90	99.20	00.80	98.92	01.08	99.19	99.06		92.30	07.70	28.61	71.39	78.79	65.73	
50	10	98.39	01.61	97.43	02.75	98.39	97.91	55.58	98.39	01.61	98.07	01.93	98.38	98.23	52.05
	20	98.00	02.00	90.25	09.75	97.83	94.26		99.24	00.76	99.52	00.48	99.24	99.38	
	30	98.14	01.68	95.00	05.00	98.07	96.59		93.81	06.19	89.74	10.26	93.55	91.81	
	40	86.90	**16.10**	80.01	**19.99**	97.38	85.92		98.32	01.68	93.44	06.56	98.23	95.93	
	50	97.43	02.75	90.00	10.00	97.23	93.84		98.44	01.56	98.32	01.68	98.48	98.38	
	60	98.68	01.32	97.61	02.39	98.66	98.14		98.32	01.68	96.37	03.63	98.29	97.35	
	70	98.92	01.08	98.20	10.80	98.91	98.56		95.03	04.97	89.92	10.08	94.76	92.53	
	80	99.76	00.24	98.63	01.37	99.75	99.19		97.85	02.15	91.92	08.08	97.71	94.96	
	90	93.96	06.40	85.56	14.44	93.40	89.91		99.62	00.38	99.55	00.45	99.62	99.55	
60	10	99.72	00.28	99.50	00.50	99.31	99.40	**95.82**	95.63	04.37	90.47	09.53	95.39	93.11	68.81
	20	99.22	00.69	99.76	00.24	99.22	99.49		98.23	01.77	95.37	04.63	98.18	96.81	
	30	96.89	03.11	92.45	07.55	96.74	94.61		99.52	00.48	99.45	00.55	99.51	99.48	
	40	90.35	09.65	90.99	09.01	90.67	90.93		98.32	01.68	95.99	04.01	96.08	98.28	
	50	99.31	00.69	99.76	00.24	99.31	99.53		89.52	10.48	31.24	68.76	74.88	64.44	
	60	98.18	01.82	99.25	00.75	98.19	98.71		90.21	09.79	85.23	14.77	89.69	87.77	
	70	58.49	41.51	83.96	16.04	66.91	72.23		98.52	01.48	96.07	03.93	98.48	97.30	
	80	99.21	00.79	99.50	00.50	99.21	99.31		99.02	00.98	98.74	01.26	99.01	98.88	
	90	96.89	03.11	92.45	08.10	92.40	92.17		98.78	01.22	98.20	01.80	98.77	98.49	
70	10	99.22	00.78	99.52	00.48	99.22	99.37	65.41	78.97	21.03	83.45	16.55	79.87	81.24	66.94
	20	98.52	01.48	96.88	03.12	98.49	97.70		90.23	09.77	93.58	06.42	90.55	91.93	
	30	99.20	00.80	98.77	01.23	99.19	98.98		98.74	01.26	98.30	01.70	98.73	98.52	
	40	98.65	01.35	98.72	01.28	87.47	86.01		86.25	**13.75**	91.98	**08.02**	87.00	89.18	
	50	97.43	02.75	90.00	10.00	97.23	93.84		95.21	04.79	96.20	03.80	95.26	95.70	
	60	98.68	01.32	97.61	02.39	98.66	98.14		94.28	05.72	95.30	04.70	94.34	94.79	
	70	98.92	01.08	98.20	01.80	98.91	98.56		98.22	01.78	96.97	03.03	98.20	97.59	
	80	99.76	00.24	98.63	01.37	99.75	99.19		99.70	00.30	98.75	01.25	99.69	99.22	
	90	93.96	06.40	85.56	14.44	93.40	89.91		95.22	04.78	96.89	03.11	95.30	96.06	
80	10	97.02	02.98	90.00	10.00	96.79	93.62	73.70	99.12	00.88	99.52	00.48	99.22	99.49	83.12
	20	98.50	01.50	97.00	03.00	98.47	97.75		99.02	00.98	99.45	00.55	99.02	99.23	
	30	99.23	00.77	98.63	01.37	98.63	99.22		97.52	02.48	93.07	06.93	97.40	95.34	
	40	98.65	01.35	98.19	01.81	98.19	98.64		91.11	08.89	92.56	07.44	91.24	91.83	
	50	92.80	07.20	29.42	70.58	80.33	66.54		99.05	00.95	99.23	00.70	99.05	99.14	
	60	93.81	06.19	87.98	12.02	93.42	90.09		98.65	01.35	93.45	06.55	98.58	96.11	
	70	98.39	01.61	96.07	03.93	98.35	97.24		99.22	00.78	99.65	00.35	99.43	72.73	
	80	99.54	00.46	99.78	00.22	99.54	99.66		55.25	44.75	87.00	13.00	80.95	66.03	
	90	98.41	01.59	96.75	03.25	98.39	97.58		90.31	09.69	92.03	07.97	90.47	91.17	
90	10	90.00	10.00	94.14	05.86	99.61	99.38	90.83	99.75	00.25	98.25	01.75	99.75	99.04	**8**4.68
	20	99.18	00.82	93.04	06.96	99.12	96.20		99.24	00.76	95.22	04.78	99.21	97.26	
	30	99.79	00.21	99.52	00.48	99.79	99.51		99.83	00.16	99.24	00.76	93.96	99.51	
	40	99.45	00.55	95.53	04.47	99.10	96.28		99.20	00.80	93.62	06.38	99.37	97.15	
	50	99.08	00.92	93.80	06.20	99.65	98.77		99.74	00.26	98.20	01.80	99.13	96.64	
	60	98.68	01.32	92.18	07.82	98.58	95.53		75.12	24.88	92.38	07.62	78.78	84.36	
	70	99.41	00.59	95.52	04.48	99.41	97.57		99.78	00.20	99.05	00.95	99.78	99.41	
	80	99.79	00.21	99.59	00.41	**99.80**	**99.71**		99.72	00.28	99.56	00.44	99.72	**99.71**	
	90	99.16	00.84	92.65	07.35	99.10	96.00		99.89	00.11	99.35	00.65	**99.89**	99.62	

FS: F1_score; Pre: precision; TP: true positive rate; FP: false positive rate; TN: true negative rate; FN: false negative rate.

False positives and false negatives analysis In this section, the AMD performance is discussed and analyzed with a particular focus on false positives and false negatives. False negatives (also known as type 2 errors) may be a significant problem. However, the majority of researchers are more likely inclined to accept an increase in false positives (type 1 errors) since they are judged as a less significant problem than the false negatives. In this analysis of the results, the aim is to keep both the type 1 and type 2 errors as low as possible. For both conducted tests, the highest values, as shown in Table 5, of FP and FN were obtained for $S^m_{min} = 50$ and $S^b_{min} = 40$ with FP= 16.10% and $FN = 19.99\%$ for the balanced data set, and with $FP = 13.75\%$ and $FN = 8.02\%$ for $S^m_{min} = 70$ and $S^b_{min} = 40$ for the imbalanced data set. In the aim of reducing the number of FPs and FNs, the base of examples can be increased with more benign and malicious patterns. But, it is important to not make the detection model over-fitting, causing detection performance degradation.

Precision interpretation Precision is used to have an idea about how precise and accurate AMD is. Precision indicates the number of predicted positive instances that are actual positive. It is a good measure to determine specially when the amount of false positives is high. For instance, in the current detection model, a false positive means that a pattern that is benign (actual negative) has been identified as malicious. Consequently, the detection model might refuse important apps if the precision is not high. From Table 5, it is noted that the best reached precision value for the balanced data set is 99.80% for $S^m_{min} = 90$ and $S^b_{min} = 80$ and 99.89% for the imbalanced data set for $S^m_{min} = 90$ and $S^b_{min} = 90$. Based on these results, it is concluded that the proposed AMD approach is able to classify new instances with a high precision. In fact, these results can be explained by the inclusion of the generated malicious patterns in the detection process which is undoubtedly benefiting in keeping the base of examples varied as much as possible.

Accuracy, recall and specificity interpretation Accuracy is the most intuitive performance measure and it is the ratio of a correctly predicted observation to the total observations. Having a high accuracy does not necessarily mean that AMD is the best. In fact, accuracy is a good measure but only when symmetric data sets are considered where values of false positives and false negatives are almost the same. Therefore, other parameters (i.e., precision and recall) were considered to evaluate the performance of AMD. For instance, from Table 6, an accuracy of 99.69% for the balanced data set for $S^m_{min} = 90$ and $S^b_{min} = 80$ and an accuracy of 99.64% for the imbalanced data set for $S^m_{min} = 90$ and $S^b_{min} = 90$ were registered. This means that AMD is approximately 100% accurate which is explained by the large number of correctly predicted observations. These good results clearly demonstrate the impact of the AMD detection model which is not dependent on a static base of examples but rather, the base of examples is quite varied thanks to the artificially generated patterns using the genetic algorithm.

On the other hand, the recall metric calculates how many of the actual positives AMD captures through labeling them as positive (true positive). Applying the same understanding, recall shall be the model metric to be used to select our best model when there is a high cost associated with false negatives. For instance, if a fraudulent behavior (actual positive) is predicted as non-fraudulent (predicted negative), the consequence

can have very bad effects on the operating system and similarly for the user. In the conducted experiments, a value of 99.54% as recall for the balanced data set for S^m_{min} = 80 and S^b_{min} = 80 and a value of 99.78% as recall for the imbalanced data set for S^m_{min} = 40 and S^b_{min} = 60 can be positively interpreted. In fact, these satisfying values can be explained by the high number of true positives accurately detected with 99.54% for the balanced data set and 99.20% for the imbalanced data set for the same values of S^m_{min} and S^b_{min}, respectively.

Based on the fact that the sensitivity (recall) quantifies the avoiding of false negatives, the specificity, then, does the same for false positives. For any prediction, there is usually a trade-off between these measures. A perfect predictor would be described as 100% sensitive, meaning all malicious instances are correctly identified as malware, and 100% specific, meaning no benign instances are incorrectly identified as malicious. Via the obtained results, a value of 99.75% for the specificity when using the balanced data set was obtained for S^m_{min} = 90 and S^b_{min} = 80 and a value of 99.89% for the specificity when using the imbalanced data set was obtained for S^m_{min} = 90 and S^b_{min} = 90 can be considered as very promising results. These results can be explained by the high number of true negatives accurately detected. With a quite varied base of examples, a better detection of malicious patterns is guaranteed.

Table 6: Accuracy, recall and specificity values for both the balanced and imbalanced data set

S_{min}		Values for the balanced data set			Values for the imbalanced data set		
S^m_{min}	S^b_{min}	Recall	Specificity	Accuracy	Recall	Specificity	Accuracy
10	10	98.62	98.01	98.31	92.50	94.88	93.66
	20	98.71	98.39	98.55	96.40	98.55	97.45
	30	98.48	97.78	98.13	98.51	99.22	98.87
	40	95.32	95.24	95.28	97.08	98.63	97.84
	50	98.34	97.31	97.82	57.44	81.27	62.02
	60	94.92	90.08	92.37	91.16	93.62	92.36
	70	98.70	98.19	98.45	93.58	98.30	95.82
	80	98.95	98.50	98.73	98.74	99.54	99.14
	90	99.34	99.77	99.56	96.00	98.37	97.16
20	10	96.80	98.47	97.62	99.58	99.78	99.68
	20	98.63	99.22	98.93	98,81	98.62	98.72
	30	97.04	98.58	97.80	97.04	90.89	93.75
	40	90.47	94.73	92.50	96.04	82.11	87.84
	50	89.79	93.51	91.55	48.95	48.17	48.67
	60	96.12	98.35	97.21	98.24	95.37	96.76
	70	56.78	82.18	61.20	96.65	93.98	95.28
	80	99.77	99.24	99.51	98.55	98.67	98.61
	90	98.55	99.19	98.87	98.96	99.04	99.00
30	10	93.24	98.08	95.54	62.46	71.10	65.67
	20	96.80	98.47	97.62	98.50	98.62	98.56
	30	98.56	99.19	98.87	96.40	98.49	97.42
	40	97.04	98.58	97.80	98.20	96.13	97.14
	50	91.25	94.78	92.95	96.40	98.49	97.42
	60	56.31	78.67	60.35	92.65	98.58	95.43
	70	96.12	98.35	97.21	98.85	98.99	98.92
	80	98.63	99.23	98.93	98.22	98.60	98.41
	90	90.36	93.56	91.90	99.85	99.13	99.49
40	10	99.54	99.24	99.39	90.69	95.98	93.18
	20	97.06	98.47	97.76	97.69	98.29	97.99
	30	98.71	99.19	98.93	98.35	98.61	98.48
	40	98.07	98.60	98.35	96.80	98.16	97.47
	50	96.90	95.29	96.08	97.69	98.29	97.99
	60	56.79	79.46	61.04	**99.78**	99.20	99.49
	70	93.58	98.30	95.82	98.63	98.61	98.62
	80	98.51	99.22	98.87	90.46	94.74	92.49
	90	98.92	99.19	99.06	56.39	78.79	60.46
50	10	97.45	98.39	97.91	98.08	98.38	98.23
	20	90.95	97.83	94.12	99.52	99.24	98.38
	30	95.15	98.07	96.57	90.14	93.55	91.78
	40	90.69	97.23	93.71	93.75	98.23	95.88
	50	97.63	98.66	98.14	98.32	98.44	93.38
	60	98.21	98.11	98.56	96.44	98.29	97.35
	70	84.59	87.47	85.98	90.41	94.76	92.48
	80	98.64	99.75	99.19	92.37	97.71	94.89
	90	86.67	93.40	89.76	99.55	99.62	99.59

60	10	99.49	99.31	99.40	90.94	95.39	93.05
	20	99.75	99.22	99.49	95.50	98.18	96.80
	30	92.77	96.74	94.67	99.45	99.52	99.49
	40	90.93	90.41	90.67	96.08	98.28	97.16
	50	99.75	99.31	99.53	56.56	74.88	60.38
	60	78.49	66.91	71.25	85.93	89.70	87.72
	70	99.49	99.21	99.35	96.16	98.48	97.03
	80	99.24	98.19	98.71	98.74	99.02	98.88
	90	91.94	92.40	92.17	98.21	98.77	98.49
70	10	99.51	99.22	99.37	82.67	79.87	81.21
	20	96.93	98.49	97.70	93.36	90.55	91.91
	30	98.77	99.19	98.98	98.31	98.73	98.52
	40	98.71	98.65	98.68	91.49	87.00	98.12
	50	96.76	98.27	97.50	96.16	95.26	95.71
	60	92.42	96.89	47.37	95.25	94.34	94.79
	70	90.42	94.30	92.28	97.01	98.20	97.60
	80	92.49	97.23	94.74	98.76	99.70	99.23
	90	99.54	99.54	99.54	96.84	95.30	96.06
80	10	90.65	96.79	93.51	99.52	99.12	99.32
	20	97.04	98.47	97.75	99.45	99.02	99.24
	30	98.63	99.22	98.93	93.37	97.40	95.30
	40	98.19	98.64	98.42	92.45	91.24	91.84
	50	56.80	80.33	61.11	99.23	99.05	99.14
	60	88.64	97.37	94.63	93.77	98.58	96.05
	70	96.15	98.35	97.23	99.65	99.22	99.44
	80	**99.54**	99.79	99.66	80.95	66.03	71.13
	90	96.80	98.38	97.58	91.89	90.47	91.17
90	10	99.08	99.55	99.27	98.28	99.75	99.00
	20	92.99	99.12	96.02	95.40	99.21	97.23
	30	99.61	99.41	99.45	99.24	99.84	99.54
	40	95.80	99.57	97.30	93.96	99.15	96.41
	50	94.27	99.13	96.45	98.23	99.74	98.97
	60	92.65	98.48	95.43	90.79	78.78	83.75
	70	95.74	99.52	97.56	99.06	99.78	99.42
	80	99.65	**99.75**	**99.69**	99.56	99.72	**99.64**
	90	93.09	99.10	95.90	99.35	**99.89**	99.62

F1_score and AUC interpretation F1_score can be defined as the mean of the precision and the recall. From Table 5, it is noted that for $S^m_{min} = 90$ and $S^b_{min} = 80$, a value of 99.71% is reached for F1_score for the balanced data set and a value of 99.71% is reached for F1_score for the imbalanced data set and this could be explained by the high values of precision and recall achieved by the AMD model. For the same values of S^m_{min} and S^b_{min}, a value of 99.84% of precision and a value of 99.82% of recall are registered for the balanced data set and a value of 99.72% of precision and a value of 99.65% of recall are registered for the imbalanced data set.

The area, for its part, measures discrimination, that is, the ability of the pattern to correctly classify positive and negative instances. When considering the situation in which instances are already correctly classified into two sets (malicious and benign), we randomly pick one from the malicious set and one from the benign set and then run the test on both instances. The instance with the highest test result should be the one from the malicious set. More precisely, the AUC is the percentage of randomly drawn pairs for which the test correctly classifies the two instances in this random pair. This means that the probability that a randomly selected malicious instance has a test result indicating greater suspicion than that of a randomly chosen benign instance. The best AUC value is obtained with the following values of S^m_{min} : $S^m_{min} = 60$ when using the balanced data set and $S^m_{min} = 90$ when using the imbalanced data set. In fact, AUC equals 95.82% for the balanced data set and 84.68% for the imbalanced data set, which means that the achieved patterns, those retained for the same values of S^m_{min} and S^b_{min}, for both experiments, can be considered efficient in separating malicious and benign instances. Hence, it is concluded that if a continuous variability is assumed to the base of examples by injecting the generated malicious patterns, a better detection of malware is guaranteed.

Graphical analysis For further analysis, a graphical based evaluation is performed. For this aim, the Receiver Operating Characteristics (ROC curve) analysis is used. Basically, the ROC curve is a graphical representation of detection probability versus false positive rate, or, true positive rate versus false positive rate. The most motivating reason behind using this analysis is its ability in summarizing the achieved accuracy of a detector system.

The obtained results are presented by the mean of twographics/curves for each test where one curve in drawn in terms of accuracy vs false positive rates and the other is in terms of true positive rate vs false positive rates. Figure 7a represents the obtained ROC curves for the first test when using the balanced data set while Figure 7b represents the obtained ROC curves for the second test when using the imbalanced data set .

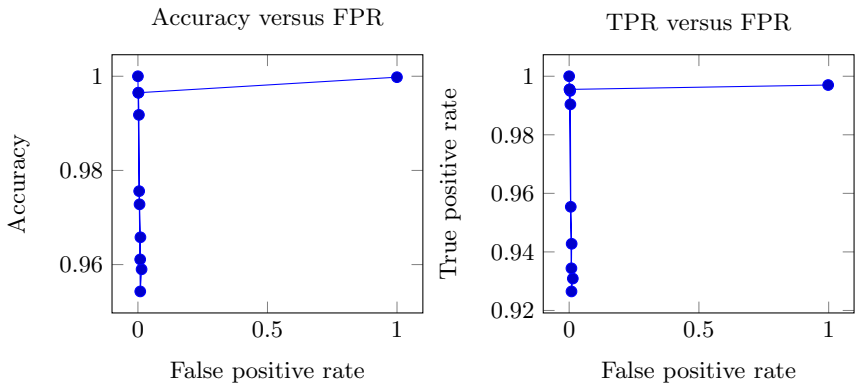
The four ROC curves are used to choose the most appropriate cut-offs for the conducted experiments. The first best cut-off in the first test has the highest accuracy of a value 99.84%, the highest true positive rate (recall) of a value 99.85%, and the lowest false positive rate (1-specificity) of 00.15%. In the second test, the best cut-off has a highest accuracy of 99.84%, the highest true positive rate of 99.72%, and the lowest false positive rate of 00.28%. All obtained ROC curves follow closely the left-hand border and also the top border of the ROC space which shows that the obtained results are accurate. Despite the good shapes obtained by plotting the ROC curves, this cannot be sufficient to give a real interpretation of the reached results. That is why, the AUC value is calculated which serves as a quantitative summary to evaluate the strength of the AMD retained patterns in classifying positive and negative instances.

From Table 6, it is noticed that for high values of S^m_{min} and S^b_{min} better accuracy results are obtained. This observation may be explained by the fact that the obtained number of malicious and benign patterns are important. Hence, the resulting detection patterns will be more accurate.

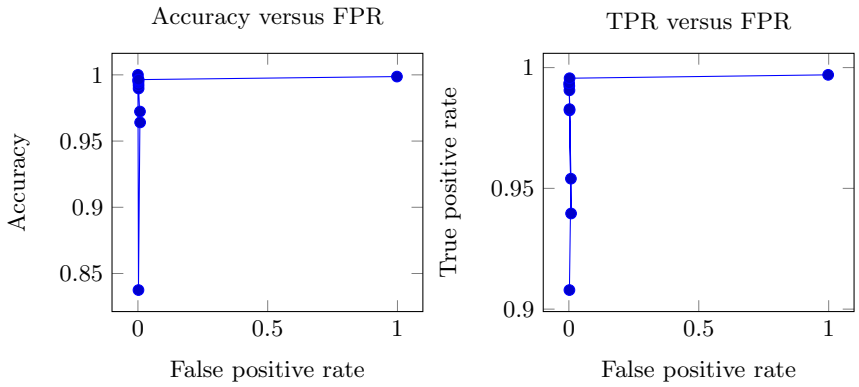
Based on these evaluations, it is important to decide which values of S^m_{min} and S^b_{min} should be kept. As previously analyzed with the different used evaluation metrics, in each row of Table 6, for each experiment, the best accuracy of each value of S^m_{min} and S^b_{min} for both experiments will be sought. In fact, the results illustrate that the values $S^m_{min} = 90$ and $S^b_{min} = 80$ are the best values as they procure the best accuracy results with 99.69% in experiment 1 and 99.64% in experiment 2. We also notice that, overall, these two values of S_{min} are the best values extracted based on the set of metrics used in previous evaluations. Hence, these two values, i.e., $S^m_{min} = 90$ and $S^b_{min} = 80$, are used to extract both of the malicious and benign patterns. Which means that 2 7534 880 malicious patterns and 9 051 957 benign patterns are kept to build the AMD detection model. It is worth mentioning that in several works such as in [53,11,35], authors focused mainly on the accuracy criterion among other evaluation metrics to select the S_{min} values. However, in this study, the choice of the S_{min} values was not only based on the best values registered for the accuracy but it was also endorsed by the other values for the different evaluation metrics. In Table 7, an example of an obtained malicious and benign patterns in AMD is given.

5.2 Evaluation of the contribution of the AMD approach

In this section, two experiments are performed as an answer to RQ3 and RQ4 highlighted in Section 4.1. The first experiment will evaluate the AMD approach in terms of sensitivity, specificity and accuracy using only the base of examples provided by [51]



(a) AMD's ROC curvesusing the balanced data set



(b) AMD's ROC curves using the imbalanced data set

Fig. 7: AMD's obtained ROC curves

Table 7: Examples of obtained malicious and benign patterns

Benign pattern	**Malicious pattern**
read,loadClass,loadClass,loadClass, read,loadClass,loadClass,loadClass, write,loadClass,loadClass,write,open read,close,close,close,close,doFinal exit,doFinal	generateKey,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass,loadClass, loadClass,loadClass,loadClass, getDisplayMessageBody,getInstance, getInstance,getInstance,getInstance,close, close,close,close,close,doFinal,doFinal, exit,doFinal

and [39]. In the second conducted experiment, the malicious sequences generated by the GA are injected in the base of examples and hence, only the best reached values of

sensitivity, specificity and accuracy are kept in order to compare them to the previously obtained values in the first experiment.

When applying the GA, the population size is fixed to 100 item sets and the number of generations to 1 000. In this way, the algorithm performs 100 000 evaluations and the obtained population (2 000 patterns) will be added to the previously gathered malicious patterns (68 000 patterns).

The obtained results in Table 8 clearly show the important gain from including the artificial malicious generated patterns in the base of examples and how they clearly improve the accuracy, the sensitivity and the specificity. An improvement of 10.32% in terms of accuracy is obtained when using the artificial patterns. Also, the recall has risen from 90.02% in the first experiment to 99.78% in the second one. Furthermore, the specificity has gone up by 9.59%.

Table 8: Effect of the use of the artificial malware patterns

Measure	Results	
	without artificial malicious patterns	with artificial malicious patterns
Accuracy	89.32%	99.64%
Recall	90.02%	99.78%
Specificity	90.30%	99.89%

The generated patterns will keep the base of examples quite varied. Also, this experiment guarantees that the result of the detection is not dependent on the base of examples. In this way, it was possible to develop a new malware detection approach that would be as independent as possible from the base of examples and thus would be much more effective in detecting different variants of malware.

5.3 Comparison with the antivirus engines and the state-of-the-art approaches

In this section, the performance of the AMD approach is compared to different antivirus engines, to Tong et al.'s approach [47] and to Kayacık et al.'s approach [26]. The recall, specificity and accuracy criteria are used to compare AMD to Tong et al.'s approach. Addiotionallt to these criteria, the FP and FN values are added in order to compare AMD to Kayacık et al.'s approach.

Comparison with antivirus engines To compare AMD to the used antivirus engines, a set of 3 000 benign and malicious applications (the same set as the one used in the first phase, i.e., the API call sequences extraction phase) is analyzed with the different antivirus engines provided by VirusTotal. This provides a decision, whether benign or malicious, for each application uploaded to the website by each of the antiviruses. This allowed to calculate their accuracies as the percentage of correctly labeled or classified apps. Table 9 summarizes the accuracy results of the used antivirus engines in comparison to the AMD approach.

Table 9: Accuracy results of our AMD and the top ten commercial engines applied by Virus-Total

Malware detection engine	Accuracy (%)
Our AMD approach	99.75
Cyren	83.03
Ikarus	82.72
VIPRE	82.53
McAfee	82.45
AVG	82.36
AVware	81.95
ESET NOD32	81.81
CAT QuickHeal	81.79
AegisLab	81.74
NANO antivirus	81.15

From Table 9, we can notice that the best antivirus engines accuracy value is set to 83%, which is lower than the accuracy obtained by AMD (99.64%). This gap, which is higher than 16%, can be explained by the effect and consequences of the obfuscation techniques that cannot be detected by the antivirus engines. Also, the benefits of introducing the artificially generated patterns in achieving such high accuracy value can be seen.

Comparison with state-of-the-art approaches In this section, a comparison between AMD and two state-of-the-art approach namely Tong et al.[47]'s approach and Kayacık et al.'s approach [26] is performed. In fact, the same set of 3 000 benign and malicious applications was analyzed to perform this comparison.

Table 10 shows the comparison results in terms of accuracy, recall and specificity for both Tong et al.'s approach and AMD. With an accuracy of 99.64%, a recall equals to 99.78% and a specificity of 99.89%, the best values reached in the second test with $S^m_{min} = 90$ and $S^b_{min} = 80$, AMD achieved better results than those achieved by the other approach. The improvements made by AMD in terms of malware detection and in comparison to Tong et al.'s approach are also shown in Table 10.

Table 10: Our AMD approach's improvements compared to Tong et al.'s approach

Measure	Tong et al.'s approach	AMD approach	Improvement compared to Tong et al.'s approach
Accuracy (%)	90.19	99.75	9.56
Recall (%)	91.66	99.65	7.99
Specificity (%)	88.88	99.84	10.96

Table 11 shows the comparison results in terms of FP and FN rates for both Kayacık et al.'s approach and AMD. With an FP rate of 0.11% and an FN rate of 00.65%, the best values reached for the imbalanced data set with $S^m_{min} = 90$ and $S^b_{min} = 90$, AMD achieved better results than those achieved by the other approach. In [26], the authors used an *Anomaly rate*, which is a combined value of false positive and false negative rates, to evaluate their work.

Table 11: Our AMD approach's results compared to Kayacık et al.'s approach

Measure	Kayacık et al.'s approach	AMD approach
False Positive (%)	—	0.11
False negative (%)	—	00.65
Anomaly rate (%)	2.70	—

—: No registered value

The improvements made by the AMD approach, from Table 10 and Table 11, show the importance of setting up a detection system that will be as independent as possible from the base of examples while at the same time taking into account the rapid evolution of malware. Furthermore, the more independent the detection model is from the base of examples, the more we ensure that the detection system will be effective in detecting different variants of malware. These results show that our AMD approach outperforms not only the antivirus engines but also the state-of-the-art methods by offering a powerful malware detection system based on the use of the genetic evolutionary algorithm.

6 Conclusion

In this paper, a new approach for Android malware detection named Artificial Malware-based Detection approach (AMD) was developed. AMD is based on the use of an evolutionary algorithm to generate artificial patterns able to detect malware. There are three major components in the proposed AMD approach, namely the API call sequences extraction, the patterns construction and the classifier. In the extraction component, each Android app is unpacked into a readable file (executable) and from which API call sequences are extracted. Then, API call sequences are formatted as item vectors. Besides, frequent sets of item vectors (frequent item sets or patterns) are extracted and selected in order to build two sets of patterns: a malicious pattern set and a benign pattern set. To keep fairly varied sets of patterns, malicious patterns are gathered using a genetic algorithm. The generated patterns are then injected into the set of the selected malicious patterns. Finally, a detection model was developed and evaluated using different performance metrics. AMD was compared to several antivirus engines in addition to Tong et al.'s approach [47] that is dedicated to the detection of malware in the Android operating system and to Kayacık et al.'s approach [26] that uses a GA to evolve mimcry attack to evaluate existing evasion attack detection systems. AMD shows promising results. Through generating both the malware pattern

set and the benign pattern set, AMD outperforms the state-of-the-art well-known detection approaches with better detection accuracy rates. Its detection accuracy rate exceeds 98%. Moreover, by analyzing the false positive and false negative values, AMD was able to achieve better results when including the artificially generated patterns: with $FP = 00.21\%$ and $FN = 00.41\%$ for the balanced data set, in which 14 775 684 malicious patterns and 10 172 203 benign patterns were analyzed, and $FP = 00.28\%$ and $FN = 00.44\%$ for the imbalanced data set in which 27 534 880 malicious patterns and 10 172 203 benign patterns were analyzed. Based on the conducted evaluations and the promising obtained results, our AMD approach can be considered as an interesting malware detection approach that is indeed able to detect obfuscated malware thanks to the use of the artificially generated patterns.

Future research will be conducted to investigate the validity of the possible threats. In the current experimental study, the generated malicious patterns are considered to have the same level of confidence as the real ones. By doing so and without considering any semantic evaluation of the generated artificial patterns present a limit for future investigations. In fact, a genetic algorithm that generates the artificial malicious patterns without any semantic evaluation will eventually consider some benign patterns as malicious ones, and vice versa. To alleviate the effects of such behaviour, we have already taken into account that the generated patterns would be as different as possible from the benign ones provided by the base of examples. Indeed, we made sure to maximize both similarity to malware patterns and dissimilarity to benign ones in the fitness function of the used GA. But still, it seems necessary to think of putting a mechanism that should be able to evaluate the generated patterns and suppresses those that are deemed inappropriate, i.e., they have a shady structure or an unrealizable behavior. In this concern, we aim to assign a weighting factor to each generated sequence that will depend on the semantics of the generated structure and to which extent it can be considered as a real malicious one. Another perspective would be the use of other metaheuristics search mechanism for item set extraction. As most population-based metaheuristics could be seen as modified versions of GA [45], the Ant Colony Optimization (ACO) is probably the only population-based algorithm that has a different search behavior that is based on the probabilistic construction of fit solutions and pheromone update. A comparison between GA and ACO within the framework of our approach could give insights about the best search mechanism for malware patterns construction. Finally, a very interesting future research path is the consideration of the adversarial learning problem [8] where the data sets contain adversarial examples that may fool or misguide the classifier. In this work, we have partially and implicitly considered this problematic as the used data sets enclose many obfuscated examples. However, it is very important to enrich our AMD approach with specific adversarial techniques. On the one hand, adversarial training [18] could be adopted where adversarial examples are injected to the model and labeled as threats. On the other hand, a defensive distillation mechanism [44] could be developed. Such mechanism aims to make the classifier model more flexible by having one model predict the outputs of another model that was trained earlier. In such settings, our approach could be hybridized with generative adversarial neural networks [20]. Based on this, it would be interesting to investigate the performance of the AMD approach when taking into account more characteristics concerning malware behaviors and with the ultimate goal of improving the detection performance.

References

1. Adebayo, O.S., AbdulAziz, N.: Android malware classification using static code analysis and apriori algorithm improved with particle swarm optimization. In: 2014 4th World Congress on Information and Communication Technologies (WICT 2014). pp. 123–128. IEEE (2014)
2. Agrawal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: Proc. 20th int. conf. very large data bases, VLDB. vol. 1215, pp. 487–499 (1994)
3. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 217–228. ACM (2012)
4. Aydogan, E., Sen, S.: Automatic generation of mobile malwares using genetic programming. In: European conference on the applications of evolutionary computation. pp. 745–756. Springer (2015)
5. Bacardit, J.: Analysis of the initialization stage of a pittsburgh approach learning classifier system. In: Proceedings of the 7th annual conference on Genetic and evolutionary computation. pp. 1843–1850. ACM (2005)
6. Bernadó-Mansilla, E., Garrell-Guiu, J.M.: Accuracy-based learning classifier systems: models, analysis and applications to classification tasks. Evolutionary computation **11**(3), 209–238 (2003)
7. Bhattacharjya, R.K.: Introduction to genetic algorithms. IIT Guwahati **12** (2012)
8. Biggio, B., Roli, F.: Wild patterns: Ten years after the rise of adversarial machine learning. Pattern Recognition **84**, 317–331 (2018)
9. Bradley, A.P.: The use of the area under the roc curve in the evaluation of machine learning algorithms. Pattern recognition **30**(7), 1145–1159 (1997)
10. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. pp. 15–26. ACM (2011)
11. Chaba, S., Kumar, R., Pant, R., Dave, M.: Malware detection approach for android systems using system call logs. arXiv preprint arXiv:1709.08805 (2017)
12. Chang, C.C., Lin, C.J.: Libsvm: a library for support vector machines. ACM transactions on intelligent systems and technology (TIST) **2**(3), 27 (2011)
13. Davis, L.: Handbook of genetic algorithms (1991)
14. Di Cerbo, F., Girardello, A., Michahelles, F., Voronkova, S.: Detection of malicious applications on android os. In: International Workshop on Computational Forensics. pp. 138–149. Springer (2010)
15. Edge, K.S., Lamont, G.B., Raines, R.A.: A retrovirus inspired algorithm for virus detection & optimization. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation. pp. 103–110. ACM (2006)
16. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 627–638. ACM (2011)
17. Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X.: Synthesizing near-optimal malware specifications from suspicious behaviors. In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 45–60. IEEE (2010)
18. Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., Marchand, M., Lempitsky, V.: Domain-adversarial training of neural networks. The Journal of Machine Learning Research **17**(1), 2096–2030 (2016)
19. Gonzblez, A., Pérez, R.: Slave: A genetic learning system based on an iterative approach. IEEE Transactions on Fuzzy Systems **7**(2), 176–191 (1999)

20. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Advances in neural information processing systems. pp. 2672–2680 (2014)
21. Griffin, K., Schneider, S., Hu, X., Chiueh, T.C.: Automatic generation of string signatures for malware detection. In: International workshop on recent advances in intrusion detection. pp. 101–120. Springer (2009)
22. https://www.pnfsoftware.com/: Jeb ([Accessed on 2017])
23. http://www.javadecompilers.com/jad: Jad ([Accessed on 2017])
24. Jang, J.w., Yun, J., Mohaisen, A., Woo, J., Kim, H.K.: Detecting and classifying method based on similarity matching of android malware behavior with profile. SpringerPlus **5**(1), 273 (2016)
25. Karbalaie, F., Sami, A., Ahmadi, M.: Semantic malware detection by deploying graph mining. International Journal of Computer Science Issues **9**(1), 373–379 (2012)
26. Kayacık, H.G., Zincir-Heywood, A.N., Heywood, M.I.: Can a good offense be a good defense? vulnerability testing of anomaly detectors through an artificial arms race. Applied Soft Computing **11**(7), 4366–4383 (2011)
27. Kim, K., Moon, B.R.: Malware detection based on dependency graph using hybrid genetic algorithm. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation. pp. 1211–1218. ACM (2010)
28. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM conference on Computer and communications security. pp. 290–299. ACM (2003)
29. Miller, B.L., Goldberg, D.E., et al.: Genetic algorithms, tournament selection, and the effects of noise. Complex systems **9**(3), 193–212 (1995)
30. Nanni, L., Lumini, A.: Generalized needleman–wunsch algorithm for the recognition of t-cell epitopes. Expert Systems with Applications **35**(3), 1463–1467 (2008)
31. Narouei, M., Ahmadi, M., Giacinto, G., Takabi, H., Sami, A.: Dllminer: structural mining for malware detection. Security and Communication Networks **8**(18), 3311–3322 (2015)
32. Nissim, N., Moskovitch, R., Rokach, L., Elovici, Y.: Novel active learning methods for enhanced pc malware detection in windows os. Expert Systems with Applications **41**(13), 5843–5857 (2014)
33. Noreen, S., Murtaza, S., Shafiq, M.Z., Farooq, M.: Evolvable malware. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. pp. 1569–1576. ACM (2009)
34. Palahan, S., Babić, D., Chaudhuri, S., Kifer, D.: Extraction of statistically significant malware behaviors. In: Proceedings of the 29th Annual Computer Security Applications Conference. pp. 69–78. ACM (2013)
35. Palumbo, P., Sayfullina, L., Komashinskiy, D., Eirola, E., Karhunen, J.: A pragmatic android malware detection procedure. Computers & Security **70**, 689–701 (2017)
36. Perdisci, R., Ariu, D., Giacinto, G.: Scalable fine-grained behavioral clustering of http-based malware. Computer Networks **57**(2), 487–500 (2013)
37. Perdisci, R., Lee, W., Feamster, N.: Behavioral clustering of http-based malware and signature generation using malicious network traces. In: NSDI. vol. 10, p. 14 (2010)
38. Rafique, M.Z., Alrayes, N., Khan, M.K.: Application of evolutionary algorithms in detecting sms spam at access layer. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation. pp. 1787–1794. ACM (2011)

39. Rashidi, B., Fung, C.: Xdroid: An android permission control using hidden markov chain and online learning. In: Communications and Network Security (CNS), 2016 IEEE Conference on. pp. 46–54. IEEE (2016)
40. https://www.hex rays.com/products/ida/: Ida-pro disassembler and debugger ([Accessed on 2017])
41. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 108–125. Springer (2008)
42. Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A.: Malware detection based on mining api calls. In: Proceedings of the 2010 ACM symposium on applied computing. pp. 1020–1025. ACM (2010)
43. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences **231**, 64–82 (2013)
44. Soll, M., Hinz, T., Magg, S., Wermter, S.: Evaluating defensive distillation for defending text processing neural networks against adversarial examples. In: International Conference on Artificial Neural Networks. pp. 685–696. Springer (2019)
45. Sörensen, K.: Metaheuristicsthe metaphor exposed. International Transactions in Operational Research **22**(1), 3–18 (2015)
46. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobilesandbox: having a deeper look into android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1808–1815. ACM (2013)
47. Tong, F., Yan, Z.: A hybrid approach of mobile malware detection in android. Journal of Parallel and Distributed Computing **103**, 22–31 (2017)
48. Umbarkar, A., Sheth, P.: Crossover operators in genetic algorithms: A review. ICTACT journal on soft computing **6**(1) (2015)
49. https://www.microsoft.com/en us/download/details.aspx?id=8002: Windows xp mode ([Accessed on 2017])
50. Vidas, T., Christin, N.: Sweetening android lemon markets: measuring and combating malware in application marketplaces. In: Proceedings of the third ACM conference on Data and application security and privacy. pp. 197–208. ACM (2013)
51. Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 252–276. Springer (2017)
52. Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., Platzer, C.: Andrubis: Android malware under the magnifying glass. Vienna University of Technology, Tech. Rep. TR-ISECLAB-0414-001 (2014)
53. Wen, L., Yu, H.: An android malware detection system based on machine learning. In: AIP Conference Proceedings. vol. 1864, p. 020136. AIP Publishing (2017)
54. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Security & Privacy **5**(2) (2007)
55. Wilson, S.W., Wilson, S., Xcs, G., et al.: Generalization in the xcs classifier system (1998)
56. Yusoff, M.N., Jantan, A.: A framework for optimizing malware classification by using genetic algorithm. In: International Conference on Software Engineering and Computer Systems. pp. 58–72. Springer (2011)
57. Zheng, M., Sun, M., Lui, J.C.: Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on. pp. 163–171. IEEE (2013)

58. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 95–109. IEEE (2012)
59. Zolkipli, M.F., Jantan, A.: A framework for malware detection using combination technique and signature generation. In: 2010 Second International Conference on Computer Research and Development. pp. 196–199. IEEE (2010)

A A comparison between malware analysis techniques

In this section, some background information about the different types of malware detection techniques is given. Three major approaches of malware detection which are proposed in literature are revised; namely the static, the dynamic and the hybrid analysis techniques. Also, a detailed comparative study of these techniques is performed based on their main characteristics. Static analysis is the process of analyzing the code segments without actually running the application. The main characteristics of these techniques are their efficiencies in terms of low resource consumption and low time computation; specifically, when compared to dynamic methods. However, due to the existence of various intrusion detection techniques such as code packing, anti-debugging, control-flow and entry point obfuscation, static analysis can be in some cases inefficient [28]. These limitations led to introduce a dynamic detection system that deals with these issues. Compared to static techniques, dynamic techniques analyze the code during runtime and this can immunize the analysis process from many obfuscation techniques and even self-modifying programs. Meanwhile, hybrid techniques combine aspects of both static and dynamic analysis to further construct control and data flow analysis. In Table 12, a comparative study of different analysis techniques is performed. In this table, the second column "Rep" (i.e., Representation) shows the type of features used by the systems to represent the input files. The third column shows the programs or the algorithms used to extract these features. The fourth column cites the classifiers used in the corresponding method. The "Platform" column shows the used operating system. The "DR" (i.e., Detection Rate) column gives the detection rate of the related system where NR means that authors did not perform an evaluation on malware data sets or did not mention this in their paper. The "FP" (i.e., False Positive) column shows the false positive rate of the system where NR means that authors did not present an evaluation on benign data sets or did not mention this in their paper. The "Freq pat" (i.e., Frequent patterns) column shows if the system uses frequent patterns such as frequent sub-graphs, item-sets or subsequences as features for extracting behaviors and then uses these for detection. The "weakness" column shows the challenge of each method for detecting malwares and mainly what are the possible evasion techniques against them. In the third, fourth and eighth columns, NR means that the information was not given in the paper.

Table 12: A comparison between malware analysis techniques (acronyms are explained under the table)

Ref	**Rep**	**Extraction method**	**Classifiers**	**Platform**	**DR (%)**	**FP (%)**	**Freq pat**	**Weakness**
Static analysis								
[21]	BYT SEQ	Kephart and Arnolds extraction work	NR	Windows	NR	<0.1	NR	Substitution, reordering, injection

[27]	DG SRC	NR	NR	Windows	88.9	NR	NR	Based on SRC
[42]	API	PE analyzer	RF	Windows	99.7	1.5	ITM	Injection
[14]	PER	Apriori algorithm	NR	Android	NR	NR	ITM	Injection
[43]	OP SEQ	NewBasic Assembler	DT, SVM, KNN, BN	Windows	95.8	2	NR	Substitution, reordering, injection
[32]	BYT NG	NG length sliding windows	SVM	Windows	85	0.8	NR	Substitution, reordering, injection
[31]	PE, DLL, API, Dependency tree	Dependency Walker	RF	Windows	100	0.03	Freq subtree	Mimicry attacks, Pruning tree
[33]	SIG	IDA pro[40]	Commercial antiviruses	Windows	98.98	1.02	NR	"Bagle" attacks only
[15]	SIG	XNR	Pattern matching	Windows	NR	NR	X	Encryption, obfuscation
[59]	SIG	NR	Pattern matching	Windows	NR	NR	NR	virus, worms and Trojan horse
[4]	Sys call	JAD[23], JEB[22]	Detectors**	Android	NR	NR	NR	Mimicry attack
Dynamic analysis								
[17]	API GRA	HOLMES	X	Windows	86	0	GRA	Graph complexity
[10]	SYS call	Strace	K-means	Android	100	NR	NR	Manual check,Sniffs attack
[25]	API GRA	gSpan	RF	Windows	96.6	3.4	GRA	Graph complexity
[36]	HTTP	Behavioral clustering[37]	Behavioral clustering [37]	Windows	68	0	HTTP	Specific malware
[34]	API GRA	Subgraph mining algorithm	Linear classifier	Windows	86.77	0	GRA	Graph complexity
[56]	Extracted features*	Windows XP Mode [49]	NB, SVM, DT, KNN	Windows	NR	NR	NR	Worms and Torjan horses only
Hybrid analysis								
[58]	PER, Behavioral footprints	DroidRanger	NR	Android	NR	FN: 5.04	NR	Only permission based filtering
[46]	PER API	Android SDK	NR	Android	NR	NR	NR	Analyzing time
[57]	PER API SEQ SIG Generator	AIS parser	SIM measurement	Linux	NR	NR	NR	Obfuscation
[50]	API	Android SDK	NR	Android	NR	FN: 4.2	NR	Only Pi calculation measurement

[52]	BYT PER HTTP DNS FTP SMTP IRC	Stowaway[16] PScout [3]	Clustering	Android	NR	NR	NR	Evasion
[24]	Behavior profile PE SN API SEQ PER	Android SDK	Classification engine, SIM metrics	Android	73.18	0.32	NR	Packing, Binary code encryption
[47]	API SEQ	Sys calls extraction	Pattern matching	Emulated environment	90.19	8.12	API SEQ	New malwares

PER: permissions; HED: executable header; SYS: system; API: application programming interface; OP: operation code; DG: dependency graph; CFG: control flow graph; SRC: source code; NG: n-gram; MET: metadata; BYT: byte code; SEQ: sequence; GRA: graph; ITM: item-set, MD: mobile device; BYT TEMplate: instruction sequences where variables and symbolic constants are used; CBA: Classification based on association; OOA MT: objective oriented association mining technique; SVM: support vector machines; IBL: instance based learner; NB: naive bayes; KE based on EL: knowledge extraction based on evolutionary learning; SN: serial number; SIM: similarity; DT: decision tree; BN: bayesian networks; RF: random forest; gSpan: graph-based substructure patternmining; NR: not reported. *: malware sample, MD5 hash, malware size, malware specific target,class operation. **: Stide, Process Homeostasis, Process Homeostasis with as chemamask, The Markov Model-based detector, Auto-associative neural network.

B Data preprocessing

In this section, the preprocessing of the gathered Android files is discussed. Indeed, this step is required in the first phase of the AMD approach (Figure 1b) to enable extracting the API call sequences from the collected data (apps).

Malicious files, which are Android applications, include all varieties of malwares that can be in the form of a virus, a backdoor, a worm, a trojan, a spyware or any other possible form. These malicious Android files are gathered from both AMD set and DroidCat malicious data sets.

Each Android application is in the form of a single file known as an Android Application Package (APK) that includes all of the applications code. The structure of an APK is shown in Figure 8 where it includes:

- Manifest (AndroidManifest.xml): is an Android manifest file, describing the name, the version, the access rights and the referenced library files for the application;
- META-INF (signatures): is a folder containing the MANIFEST.MF file, which stores meta data and also the signature of an APK;
- Assets: is a directory containing applications assets;
- Compiled resources (ressources.arsc): this file contains precompiled resources, in binary XML;
- Native libraries ($lib\backslash$): is a directory containing the compiled code i.e., native code libraries.
- Dalvik bytecode (classes.dex): contains the classes compiled in the dex file format; and
- Ressources (res/): contains resources not compiled into resources.arsc;

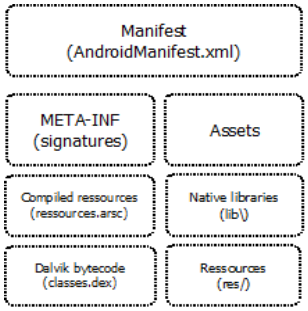


Fig. 8: The structure of an APK

To extract the API calls, a compilation of the Dalvik Executable (.dex) is required. This process will make use of several needed functions located in different libraries (.so) such as in the native library (*lib*\). As malicious files are dealt with, and in order to capture the interactions of the application with the operating system, the Android Studio's emulator is used allowing to run files in a controlled environment. Under this environment, to analyze Android apps, first, the APK is unzipped to get the .dex file. Formally, in a .dex file, five different methods can be detected to invoke an API call. These are presented in Table 13. To gather these API calls, the Strace module is used.

Table 13: Methods to invoke API calls

Method	Role
invoke-static	invokes a static method with parameters
invoke-virtual	invokes a virtual method with parameters
invoke-direct	invokes a method with parameters without the virtual method resolution
invoke-super	invokes the virtual method of the immediate parent class
invoke-interface	invokes an interface method

C Performance indicators

The performance of AMD is evaluated in terms of precision, recall, specificity, F1 score, Area Under the Receiver Operating Characteristics (ROC) Curve (AUC) and accuracy. These are defined in this section.

– Precision, also referred to as positive predictive value (PPV): It is defined as the number of true positives divided by the number of true positives plus the number of false positives as shown in Equation 13.

$$Precision(orPPV) = \frac{TN}{TN + FP} \tag{13}$$

- Recall, also referred to as sensitivity or as the true positive rate (TPR) measure: It is able to determine positive instances (i.e., malware files) correctly, as shown in Equation 14.

$$Recall(orSensitivityorTPR) = \frac{TP}{TP + FN} \tag{14}$$

To be able to draw our ROC curves of the AMD approach, in Section 5.1, both the recall and specificity values are needed. In fact, the ROC curve is created by plotting the true positive rate (TPR) against the (1-specificity) values at various thresholds of TN, TP, FP and FN. To do so, the definition of the specificity is added:

- Specificity or the true negative rate (TNR): It determines negative instances (i.e., benign files) correctly identified (Equation 15).

$$TNR(orSpecificity) = \frac{TN}{FP + TN} \tag{15}$$

- Accuracy: measures the number of correctly classified instances, either positive or negative, divided by the entire number of instances as shown in Equation 16.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{16}$$

- F1_score, also referred to as F-score or F-measure: Is a measure of a test's accuracy (predicting a pattern's class). It considers both the precision and the recall. The F1_score is the harmonic average of the precision and recall, where an F1_score reaches its best value at 1 (perfect precision and recall) and worst at 0. The F1_score formula is given in Equation 17:

$$F1_score = 2 * \frac{precision * recall}{precision + recall} \tag{17}$$

- The area under a ROC curve (AUC): quantifies the overall ability of the test, the prediction of a pattern's nature in our case, to discriminate between positive instances and negative ones[18]. A useless pattern (unable on identifying true positives) has an area of 0.5. A perfect test (one that has zero false positives and zero false negatives) has an area of 1.
The simplest way to calculate the AUC is to use trapezoidal integration [9]. It consists on a non-parametric method based on constructing trapezoids under the curve as an approximation of area. In fact, we need to use the points, referred to as i in the below equations (Equation 18, Equation 19 and Equation 20), on the ROC curve. To do so, we suppose that the $FPR = \alpha$ and the $TPR = 1$ - β.

$$AUC = \sum_i \{(1 - \beta_i) * \Delta\alpha + \frac{1}{2}[\Delta(1 - \beta) * \Delta\alpha]\} \tag{18}$$

where

$$\Delta(1 - \beta) = (1 - \beta_i) - (1 - \beta_{i-1}) \tag{19}$$

$$\Delta\alpha = \alpha_i - \alpha_{i-1} \tag{20}$$

A rough guide for classifying the accuracy of a detection test is the academic point system given in [9] and is presented in Table 14. For instance, if the area equals 0.84 this means that the pattern could be considered as "good" at separating positive and negative instances.

[18] `http://gim.unmc.edu/dxtests/roc3.htm`

Table 14: Classification of the detection test

AUC's value	Evaluation
0.90 - 1	Excellent
0.80 - 0.90	Good
0.70 - 0.80	Fair
0.60 - 0.70	Poor
0.50 - 0.60	Fail