Android Malware Detection as a Bi-level Problem

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

Android Malware Detection as a Bi-level Problem

Manel Jerbi[1], Zaineb Chelly Dagdia[2,3], Slim Bechikh[1], 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 Paris-Saclay, UVSQ, DAVID, France
[3] LARODEC, Institut Supérieur de Gestion de Tunis, Tunis, Tunisia,
zaineb.chelly-dagdia@uvsq.fr

Abstract. Malware detection is still a very challenging topic in the cybersecurity field. This is mainly due to the use of obfuscation techniques. To solve this issue, researchers proposed to extract frequent API (Application Programming Interface) call sequences and then use them as behavior indicators. Several methods aiming at generating malware detection rules have been proposed with the goal to come up with a set of rules that is able to accurately detect malicious code patterns. However, the rules generation process heavily depends on the training database content which will affect the detection rate of the model when confronted to new variants of malicious patterns. In order to assess a rule's detection accuracy, we need to execute the rule on the whole malware database which makes the detection rule quality evaluation very sensitive to the database content. To solve this issue, we suggest in this paper to consider the detection rules generation process as a BLOP (Bi-Level Optimization Problem), where a lower-level optimization task is embedded within the upper-level one. The goal of the upper-level is to generate a set of detection rules in the form of: trees of combined patterns. Those rules are able to detect not only the real patterns from the base of examples but also the artificial patterns generated by the lower-level. The lower-level aims to generate a set of artificial malicious patterns that escape the rules of the upper-level. An efficient co-evolutionary algorithm is adopted as a search engine to ensure optimization at both levels. Such an automated competition between the two levels makes our new method BMD (Bi-level Malware Detection) able to produce effective detection rules that are capable of detecting new predictable malicious behaviors in addition to existing ones. Based on the statistical analysis of the experimental results, our BMD method has shown its merits when compared to several relevant state-of-the-art malware detection techniques on different Android malware datasets.

Keywords: Android malware detection · Bi-level optimization · Detection rules generation · Artificial malicious patterns · Evolutionary algorithms

1 Introduction

Malware, or malicious software, is any program or file that is harmful to a computer system. These malicious programs can perform a variety of functions including stealing, encrypting, deleting sensitive data, altering or hijacking core computing functions and even monitoring the users' computer activities without their permission[4]. The use of packers and obfuscation techniques have further empowered the malware coders to redevelop malware variants. This requires the use of effective techniques to detect malware. Indeed, several malware detection methods have been proposed in literature and these can be classified as static, dynamic and hybrid approaches; based on how the code is analyzed. While the static approaches try to identify the malicious code without any execution, the dynamic approaches analyse the code during runtime. Hybrid approaches use static and dynamic features as they combine both static and dynamic approaches. Most of the existing approaches in literature derive from the static analysis and this is due to the limitations in power consumption of the majority of mobile devices. However, the static approaches are inefficient against some obfuscation techniques and new attacks. Also, hybrid approaches may have limitations related to both static and dynamic aspects. Therefore, in recent years, researchers have focused on improving the detection techniques in order to be effective not only against known attacks but also against unknown attacks relying on several research fields and areas. Among these are approaches which create new malware and variants of known malware using mainly Evolutionary Algorithms (EA) which enable them to mimic mobile malware evolution [44] [26] [30] [11] [37] [20]. Despite the good performance of these methods, in these research papers, the weaknesses of the proposed detection tools were indeed highlighted when it comes to creating new attacks. The need for new detection techniques which should be more suited to mobile devices was also emphasized. Additionally, it is important to mention that in literature, most of the approaches that create new attacks are either not fully automated or are only proposed for a specific attack type.

In this paper, we propose a new effective and robust malware detection method named "Bi-level Malware Detection" (BMD). BMD improves its detection rate thanks to the evaluation of each detection rule based on the generation of more evasive artificial malware patterns and in recognizing new variants of existing and even unseen malware patterns. Let us precise that the generation of artificial malware behaviors can be considered as a great alternative to face one of the most inconvenient problems to researchers in different fields which is the problem of data availability. In fact, in order to assess a given method, benchmarks with limited size are available. To face such shortcoming, we propose to generate artificial malicious patterns that come to enrich the used base of examples. Also, let us mention that in our previous work [19], a dynamic detection method, named Artificial Malware-based Detection (AMD), is proposed to solve the obfuscation issue. AMD makes use of not only extracted malware patterns

[4] https://searchsecurity.techtarget.com/definition/malware

(i.e., extracted frequent API call sequences also called behaviors) but also artificially generated ones. The artificial malware patterns are generated using a Genetic Algorithm (GA). 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. In AMD, an executable is classified as malware or benignware based on its similarity to the benign patterns, to the real available malicious patterns, and to the artificially generated ones. More precisely, when AMD receives a new executable program P for analysis, it first extracts its API call sequences and then computes the support of all these sequences. Afterwards, two metrics are calculated: (1) the malicious credence value of all malicious patterns $CM(P)$ and (2) the benign credence value of all benign patterns $CB(P)$. If the former is strictly greater than the latter, then P is classified as malicious; otherwise, it is judged to be benign. Eventually, if P is labeled as malicious, it will be executed on a sandbox system and then analyzed in a dynamic way. Regarding the artificial patterns, these are generated based on the bases of examples of malicious and benign patterns using an EA.

It seems that it is of primordial importance to mention that our previously proposed method AMD presents some deficiencies that occur in three main aspects:

1. AMD uses a static detection rule that is defined in an apriori manner based on similarity measures to benign and malware patterns.
2. The detection rule is prespecified independently of the generated malware patterns; and thus the rule definition and the classification tasks are done separately without any interaction.
3. In AMD, the artificial malware patterns are generated in a global ad-hoc manner based on their similarities to real malware and benign patterns from the bases of examples, which may increase the number of false positives, as some generated artificial patterns could be benign.

To overcome the limitations of AMD, BMD is proposed in this paper. In fact, based on Figure 1, unlike AMD in which the classification of a new app relies on a pattern matching process where two similarity metrics are used to determine its nature, our BMD approach relies on the use of detection rules. An app is classified based on the nature of its extracted patterns. Indeed, each gathered pattern is analyzed using each of the BMD rules that are generated using an evolutionary optimization process. If one of these rules returns true, the pattern (and hence the app) is classified as malicious.

Besides, to prevent the detection process from depending, heavily, on the base of examples, AMD [19] diversifies the base of examples with new artificial malicious patterns using a GA [12]. The malicious patterns base remains independent from the malware detection task. Similarly to AMD, BMD uses artificial patterns, but its main distinction and originality rely on the modeling of the malware detection task as a bi-level optimization process. In BMD, the upper-level generates a set of detection rules, each encoded as a tree of combined patterns

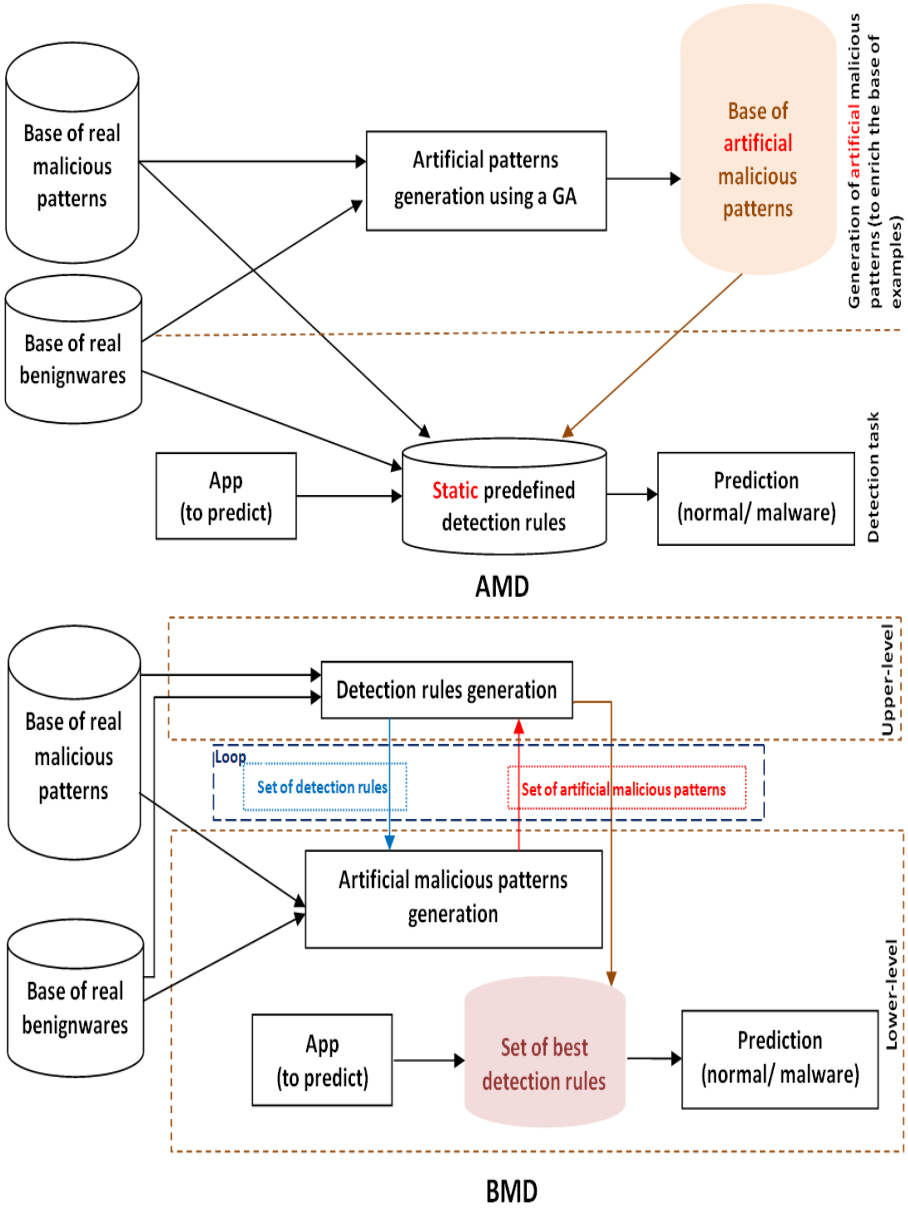

Fig. 1: Main differences between BMD and AMD [19].

(each defined as a set of frequent API call sequences), where the quality optimization of each one of them consists in maximizing the coverage of not only the patterns extracted from the base of examples but also the artificial malicious ones generated by the lower-level. In this study, the malware detection task is framed as a bi-level optimization problem in which the upper-level role is to design a set of effective malware detection rules, while the lower-level one is to generate a set of challenging artificial malware patterns for each rule. In this way, in our bi-level approach, the upper population and the lower one are not co-evolved in parallel (without hierarchy), which avoids the issue that one population converges before the other. Moreover, in such formulation, both populations depend on each other. Finally, another distinction of our BMD approach in comparison to AMD [19] corresponds to the hierarchical quality evaluation and optimization of each upper-level detection rule using not only malware patterns from the base of examples but also artificial ones designed by the lower-level search. Such hierarchical process creates a competition between both levels' populations and thereby allows optimizing both rules and artificial patterns. This competition may significantly reduce the number of false positives, which is not the case of AMD.

To sum up, the BMD evolutionary process is composed of a number of generations each ensuring a single competition between each generated upper-level detection rule and its related lower-level artificial malicious patterns. In this way, a sequence of competition rounds are performed over the search process, which allows designing at the same time a set of effective rules with maximized detection rates and a set of optimized artificial patterns with maximized evasion rates with respect to their corresponding rule. These rounds strengthen the ability of BMD in minimizing the number of false artificial malicious patterns. The main contributions of this paper are:

1. Modeling the malware detection rule generation process as a BLOP, where the upper-level generates a set of detection rules maximizing the coverage of not only the patterns extracted from the base of examples but also the artificial malicious ones generated by the lower-level.
2. The evolutionary optimization at both levels using an efficient EA called CODBA [10], which makes use of decomposition, co-evolution, and multi-threading to reduce as much as possible the computational cost of bi-level optimization.
3. The demonstration of the benefits of the bi-level competition between both levels since for every detection rule, there exists a whole search space of possible artificially generated malicious patterns that should be effectively sampled to come up with fit and challenging artificial patterns that positively affect the detection quality of the corresponding upper-level rule.
4. The evaluation of the outperformance of our BMD approach compared to several state-of-the-art detection methods in terms of accuracy maximization and false alarms minimization.

The rest of this paper is organized as follows: Section 2 presents a detailed description of the related work. Section 3 describes our proposed approach. The

experimental setup and the results of the performance analysis are given in Section 4. The conclusion is given in Section 5.

2 Related work

In this section, some background information is provided about the different types of malware detection techniques which are based on the use of evolutionary algorithms aiming either to optimize the malware classification task or to perform pattern generation.

2.1 Classification-based evolutionary detection methods

Among the works that tried to address the classification problem, we cite the work of [43] where a framework for Android malware application detection using machine learning techniques was proposed. It extracts permission features from several downloaded applications from Android markets. In this work, a classifier named MSGP Malware System (MSGP-MS) was developed by combining GA and Particle Swarm Optimization (PSO) with Random Forest (RF) to classify Android APK files. Yusoff et al. [49] proposed a framework that optimizes the classification problem addressed in [43] by using a GA. Authors proposed the use of Decision Trees (DT) and a GA to enhance and optimize the performance of a new classifier. The results of this work provided a solid setting designed to improve, as much as possible, the classification of harmful programs or apps, specifically worms and Trojan horses, targeting Windows operating systems. The work of D'Angelo et al. [13] performs the malware classification using a recurring subsequences alignment-based algorithm that exploits associative rules. More precisely, authors in [13] use Markov chains to model: (i) the set of states which are API calls and (ii) the edge between two states which reflect the probability of transitioning from two API invocations (the probability of a given state (API) to be invoked by another state). In [1], authors proposed MalFamAware, an approach to malware family identification based on an online clustering algorithm, namely BIRCH, which updates clusters as new samples are fed without requiring to re-scan the entire dataset. MalFamAware is able to classify new malware in existing families and identify new families at runtime. Firdaus et al. [16] started by extracting relevant features, named strings, which consist of permissions, words in double quotes, functions, intents, Linux commands, directory paths, and system commands. This was achieved by applying an evolutionary algorithm to search for optimal and relevant features in multiple categories. Then, authors applied the Genetic Search (GS), based on a GA, process to select the best features (i.e., minimum number of features in multiple categories) among all the extracted strings obtained in the string identification phase. The final phase involved a machine learning classifier which trained the information in the dataset to construct a detection model which predicts an application to be either benign or malware. In [9] , authors proposed an approach for malware detection

and phylogeny studying based on dynamic analysis using process mining. The approach exploits process mining techniques to identify relationships and recurring execution patterns in the system call traces gathered from a mobile application in order to characterize its behavior. In another study [4], authors used permissions and API calls to discriminate between the malware and goodware applications. For this purpose, two features ranking algorithms, Information Gain (IG) and Pearson CorrCoef (PC), were used to rank the individual permissions and API calls based on their importance for classification. In addition, authors proposed a hybrid method for Android malware detection based on the combination of the adaptive neural fuzzy Inference System (ANFIS) with PSO. The PSO was utilized to optimize the ANFIS parameters by tuning its membership functions to generate more precise fuzzy rules for Android apps classification. Also, in [50], authors proposed a method to detect Android malware based on the combination of multiple types of features and a machine learning algorithm; Rotation Forest. The static information considered in this mechanism includes permissions request, monitoring system events, sensitive APIs and permission-rate. In [15], authors proposed a novel approach that constructs frequent subgraphs (fregraphs) to represent the common behaviors of malware samples that belong to the same family. Moreover, authors developed FalDroid, a system that automatically classifies Android malware and selects representative malware samples in accordance with fregraphs. In [47], a deep learning and machine learning combined model is proposed for malware behavior analysis. One part of it analyzes the dependency relation in API call sequence at the functional level, and extracts features for random forest to learn and classify. The other part employs a bidirectional residual neural network to study the API sequence and discover malware with redundant information preprocessing. The work of Aksu and Aydin [3] proposes an intrusion detection framework based on feature selection and a set of classifiers. Authors propose a meta-heuristic algorithm called modified genetic algorithm (MGA) m-feature selection for dimensionality reduction by selecting optimal feature subset based on k-fold cross validation. Then, they use five different linear and nonlinear classifiers: support vector classifier (SVC), logistic regression classifier (LRC), decision tree classifier (DTC), k-nearest neighbors classifier (KNC), and linear discriminant analysis classifier (LDAC), as candidate classifiers to develop an efficient IDS. Finally, they select the best classifier from the candidates and build an IDS. Also, authors in [18] propose a hybrid optimization and deep-learning-centric IDS to face the IoT-enabled smart cities' intrusion threats. The dataset initially undergoes pre-processing. Then, feature selection and clustering are performed utilizing the Hybrid Chicken Swarm Genetic Algorithm (HCSGA) and MK-means Algorithm. Lastly, the transformed data is loaded to the Deep Learning-based Hybrid Neural Network (DLHNN) classifier, classifying the normal and attack data. In the work of [8], authors proposed a multi-dimensional machine learning approach to predict the Stuxnet malware from a dataset that consists of malware samples by using five distinguishing features of advanced malware. Authors in [8] defined the features by analyzing advanced malware samples in the wild. This approach uses regression

models to predict malware. Authors created a malware dataset from existing datasets that contains real samples for experimental purposes. Authors in [31] proposed a framework named AdDroid, for analyzing and detecting malicious behaviour in Android applications based on various combinations of artefacts called rules. The artefacts represent actions of an Android application such as connecting to the Internet, uploading a file to a remote server or installing another package on the device. AdDroid employs an ensemble-based machine learning technique where Adaboost is combined with traditional classifiers in order to train a model founded on static analysis of Android applications that is capable of recognizing malicious applications. Feature selection and extraction techniques are used to get the most distinguishing rules. Also in [34] authors proposed an autonomous Host-based Intrusion Detection System (HIDS) for Android mobile devices that overcomes the limitation of continuous connectivity to a central server and addresses the risk of data leakage due to the communication of the intrusion detection system with the remote central server. The proposed system is based on dynamic analysis of the device's behaviour characterised by a vector of features and continuously monitors a specific set of features at the device level in order to define the runtime behaviour of the mobile device and applies detection algorithms (i.e., Machine Learning (ML) and statistical algorithms) to classify it as benign or malicious.

2.2 Pattern generation-based evolutionary detection methods

In comparison to the different ways of classification, malware generation tools attempt to create new versions of dangerous programs or apps in order to shield the analysis process from several obfuscation techniques and even mutating programs. About the second category, i.e., the generation of new patterns, in literature, different works proposed new approaches to detect malicious programs without lying on the use of a static base of malware signatures. Among these, we can cite the work proposed in [7] which evolved new malware, specifically new versions of known malware, by using Genetic Programming (GP) in order to figure out the performance of existing static analysis tools. Also, [51] proposed a technique that combines a signature-based technique with a GA. In another work proposed in [14], 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 combined known information from past viruses with a type of prediction for future viruses. Authors generated antibodies (new virus signatures) from antigens (string of known virus signatures). Also, [29] proposed a framework based on the concept of evolution in viruses on a well-known virus family, called "Bagle". In [29], features were extracted based on the assembly code and were evolved using GAs. The generated virus files were afterwards tested using a commercial antivirus. The work of [21] proposed a static anomaly based approach to detect malware. This work focused 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 [21] mainly focused on generating malware, particularly buffer overflow attacks, rather than detecting them. Already developed detectors were used to evaluate the generated attacks. In [45], authors demonstrated how malware can take advantage of the ubiquitous and powerful graphics processing unit (GPU) to increase its robustness against analysis and detection. Authors presented, respectively, the design and implementation of brute-force unpacking and run-time polymorphism, two code armoring techniques based on the general-purpose computing capabilities of modern graphics processors. By running part of the malicious code on a different processor architecture with ample computational power, these techniques pose significant challenges to the existing malware detection and analysis systems, which are tailored to the analysis of CPU code. In [48], authors applied the dynamic code generation and loading techniques to produce malware in order to assess the existing anti-malware tools at runtime. Also, the work of [6] proposed a cloud-based malware detection method. Malicious behavior patterns (system calls with corresponding relationships) are determined. Those behaviors are trained using learning algorithms (learning-based detection module). Afterwards, they are evaluated regarding their repeated frequencies (rule-based detection module). The final list of behaviors is used to predict new apps. In[27], authors developed a framework, MYSTIQUE, to automatically generate malware with specific features covering four attack features (triggers / permissions/ intent filter/ source and sink) and two evasion features (control based evasion / data based evasion), using a GA. In [19], AMD, thoroughly presented in Section 1, was proposed to solve the obfuscation issue. However, it suffers from some limitations which are: (i) the use of static detection rules based on similarity measures, (ii) the rule definition task and the detection task are done separately and (iii) the possible existence of false patterns among the artificial generated patterns, as previously highlighted.

As most of these cited approaches are not fully automated or are only proposed for a specific attack type, in [38], the goal was to investigate the use of co-evolutionary computation techniques on the development of mobile malware and anti-malware as well as to design a fully automated system. In most of these state-of-the-art methods, among many others, a system using co-evolutionary algorithms for malware detection is proposed where a first population generates detection rules, and a second population generates artificial malware. Both populations are executed in parallel without hierarchy. The problem with such co-evolutionary approaches is that one population may converge before the other. Contrariwise, in our proposed BMD bi-level approach, there is a hierarchical evolution process that allows avoiding the problem of premature convergence of one population over the other. Indeed, the evaluation of every detection rule solution (upper-level) requires running a search algorithm to find the least detectable and undetectable malicious patterns for the considered rule. This avoids driving the search towards uninteresting directions. Furthermore, the state-of-the-art co-evolution approaches treat the two populations independently; however, the BMD approach proposes a bi-level modeling using an existing co-evolutionary

algorithm so that a competition is ensured between detection rules and their related artificial malicious patterns. In our BMD approach, the evaluation of solutions in the upper-level depends on the lower-level (both populations cannot be executed in parallel). To the best of our knowledge, this is the first work that applies bi-level optimization to malware detection.

2.3 Bi-level Optimization

Most state-of-the-art optimization problems concern a single level of optimization. However, in practice, several problems are naturally described by two levels. These are called BLOPs [23]. In such problems, we find a nested optimization problem within the constraints of the outer optimization one. The outer optimization task is usually referred to as the upper-level problem or the leader problem. The nested inner optimization task is referred to as the lower-level problem or the follower problem, thereby referring to the bi-level problem as a leader-follower problem or as a Stackelberg game [42]. The follower problem appears as a constraint to the upper-level, such that only an optimal solution to the follower optimization problem is a possible feasible candidate to the leader one (see Figure 2). A BLOP contains two classes of variables: (i) the upper-level variables $x_u \in X_U \subset R^n$, and (ii) the lower-level variables $x_l \in X_L \subset R^m$. For the follower problem, the optimization task is performed with respect to the variables x_l while the variables x_u act as fixed parameters. Thus, each x_u corresponds to a different follower problem, whose optimal solution is a function of X_l and needs to be determined. All variables (x_u, x_l) are considered in the leader problem for given values of x_l (Figure 2). In what follows, we give the formal definition of BLOP. Assuming $L : R^n \times R^m \to R$ to be the leader problem and $f : R^n \times R^m \to R$ to be the follower one, a BLOP could be defined as follows:

$$\underset{x_u \in X_U, x_l \in X_L}{Min} L(x_u, x_l) \text{subject to} \begin{cases} G_k(x_u, x_l) \leq 0, k = 1, \ldots, K. \\ x_l \in ArgMin\{f(x_u, x_l) : \\ g_j(x_u, x_l) \leq 0, j = 1, \ldots, J\} \end{cases} \quad (1)$$

In the given formulation, L represents the upper-level objective function, f represents the lower-level objective function, x_u represents the upper-level decision vector and x_l represents the lower-level decision vector. G_k and g_j represent the inequality constraint functions at the upper and lower levels, respectively.

Existing methods to solve BLOPs could be classified into two main families: (1) classical methods and (2) evolutionary methods. The first family includes among others extreme point-based approaches [41]. The main problem of these methods is that they strongly depend on the mathematical traits of the BLOP. The second family includes metaheuristic algorithms that are mainly Evolutionary Algorithms (EAs). Recently, different EAs proved their efficacy in tackling such types of problems thanks to their immunity against the mathematical features of the problem in addition to their ability to tackle large-size problem instances by delivering acceptable solutions in a reasonable time. Some representative works are the works proposed in [40] [42] [25] [22].

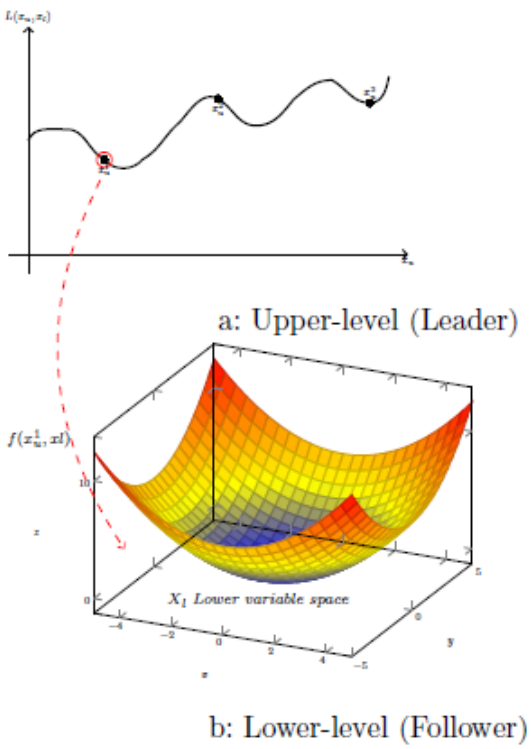

Fig. 2: Illustrating how each upper level solution has its own lower level search space in bi-level optimization. (Inspired by Sinha et al. [42])

3 Proposed approach: Bi-Level Malware Detection

In this section, we give a description of our proposed BMD approach. We first give a general description of the proposed model, then, we detail the approach by describing its different phases.

3.1 General overview and motivation

The majority of the previous proposed techniques accomplish high and quick detection results. However, the greater parts of them are less adapted for real-world requirements for malware detection as they have to be robust against evolving malware. Different requirements for the use of malware detection systems in the real-world need to be considered. One such requirement is that the used approaches should be tested against continuously changing data. An important number of previous works have proposed to extract frequent API call sequences from already-met harmful apps 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 to benign-ware ones.

In this paper, we present our proposed BMD evolutionary based solution which 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 state-of-the-art methods, BMD diversifies the base of examples in an automatic way and detects those new variants of malware. This is achieved via the development of a bi-level optimization technique which relies on the use of an automatic generation task of malicious patterns; using a GA. The leader (upper-level) uses (i) patterns extracted from both the base of examples (input), i.e., set of malicious patterns and (ii) the artificial malicious generated patterns (the red box, dotted line in Figure 1) to produce detection rules. The detection rules generation process consists of creating a combination of patterns used to detect malicious patterns from new files. For example, for a new file P having a set of patterns, we can decide the nature of the extracted patterns by comparing them to our base of association rules: if it matches a rule in the malicious set of rules than P is malicious otherwise it is benign. In the bi-level formulation of BMD, the lower-level problem allows to find new malicious artificial patterns (the red box, dotted line in Figure 1). The evaluation of a detection rule is based on its accuracy using both the base of examples (input) and also the artificial malicious patterns generated by the lower-level problem. We aim to maximize the detection accuracy rate of the rules. The follower (lower-level) uses patterns from the base of examples, i.e., the set of malicious and benign patterns, to generate artificial malicious patterns. A GA is used in order to perform the generation process of artificial malicious patterns that maximizes not only the number of new artificial malicious patterns but also the number of generated malicious patterns that are not detected by the leader (detection rules). The upper-level keeps exchanging solutions with the lower-level, i.e., the upper-level sends detection rules to the lower-level and the lower-level sends the generated artificial malicious patterns to the upper-level, until a stopping criterion is met (e.g., number of iterations). Within these exchanges, the detection rules are improved from one iteration to another as they are capable of detecting the new generated malicious patterns. On the other side, within the lower-level, the generated malicious artificial patterns are improved from one iteration to another that they can escape being detected by the produced detection rules which are sent by the upper-level. At the end of these exchanges, the best detection rules present the final output produced by our BMD approach. Figure 1 shows the key parts (upper-level and lower-level) reflecting our main contributions. Details related to each of the BMD phases will be given in Section 3.2.

3.2 BMD phases

As previously illustrated in Figure 1, BMD is based on two main phases: (1) detection rules generation (upper-level problem) and (2) generation of artificial

malicious patterns (lower-level problem). The first phase (Section 3.2) invokes a detection model that uses an enriched collection of malicious patterns, i.e., the malware patterns from the base of examples — these are stored in the database of malicious API call sequences (*MPDB*) — and the artificially generated ones (the output of the second phase) — these are stored in the artificial malicious patterns database (*AMDB*) — to generate a set of detection rules (*SDR*). Throughout this phase, malicious programs will be detected among the new apps by using the generated detection rules. The evaluation of the generated detection rules (upper-level) is based on the coverage of the base of examples (input) and also the coverage of the artificial malicious patterns generated by the lower-level. These two measures are used to be maximized by the population of detection rules solutions. The second phase (Section 3.2) explains the required steps for the generation of artificial malicious patterns, and is defined via two main steps: The first step is responsible mainly for extracting the API call sequences with their corresponding depths from the collection of normal and malicious applications — i.e., from the database of benign sequences (*DBB*) and the database of malicious sequences (*DBM*) — to transmit them afterwards to the next step. Through the second step, the process of the patterns construction is subdivided into two main sub-steps: First, the frequent API call sequences, referred to as frequent item sets (also called patterns) — these are stored in the database of malicious patterns (*DBMFIV*) and the database of benign patterns (*DBBFIV*) —, 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. The output is stored in both databases: the filtered malicious patterns database (*MPDB*) and the filtered benign patterns database (*BPDB*). In the second sub-step (Section 3.2), a database of artificially generated malware patterns (*AMDB*) is created using the set of the selected patterns. This is achieved via the use of a GA aiming at diversifying the base of malware examples with unseen artificial malicious patterns in order to escape the detection rules in the upper-level.

The generation process of artificial malicious patterns is performed using a GA that maximizes the distance between the generated malicious patterns and the reference benign patterns (input, not-generated patterns), and minimizes the distance between the generated malicious patterns and the reference malicious ones. Also, the GA maximizes the number of the generated malicious patterns that are not detected by the leader; i.e., by the detection rules.

Based on this bi-level BMD hierarchy, the upper-level is executed for a number of iterations, then the lower-level for another number of iterations. After that, the best solution found in the lower-level will be used by the upper-level to evaluate the associated solution, i.e., the detection rules, and then this process is repeated several times until reaching a termination criterion (e.g., number of iterations). Both levels are dependent. As presented, the evaluation of every detection rule solution (upper-level) requires running a search algorithm to find the best undetectable artificial malicious patterns by the upper-level solution.

The ultimate output of our BMD approach is the best set of detection rules. An example of such a rule is given as follows (also schematized in Figure 3): DR1: IF (MF301 AND MF35 AND MF405) OR ((MF21 AND MF211) OR (MF301 AND MF311 AND MF78)) THEN App is malicious. In this sample rule ($DR1$) which shows that an app App is considered as a malicious one, the antecedent corresponds to a succession of patterns (i.e., $MF301$, $MF35$, etc.) with a set of logical operators. The consequent of a detection rule determines its nature (malicious/benign).

Detection rules generation phase In order to produce a set of effective detection rules, and as shown in Figure 1 and Algorithm 1, the upper-level's first step consists of generating a set of detection rules (Algorithm 1, line 1) which will go through an evaluation process (Algorithm 1, lines 2-3). This evaluation is based on the coverage of the base of examples (input) and also the coverage of the artificial malicious patterns generated by the lower-level. These two measures are used to be maximized by the population of detection rules solutions (Algorithm 1, lines 4-6). The output of this module is a set of final detection rules (RDB) that will be used by the detection task which is responsible for labeling new apps either as malicious or as benign. As the upper-level relies upon a GP process and in order to evaluate a generated detection rule, an objective function is formulated. This function helps maximizing the coverage of patterns from the base of examples (input), i.e., $MPDB$, and to maximize the coverage of the generated artificial patterns at the lower-level, i.e., $AMDB$. Thus, the objective function of a detection rule (DR), at the upper-level, is defined as follows:

$$f_{upper}(DR) = Max\left(\frac{\frac{Precision(DR)+Recall(DR)}{2} + \frac{\#damp}{\#amp}}{2}\right) \tag{2}$$

where $\#damp$ refers to the number of detected artificial malicious patterns and $\#amp$ refers to the number of artificial malicious patterns and

$$Precision(DR) = \frac{\sum_{i=1}^{p} DR_i}{t} \tag{3}$$

$$Recall(DR) = \frac{\sum_{i=1}^{p} DR_i}{p} \tag{4}$$

p is the number of detected malicious patterns after executing the solution, i.e., the detection rule, on the base of malicious patterns examples ($MPDB$), t is the total number of malicious patterns within $MPDB$, and DR_i is the i^{th} component of a detection rule DR such that:

$$DR_i = \begin{cases} 1 \text{ if the } i^{th} \text{ detected malicious pattern exists in the} \\ \text{malicious base of examples} \\ 0 \text{ otherwise} \end{cases} \tag{5}$$

The evaluation of the upper-level detection rules depends on the lower-level artificial malicious patterns . Thus, the fitness function of solutions at the upper-level (detection rules) is calculated, at each iteration, after the execution of the optimization algorithm in the lower-level. For each solution (detection rule) of the upper-level an optimization algorithm, at the lower-level, is executed to generate the best set of artificial patterns that cannot be detected by the detection rules at the upper-level. An objective function is formulated at the lower-level to maximize the number of undetected artificial patterns that are generated (Equation 6 in Section 3.2). The technical description of the two BMD levels is given in Algorithm 1 (upper-level) and in Algorithm 2 (lower-level).

Algorithm 1: Upper-level algorithm

Inputs: $MPDB$: set of malicious patterns, NDR: number of generated rules, NAP: number of generated artificial patterns in $AMDB$, NU: number of iterations in the upper-level, NL: number of iterations in the lower-level
Output: Set of detection rules RDB

1: $SDR_0 \leftarrow$ Initialization(NDR,$MPDB$) /*First generation of detection rules*/
2: For each DR_0 in SDR_0 do /*DR means detection rule*/
2.1: $SAP_0 \leftarrow$ APGeneration(DR_0,$MPDB$,NAP, NL) /*call lower-level*/
2.2: $DR_0 \leftarrow$ Evaluation(DR_0,$MPDB$,SAP_0)
3: End For
4: $t \leftarrow 1$
5: While ($t < NU$) do
5.1: $Q_t \leftarrow$ Variation(SDR_{t-1})
5.2: For each DR_t in Q_t do /*Evaluate each rule based on upper fitness function*/
5.2.1: $DR_t \leftarrow$ UpperEvaluation(DR_t,$MPDB$)
5.2.2: $SAP_t \leftarrow$ APGeneration(DR_t,$MPDB$,NAP,NL)
5.2.3: $DR_t \leftarrow$ EvaluationUpdate(DR_t,SAP_t)
5.3: End For
5.4: $U_t \leftarrow Q_t \cup SDR_t$
5.5: $SDR_{t+1} \leftarrow$ EnvironmentalSelection(NDR,U_t)
5.6: $t \leftarrow t+1$
6: End While
7: $RDB \leftarrow$ FittestSelection(SDR_t)

When using bi-level optimization, it is necessary to define problem-specific genetic operators to obtain the best performance. To adapt bi-level optimization to our malware detection problem, the required steps are to create for both levels (algorithms): (1) solution representation, (2) solution variation, and (3) solution evaluation. We examine each of these in what follows.

Solution representation One key issue when applying a search-based technique is to find a suitable mapping between the problem to be solved and the techniques to be used when detecting malicious apps. A GP algorithm is used

Algorithm 2: Lower-level algorithm

Inputs: *MPDB*: set of malicious patterns extracted from the base of examples, *BPDB*: set of benign patterns extracted from the base of examples, *SDR*: set of generated detection rules, *G*: number of generations, *N*: population size
Output: Set of generated malicious artificial patterns *AMDB*
1: $SAP_0 \leftarrow$ Initialization(*BPDB*,*MPDB*,*N*,*G*)
2: $SAP_0 \leftarrow$ Evaluation(SAP_0,*BPDB*,*MPDB*,*SDR*) /*Evaluation depends on SDR*/
3: $t \leftarrow 1$
4: While ($t < G$) do
4.1: $Q_t \leftarrow$ Variation(SAP_{t-1})
4.2: $Q_t \leftarrow$ Evaluation(Q_t,*BPDB*,*MPDB*,*SDR*)
4.3: $U_t \leftarrow Q_t \cup SAP_t$
4.4: $SAP_{t+1} \leftarrow$ EnvironmentalSelection(N,U_t)
4.5: $t \leftarrow t+1$
5: $AMDB \leftarrow$ FittestSelection(SAP_t)
6: End While

[17] for the upper-level optimization problem. In GP, a solution is composed of terminals and functions. When applying GP to solve a specific problem, terminals and functions should be carefully selected and designed. After evaluating many parameters related to the malware detection problem, the terminals and the functions are decided in order to meet the current problem's requirements. In fact, the terminals correspond to different patterns (frequent API call sequences). The functions that can be used between these patterns are Intersection (*AND*) and Union (*OR*). More formally, each candidate solution in this problem is a detection rule that is represented by a tree:

1. Each leaf-node (Terminal) belongs to the set of patterns.
2. Each internal-node (Functions) belongs to the connective set (logic operators {AND, OR}).

An individual in the upper-level has the form of a GP tree as illustrated in Figure 3. Each individual produces an if/then rule to determine the maliciousness of an application which is being analyzed. An example of a detection rule (*DR*1) was previously given in Section 3.2 for the GP tree presented in Figure 3. For the crossover operation, new offspring if/then rules are created for the new population by exchanging randomly chosen parts of two selected parent GP trees. For example, the rightmost sub-tree $MF11 - AND - MF78$ could be exchanged with another sub-tree selected in another individual. For the mutation operator, one new offspring if/then rule is generated for the new population by randomly mutating a randomly chosen part of one selected GP tree. For example, the node *AND* could be mutated to the *OR* operator.

As previously mentioned, for the lower-level optimization problem, a GA is used to generate artificial patterns. The generated artificial patterns are composed of API call sequences represented as item vectors. API call sequences

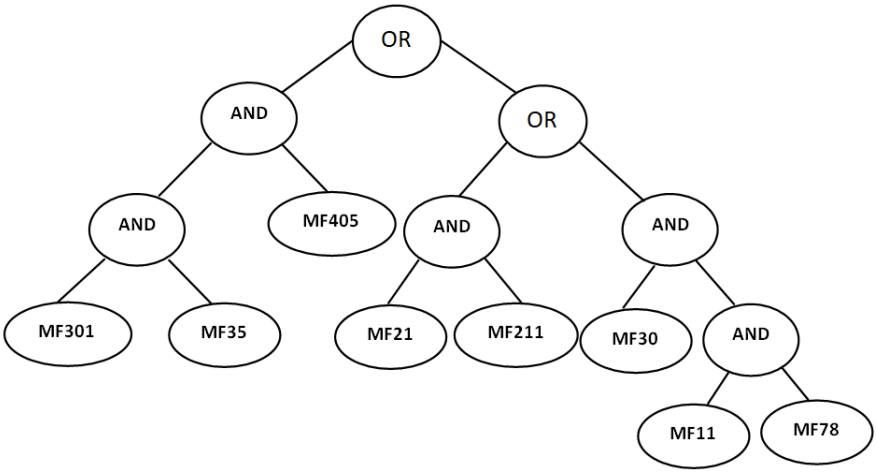

Fig. 3: An example of a malware detection rule encoded as a GP tree.

are 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. To generate an initial population for the GP, we start by defining the maximum tree's length (maximum number of API call sequences per solution). The tree's length is proportional to the number of API call sequences to use for malware detection. A high tree's length does not necessarily mean that the results are more precise. These parameters can be either randomly chosen or specified by the user.

Solution variation Specific variation operators have to be designed to combine information from individuals (parents). More precisely, basic operators (i.e., crossover and mutation detailed in the following) should be adapted to our solution representation.

In the upper-level, the GP mutation operator can be applied to a function node or to a terminal node. It starts by randomly selecting a node in the tree. Then, if the selected node is a terminal (pattern), it is replaced by another terminal; if it is a function (AND-OR), it is replaced by a new function; and if tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree.

As for the GP crossover operator, two parent individuals are selected, and a sub-tree is picked on each selected parent. The crossover swaps the nodes and their related sub-trees from one parent to the other. This operator must ensure the respect of the depth limits. The crossover operator can be applied with only parents having the same rule aim (malicious or benign pattern to detect). Each child thus combines information from both parents. In any given generation, a variant will be the parent in at most one crossover operation.

Solution evaluation The encoding of an individual should be formalized as a mathematical function called the "fitness function". The fitness function quantifies the quality of the proposed detection rules and the generated artificial malicious patterns. The goal is to define efficient and simple fitness functions in order to reduce the computational cost. For our GP adaptation (upper-level), we used the fitness function f_{upper} defined in Equation 2 to evaluate detection-rules solutions. For the GA adaptation (lower-level), we used the fitness function f_{lower} defined in Equation 6 to evaluate the generated artificial malicious patterns.

Generation of malicious patterns phase In the lower-level, an optimization algorithm (Algorithm 2) is executed to generate the best set of artificial patterns that cannot be detected by the detection rules at the upper-level. The second population (lower-level) should seek to optimize the following two objectives:

1. Maximize the generality of the generated "artificial" patterns by maximizing the similarity with the reference malicious patterns examples, and by minimizing their similarity with the benign patterns examples.
2. Maximize the number of uncovered artificial malicious patterns by the solutions of the first population (detection rules).

These two objectives define the cost function that evaluates the quality of a solution, an artificial malicious pattern (AP), and then guides the search. The cost of a solution which is a set of generated malicious patterns (referred to as SAP), is evaluated as the average costs of the included malicious patterns. Formally, the fitness function to maximize is:

$$f_{lower}(AP) = Max(z + \sum_{i=1}^{N} f_{Qual}(AP_i)) \tag{6}$$

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

$$z = \#gamp - \#dagmp \tag{7}$$

$\#gamp$ refers to the number of generated artificial malicious patterns and $\#dagmp$ refers to the number of detected artificial generated malicious patterns.

The function $f_{Qual}()$, defined in Equation 8, guarantees the diversity of the generated malicious patterns.

$$f_{Qual}(AP_i) = \frac{Sim(MS,AP_i) + Sim(BS,AP_i) + Overlap(AP_i)}{3} \tag{8}$$

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

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

$$Sim(MS, AP_i) = \frac{\sum MS_{j \in MS} Sim(AP_i, MS_j)}{|MS|} \tag{9}$$

where $j \in [1, m]$; m indicates the total number of malicious patterns.

2. $Sim(BS, AP_i)$ refers to the similarity between the generated pattern AP_i and the benign patterns (BS) which has to be the lowest.

$$Sim(BS, AP_i) = \frac{\sum BS_{k \in BS} Sim(AP_i, BS_k)}{|BS|} \tag{10}$$

where $k \in [1, p]$; p indicates the total number of benign patterns.

3. Overlap(AP_i) is measured as the average value of the individual $Sim(AP_i, AP_l)$ between the generated pattern AP_i and all the other generated patterns AP_l in the generated dataset $AMDB$. l refers to the total number of the generated artificial patterns.

$$Overlap(AP_i) = 1 - \frac{\sum AP_{l, i \neq l} Sim(AP_i, AP_l)}{|AP|} \tag{11}$$

To calculate the similarity $Sim()$ between two patterns, we adapted the Needleman-Wunsch [28] alignment algorithm to our context. A detailed description of the similarity function $Sim()$ can be found in [19].

Let us recall that the lower-level's aim is to generate a set of artificial malicious patterns in order to maintain a fairly varied and renewed database of malicious patterns that try to escape being detected by the detection rules of the upper-level. To present this process, we start by giving a general overview highlighting the process of artificially generating the malicious patterns. Then, we detail the description of the pattern's encoding schema using a GA.

General overview As shown in Algorithm 2, the process of artificially generating malicious patterns using a GA goes through three main steps. In the first step, (Algorithm 2, line 2), a set of malicious patterns is produced with compositional characteristics similar to those of the real patterns stored in the filtered malicious patterns database ($MPDB$) that comprises the malicious patterns. In the second step, each generated pattern is evaluated according to a fitness function; (Algorithm 2, line 3); this is to keep only the best fitting patterns (Algorithm 2, line 4). The third step described via its sub-steps (Algorithm 2, lines 8-9) 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 (the number of generation is reached). Once the artificial set of malicious patterns ($SAMP$) is generated, it will be stored in its related artificial set of malicious patterns database ($AMDB$).

Pattern encoding using a GA Let us recall that a GA 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. Our used

GA begins with a set of suitable solutions which are, in our case, 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 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 GA; 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 our problem in a form which can be processed by the GA. We consider that 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. We will encode the solutions as identifier elements as $\{M1, M2, \ldots, MX\}$ where X represents the total number of extracted item vectors (API call sequences). 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 representation of a gene and a chromosome is given in Figure 4.

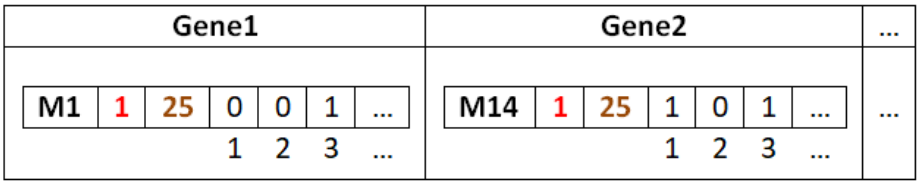

Fig. 4: A GA chromosome representation: A chromosome is a sequence of genes each encoding an item vector corresponding to a particular behavior defined by a sequence of API calls.

Please, note that this second phase, i.e., the generation of malicious and benign patterns phase, was formerly proposed and detailed in [19].

3.3 Detection model process based on detection rules

Throughout this phase, our model will perform its classification task (upper-level problem) where a new app, the executable, will be classified either as a

malware or as a benign. This is achieved using the set of detection rules (*SDR*). Formally, the first step 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 *MPDB* and *BPDB* databases.Then, the obtained patterns are compared to the antecedent of *SDR*. The comparison will allow the executable to be either classified as a malware or as a benign app.

4 Experimental study

4.1 Research questions and benchmark datasets

Our proposed malware detection approach is evaluated by conducting a research study. The study is conducted to quantitatively assess BMD's performance when applied in real-world settings. More precisely, a comparative study with a set of well known state-of-the-art malware detection approaches is performed with the aim to answer the following research questions (RQs):

- RQ1: To what extent can BMD detect malicious patterns?
- RQ2: How does BMD perform when compared to the state-of-the-art methods?
- RQ3: What are the benefits of using a bi-level approach?
- RQ4: How does BMD perform in terms of efficiency?

To answer RQ1, we evaluate the performance of BMD using precision, recall, specificity, F1_score, Area Under the Receiver Operating Characteristics (ROC) Curve (AUC), and accuracy. To answer RQ2, we compare our obtained results to those generated by recent prominent state-of-the-art methods. For RQ3, we demonstrate the benefits of using an evolutive anti-malware system against unknown generated attacks. To do so, we analyze the results based on the used evaluation metrics by comparing them to the results obtained by other anti-malware systems (state-of-the-art approaches and anti-viruses systems) by confronting them to a set of unknown variants of malware. To answer RQ4, we evaluate the execution time, using the CPU Time measure, required by our proposed approach based on different parameter settings. In fact, there is a cost in solving every lower-level optimization problem in each iteration. To demonstrate the ability of our approach to detect malicious apps within a reasonable time-frame, an evaluation of the execution time is required (discussed in Section 4.6). We show that our proposed BMD solution outperforms existing malware detection approaches based on the previous research questions. The details of the used methods for comparison are highlighted in Section 4.3.

To conduct our experiments, we gathered 3 000 Android apps where 2 000 are malicious obtained from the Android Malware Data set (AMD set) [46] and from the DROIDCat dataset [32]. The rest of the 1 000 apps are benign files gathered from the DroidCat dataset [32] and also from various portable benign tools such as Google play. The number of apps used within the experimental study are explained in Section 4.2. Let us recall that the main goal is to artificially

generate malicious patterns. This goal is of utmost importance as it can bring a solution to a great problem that researchers in different fields have to deal with and which is the problem of data availability. In fact, the use of this precise number of apps is related to the accessible and available benchmarks within the malware detection field. We try to deal with such shortcomings by producing artificial malicious patterns that come to enrich the base of examples.

4.2 The choice of the number of malware and the benign apps

To explain the used number of apps in the conducted experiments within our BMD method, we can refer to the AMD paper [19]. In fact, in AMD, to conduct the experimental analysis, two tests were performed. For the first experiment, a balanced dataset consisting of 800 malicious executables along with 800 benign executables was created. In this setting, based on the API calls, a total of 4 605 distinct malicious item sets (API call sequences) and a total of 1 552 distinct benign item sets were extracted. For the second experiment, an unbalanced dataset was created and where all of the collected apps resulting in 2 000 malicious executables and 1000 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 second experiment (unbalanced dataset) allowed us to have better results based on the considered evaluation metrics. Based on these results, we have built Experiment 1, which uses 2000 malicious patterns and 1000 benign ones, in the current paper. In this version of our manuscript, we have added another experiment (Experiment 2) to investigate the effect of the use of different amounts of both benign and malicious examples on the detection rates of BMD. This additional experiment uses 3000 benign apps and 6000 malicious applications gathered from different databases (TheZoo[5], AMD set and the DROIDCat datasets). These apps will serve to extract the frequent API call sequences. The detailed obtained numbers are shown in Table 1:

Table 1: Number of patterns extracted for each experiment

Experiments	Number of apps	Number of obtained distinct item sets (API calls)	Number of frequent item sets (API call sequences or patterns)
Experiment 1	2000 Malicious	29 483 201	27 534 880
	1000 Benign	11 302 447	10 172 203
Experiment 2	6000 Malicious	74 582 915	67 124 623
	3000 Benign	24 864 025	22 377 622

In order to assess the impact of increasing the number of apps on the detection rates of BMD, the precision, recall, specificity, accuracy, F1_score (FS), false

[5] https://github.com/ytisf/theZoo

positive rate (FPR) and false negative rate (FNR) are calculated. The obtained results are presented in Table 2:

Table 2: The different obtained measures for both Experiment 1 and Experiment 2 in terms of TP, FP, TN, FN, precision, F1_score and AUC.

	Precision	Recall	Specificity	Accuracy	FS	FPR	FNR	Execution time (hours)
Experiment 1	98,06	98,34	98,33	98,18	97,79	04.63	01,63	5.2
Experiment 2	95,81	98,37	98.64	97,12	97,08	04,09	01,62	12.4

We can deduce from Table 2 that increasing the number of patterns certainly improved slightly the FPR and FNR rates but the accuracy dropped which can be explained by the difference between the numbers of malicious and benign apps used. Accuracy is better when having symmetric datasets which is not the case of our method which needs more malicious apps in order for the GA to produce malicious patterns. The recall and specificity values increased as the number of examples increased, although this minor improvement came at a cost in terms of execution time (7.2 additional hours). When using an important number of patterns, the EAs in both levels needed more time to produce good solutions (detection rules in the upper-level and artificial malicious patterns in the lower-level). We can conclude that when increasing significantly the base of examples it is clear that the detection rates improve but that is only interesting when using advanced equipments (e.g., Graphics Processing Unit (GPU)). Also, in order to fairly compare our work to other state-of-the-art methods in terms of the used number of apps, we have conducted comparisons to four different state-of-the-art methods which are Sen et al.'s method [38], Zhu et al.'s method [50], D'Angelo et al.'s method [13], and Aslan et al.'s method [6].

The following table describes the used numbers of applications to build the respective state-of-the-art-methods.

Table 3: The numbers of applications used to build different state-of-the-art-methods.

<table>
<tr><th>Method</th><th>Used dataset</th><th>Number of used apps</th></tr>
<tr><td>Sen et al, 2018</td><td>MalGenome (2011) Android Malgenome Project</td><td>1,260 Android malware samples</td></tr>
<tr><td>Zhu et al., 2018</td><td>Android official app store and VirusShare. The total number of samples gathered is 2,130: 1,065 benign samples and 1,065 malicious samples.</td><td>The used samples as the training dataset to the Rotation Forest algorithm are 600 benign samples and 600 malware.</td></tr>
<tr><td>D'Angelo et al., 2021</td><td>TEKDEFENSE malware dataset, 2019 and Malware dataset for security researchers, data scientists, 2019 (Windows malware dataset).
The whole dataset is split into eight malware families and into a training sample and a testing sample.</td><td>The used numbers of malware types for the training phase are as follows:
<table>
<tr><th>Family</th><th>Number of samples</th></tr>
<tr><td>Adware</td><td>265</td></tr>
<tr><td>Backdoor</td><td>700</td></tr>
<tr><td>Downloader</td><td>700</td></tr>
<tr><td>Dropper</td><td>624</td></tr>
<tr><td>Spyware</td><td>582</td></tr>
<tr><td>Trojan</td><td>700</td></tr>
<tr><td>Virus</td><td>700</td></tr>
<tr><td>Worms</td><td>700</td></tr>
</table>
The testing phase uses different malware samples (from the whole base of examples) where the detection of each family is performed separately.</td></tr>
<tr><td>Aslan et al., 2021</td><td>Malware samples (20 000) were collected from various sources:
Das Malwerk, MalwareBazaar, Malware DB, Malware Benchmark, Malshare, Tekdefense, ViruSign, VirusShare, KernelMode and they are split into 14 malware families. Benign samples (3000) were collected from office documents, games, system tools, and other third party's software.</td><td>Authors used only 7000 malicious samples, split between 14 malware families, among the collected set of malwares.</td></tr>
</table>

We can deduce from Table 3 that the number of used apps seem fair when compared to different state-of-the-art methods and can be considered reliable to make conclusions regarding the obtained results.

4.3 Peer algorithms and parameters settings

To compare our results to other existing works, we choose four recently published state-of-the-art approaches that are similar to our BMD approach. These are the Zhu et al.'s approach [50], the AMD approach [19], the Sen et al.'s approach [38] and Mystique [27]. In [50], authors proposed a method to detect Android malware based on the combination of multiple types of static features like permissions request, monitoring system events, sensitive APIs and permission-rate. Also, the authors make use of a machine learning algorithm which is Rotation Forest (RF). The RF classifier is a method for generating classifier ensembles based on feature extraction proposed by Rodriguez [35] in which each base classifier on the entire dataset is trained. Because it preserves all of the main elements and uses the whole training dataset for each individual classifier, RF is thought to be robust. It also adopts Principal Component Analysis (PCA) [39] to handle the feature subset randomly extracted for each base classifier in order to intensify the diversity. Concerning the AMD approach [19], the Sen et al.'s approach [38] and Mystique [27], they were explained in Section 2.2.

Parameter settings have a great impact on the performance of a search algorithm. To ensure fairness of comparisons between evolutionary approaches, we use the parameter settings specified in Table 4. In this way, all of the evolutionary approaches perform 810 000 function evaluations in each run. For our experiments, we generated 476 000 malicious patterns with Eclipse[6] (about the half of the number of the generated malicious patterns in our experiment) with a total number of 4 407 API calls (items). Both levels are run with a population of 30 individuals and 30 generations. The algorithm will perform 810 000 fitness evaluations for each level. In both levels, we used the trial and error method to set the population size and the number of generations. This means that we have made several experiments using different values for these parameters. Following these experiments, we concluded that when using a population size of 30 for both levels, the fitness functions become stabilized around the 50^{th} generation. For these reasons, the algorithms 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, we used a crossover rate of 0.9 and a mutation rate of 0.5 for both algorithms.

4.4 Performance analysis

In this section, we discuss the results obtained using our BMD approach and thereby we respond to RQ1 and RQ2 highlighted in Section 4.1.

Cross validation and overall view of the results To estimate how accurately our predictive model will perform in practice, 10-fold cross-validation was used to evaluate the approach, we consider all of the collected apps resulting in 2 000 malicious executables and 1 000 benign executables. For this test, a total

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

Table 4: EAs' parameters used by each approach.

Approach	Parameters					
	Population size		Generation size	Crossover rate	Mutation rate	Number of evaluations
BMD	Upper- level	30	30	0.9	0.5	810 000
	Lower-level	30	30	0.9	0.5	810 000
AMD	180		4 500	0.9	0.5	810 000
Sen et al.	Malware generation	500	1 000	0.1	0.9	500 000
	Anti-malware generation	310	1 000	0.1	0.9	310 000

of 29 483 201 (27 534 880 patterns) distinct malicious item sets and a total of 11 302 447 (10 172 203 patterns) distinct benign item sets were extracted. The conducted test is summarized in Table 5. The goal of using cross-validation is to test our model's ability to predict new apps and to give an insight on how the model will generalize to an independent dataset.

Table 5: Ten-fold cross validation results.

Classifier	TP	FP	TN	FN	Recall	Specificity	Accuracy	Precision	FS	AUC
BMD	98.12	01.88	98.18	01.62	98.34	98.33	98.18	98.06	97.79	86.80
LR	90.21	09.79	85.23	14.77	85.93	89.70	91.90	90.21	88.02	86.21
LDA	78.92	21.08	96.75	03.25	96.04	82.11	97.84	78.92	86.64	81.12
RF	95.23	04.77	98.29	01.71	98.24	95.37	96.76	95.23	96.71	87.30
J48	98.65	01.35	97.03	02.97	98.16	93.17	95.53	92.80	95.40	82.39
NB	92.30	07.70	28.41	71.59	56.32	78.65	60.36	92.30	69.95	59.23
k-NN	86.25	13.75	91.98	08.02	91.49	87.00	89.12	86.25	88.79	83.69

LR: Logistic Regression; LDA: Linear Discriminant Analysis; RF: Random Forest; J48: Decision Tree; NB: Naive Bayes;k-NN: k-Nearest Neighbours; FS: F1_score.

Our BMD approach is compared to six different classifiers namely: Logistic Regression (LR), Linear Discriminant Analysis (LDA), Random Forest (RF), Decision Tree (J48), Naive Bayes (NB) and k-Nearest Neighbours (k-NN), as presented in Table 5 which also shows the true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), recall, specificity, accuracy, precision, F1_score and the Area Under the Receiver Operating Characteristics (ROC) Curve (AUC) of the obtained dataset.

Based on the obtained results, and in terms of accuracy, it can be observed that BMD outperformed all of the six classifiers (accuracy = 98.18%). Also, we can state that the four classifiers (RF, J48, k-NN and LDA) have higher accuracy rate while the other two classifiers (LR and NB) have a very close accuracy. Figure 5 shows the box plot output of the classifier's accuracy collected during 10-fold cross validation. It is observed that RF and J48 have the highest specificity values whereas NB and LR have the lowest specificity values. Among the classifiers, with the same configuration, RF consumed the highest training

time, 8.25 seconds, and J48 has taken the lowest training time, 0.83 seconds. It is also observed that the training time is less for LR and NB than k-NN, for example, both LR and NB have taken 0.02 seconds less time than k-NN.

False negatives and false positives analysis In this section, we will analyze BMD's performance 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, or type 1 errors, since they are judged as a less significant problem than the false negatives. In our analysis of the results, we aim at keeping both the false positive and the false negative rates as low as possible. In the conducted experiment, and by analyzing 27 534 880 malicious patterns and 10 172 203 benign patterns, the highest values of FP and FN registered among the classifiers, as shown in Table 5, were obtained by the k-NN and NB classifiers respectively with $FP = 13.75\%$ and $FN = 71.59\%$. BMD got the best values with $FP = 01.88\%$ and $FN = 01.62\%$. In the aim of reducing the number of FPs and FNs, we can increase our base of examples with more benign and malicious patterns. But, we should keep in mind that making the detection model over-fitting may cause the degradation of the detection performance.

Precision interpretation Precision is a good measure to determine specially when the amount of false positives is high. For instance, in our 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, we can see that the best reached precision value is 98.06% for BMD and RF classifier ranked second with 95.23%. At this level, we can tell that our BMD 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 our detection process which is benefiting in keeping the base of examples fairly varied.

Accuracy, recall and specificity interpretation Having a high accuracy does not necessarily mean that our model is the best. Therefore, we have to look at other metrics (i.e., precision and recall) to evaluate the performance of our model. For instance from Table 5, among the six classifiers LDA registered the highest value of 97.84%. BMD got the best accuracy value of 98.18% for BMD which means that our model is approximately 100% accurate which is explained by the large number of correctly predicted observations. These good results demonstrate the impact of our BMD 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.

In our BMD detection model, the recall metric calculates how many of the actual positives our model captures through labeling it as positive (true positive). For instance, in our malware detection model, the consequence of a fraudulent

behavior (actual positive) that is predicted as non-fraudulent (predicted negative) can be noxious to the operating system and to the user. In our case a value of 98.34% as recall for BMD can be positively interpreted. In fact, this satisfying value can be explained by the high number of true positives accurately detected (98.12%).

Based on the fact that the sensitivity (recall) quantifies the avoiding of false negatives, the specificity does the same for false positives. In our case, we can consider that the reached value of 98.33% of specificity obtained by BMD is indeed a promising result. In fact, the high number of true negatives accurately detected explains the obtained results. The obtained varied base of examples guarantees a better detection of malicious patterns.

F1_score and AUC interpretation When measuring how well our detection approach is doing, it is useful to have the F1_score to describe its performance. In our obtained results, in Table 5 we can see that BMD reached 97.79% of F1_score and this could be explained by the high values of precision and recall achieved by our detection model. In fact, we also registered 98.06% of precision and 98.34% of recall.

The area, for its part, measures discrimination, that is, the ability of the pattern to correctly classify positive and negative instances. The best AUC value is obtained with our BMD approach. In fact, AUC equals 86.80% which means that BMD could be considered efficient in separating malicious and benign instances. We can affirm that when we assure a continuous variability to our base of examples by injecting the generated malicious patterns, we guarantee a better detection of malware.

Graphical analysis of the ROC curve In order to perform a graphical based evaluation of our conducted approach, we use the ROC curve analysis. We represent the obtained results by the mean of two graphics/curves where one curve is drawn in terms of accuracy vs false positive rates and the other is in terms of true positive rate vs false positive rates. Figure 5 represents the obtained ROC curves. To choose the most appropriate cut-offs for our experiment we need the ROC curves. The best cut-off has a highest accuracy of 98.18%, the highest true positive rate of 98.37%, and the lowest false positive rate of 01.87%. 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, we previously calculated and discussed the AUC value which serves as a quantitative summary to evaluate the strength of the BMD retained patterns in classifying positive and negative instances.

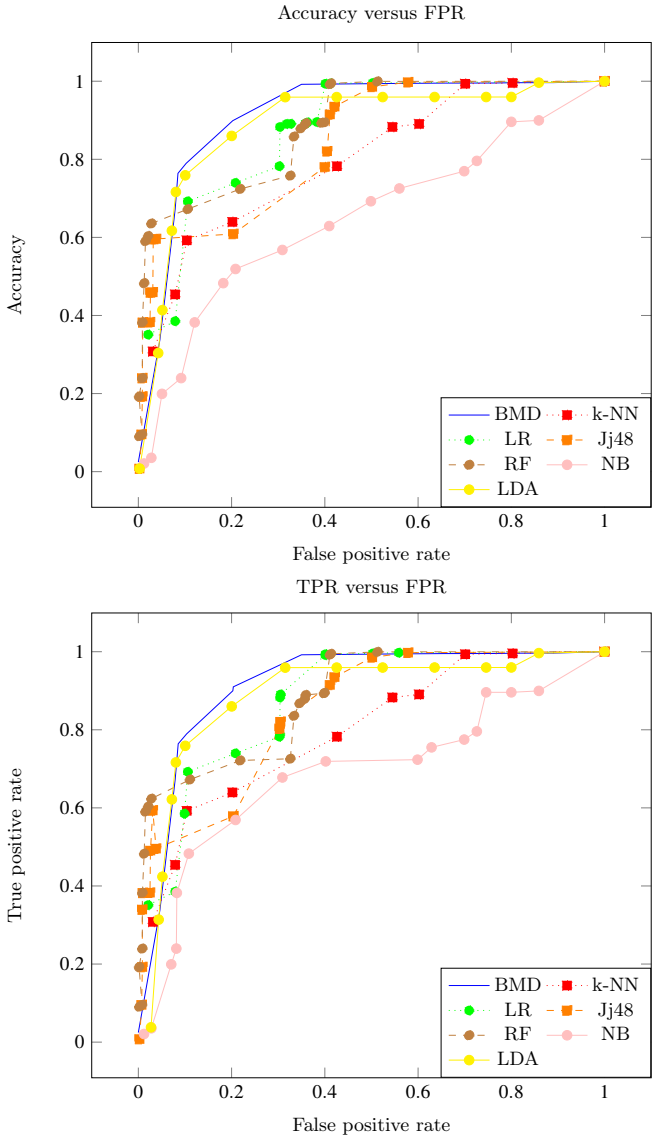

Fig. 5: BMD's obtained ROC curves with different classifiers.

4.5 Evaluation of the contribution of the BMD approach and comparison with state-of-the-art approaches

As an answer to RQ3 highlighted in Section 4.1, and in order to perform comparisons, it is very interesting to show that our proposed approach outperforms existing malware detection approaches. Also, a comparison with existing state-of-the-art approaches namely Zhu et al. [50], Sen et al.[38] and AMD [19] is

shown in Table 7. This comparison is helpful to evaluate the benefits of the use of our bi-level approach in the context of malware detection. We perform an evaluation in terms of accuracy, recall, precision and false positive rate against the new variants of malware and 0-day attacks in the Drebin dataset [5] and a more recent set of malicious apps which is Android Adware and General Malware dataset (AAGM) [24]. The Drebin dataset contains 123 453 benign applications and 5 560 malware samples whereas AAGM dataset contains 1 900 apps. Since these samples are not used in the generation of artificial malware process, they are considered as unknown malwares by the developed system. Further comparisons are made against a set of different antivirus engines also using the same dataset. We used VirusTotal[7], 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 results demonstrate that the BMD approach effectively detected new variants of known attacks. The performance of the developed system in the Drebin dataset and AAGM dataset can be seen in Table 6.

Table 6: Accuracy results of BMD, AMD, Zhu et al., Sen et al.'s approaches and top ten commercial engines by Virus-Total on Drebin dataset [5] and AAGM dataset [24].

Anti-malware	Reference	Accuracy(%)	
		Drebin dataset	AAGM dataset
BMD	Our current approach	96.76	97.05
Sen et al.	[38]	95.15	96.46
AMD	[19]	92.28	94.15
Zhu et al.	[50]	88.26	89.01
ESET NOD32	https://www.eset.com	66.68	69.26
AegisLab	www.aegislab.com	66.23	69.13
NANO antivirus	http://www.nanoav.ru	66.23	69.09
VIPRE	https://www.vipre.com	62.53	64.99
McAfee	https://www.mcafee.com	56.21	58.58
Ikarus	https://www.ikarussecurity.com	55.65	57.99
AVG	https://www.avg.com	55.56	57.00
CAT QuickHeal	www.quickheal.com	54.23	54.13
AVware	http://www.avware.com.br/comprar.php	45.56	45.21
Cyren	https://www.cyren.com	45.23	44.89

The improvements made by our BMD approach reported in Table 7 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 from the base of examples our detection model is, the more we ensure that the detection system will be effective in detecting different variants of malware. These results show

[7] https://www.virustotal.com

that our BMD approach outperforms the state-of-the-art methods by offering a powerful malware detection system based on the use of our bi-level approach.

Table 7: Our BMD approach's achieved results compared to Zhu et al., Sen et al. and AMD approaches on Drebin dataset [5] and AAGM dataset [24].

Measure	Zhu et al.'s approach	Sen et al.'s approach	AMD	BMD
Drebin dataset				
Accuracy (%)	88.26	95.15	92.28	96.76
Recall (%)	88.40	87.91	90.42	98.24
Precision(%)	88.16	94.58	94.30	95.23
False positive rate (%)	NR	NR	05.69	04.63
AAGM dataset				
Accuracy (%)	NR	96.46	94.15	97.05
Recall (%)	NR	88.81	90.42	98.79
Precision(%)	NR	95.86	96.37	97.83
False positive rate (%)	NR	NR	03.60	02.20

NR: Not reported

4.6 Execution time evaluation

To answer RQ4, it is important to evaluate the execution time (CPU time) of our BMD approach. To do so, we compared BMD to the three EA-based approaches namely AMD [19], Sen et al. [38] and Mystique [27]. It is expected that BMD requires higher execution time than the other approaches, since BMD has two EAs to be executed in an embedded way to optimize both the upper and lower levels.

All conducted experiments are run on an *Intel*® *Xeon*® Processor CPU E5-2620 v3, 16 GB RAM. To further evaluate the scalability of the performance of bi-level evolutionary algorithms for systems of increasing size, we executed our bi-level tool on Eclipse that contains more than 3.5 MLOCs, without assessing the precision and recall scores.

In fact, as reported from Table 8, the average execution time for BMD is 5.2 hours. Concerning AMD and Sen et al. each one of them took respectively 1.56 hours and 2.2 hours. This can be explained by the fact that in AMD, there is only one EA that evolves artificial malware. Furthermore,in Sen et al., the EAs are independent and executed in a sequential way.

Despite the fact that BMD took higher execution time than AMD and Sen et al.'s approaches, the execution time for BMD seems reasonable because the two EAs within BMD are embedded. Also, the whole process is executed only once in order to generate the rules that will be used to detect the malicious patterns. A new execution of the bi-level algorithm is recommended when major updates are performed on the base of examples used by the upper-level. In addition, this

Table 8: Comparison of BMD with different EA-based state-of-the-art methods.

Method	Used datasets for training	Used features	Architecture	Execution time	Detection rate (Drebin dataset)	Number of evaluations**
AMD [19]	AMD set [46] and DROIDCat [32]: 3000 apps	4407 API calls used to generate API call sequences	A single GA	1.56 hours	92.28	800 000
Sen et al. [38]	MalGenome: 1,260 apps	100 API calls/ 40 permissions	Two independent GPs (run in parallel)	2.2 hours	95.15	772 000
Mystique [27]	MalGenome: 1,260 apps	266 attack features (triggers / permissions/etc) and 14 evasion features (storing data methods/ transmission data methods/etc)	A single GA The system generates malware with specific features covering only 4 attack features and 2 evasion features. N.B.: The model only generates specific malware. There is no specific detection module developed by the authors.	Not reported	The used commercial engines succeed to have less than 30% of detection rate when confronted to the generated malware.	Not reported
BMD	AMD set [46] and DROIDCat [32]: 3000 apps	4407 API calls used to generate API call sequences	Bi-level architecture (with interaction between a GA and a GP)	5.2 hours	96.76	742 000

** The required number of evaluations (with F1_Score = 0.8) to generate acceptable solutions (with good detection rates)

relatively high run-time is not an issue because we are not in a real-time setting. In fact, the algorithms could run in a continuous way. Once we need new detection rules to enrich the base of examples, we select the best rules from BMD in a delayed-mode. Figure 6 shows the number of evaluations needed to generate efficient detection rules. We used the F1_score metric to evaluate the quality of the best solution at each iteration for our BMD approach. We considered an F1_score value higher than 0.8 as an indicator of an acceptable detection rules solution based on our corpus. We selected a threshold value of 0.8, since it represents a good balance between precision and recall that can lead to acceptable detection solutions. In fact, after around 740 000 evaluations, BMD generated detection rules that have 0.8 as the F1_score value. Although BMD needs important execution time, it is clear that the good solutions provided by a single-level approach can be reached quickly by our bi-level adaptation as described in Figure 6. Therefore, we can conclude that the lower-level helped the upper-level to quickly generate good efficient detection rules. We can conclude that an execution time of 5.2 hours is acceptable and reasonable, since the developers will not use our tool in their daily activities, they just need to execute it once to extract the rules.

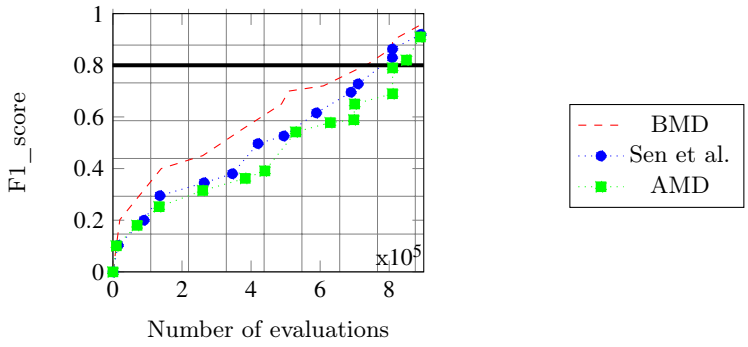

Fig. 6: The required number of evaluations to reach suitable results by the different algorithms (BMD, AMD and Sen et al.) (F1_score = 0.8).

5 Conclusion and future work

In this paper, we have proposed BMD as a new approach for Android malware detection. BMD is based on an efficient rules generation process which is framed as a bi-level optimization process; where the upper-level maximizes the accuracy of designed detection rules, while the lower-level builds a set of artificial malicious patterns that are not and/or less detectable for every upper-level rule. Such a competition between both levels makes the detection rules not only less dependent on the training database content but also more able to detect

new predictable malicious behaviours. BMD has shown its outperformance over many state-of-the-art methods with an accuracy rate that exceeds 97%. This could be mainly explained by the competitive interaction between both levels, which corresponds to the main originality of our work.

Following the study of the threats to validity, several interesting perspectives could be investigated. The internal validity is related to the parameters tuning of the different compared optimization algorithms. Although this tuning is still so far an interesting research direction, there is no consensus among researchers on how to define the parameters' values. For this reason, most practitioners use the trial-and-error method. An interesting future direction could be the design of an adaptive parameter tuning strategy that aims at approximating the best parameters' values for a particular algorithm. The construct threat corresponds to the choice of the peer algorithms. To date, there is no bi-level optimization works in the malware detection field. This obliged us to compare our BMD method to three single-level evolutionary methods [38] [19] [27], one random-forest-based method [50], and several antivirus software tools given in Table 6. We believe that our current work would encourage researchers to adopt bi-level optimization in the design of malware detection methods as done in many other domains such as software engineering [36] and Web service computing [33]. The construct validity refers to the generalization of our results. In this work, we focused on Android malware programs. The consideration of other operating systems could be a very important direction to show the versatility of our BMD approach. Another interesting perspective that we would like to explore consists in the fact that generated malicious patterns are assigned the same level of confidence as the real ones. We believe that structural and semantic analyses of generated artificial patterns could be a motivating direction to assign an adaptive confidence level to each artificial malicious pattern.

References

1. Ab Razak, M.F., Anuar, N.B., Othman, F., Firdaus, A., Afifi, F., Salleh, R.: Bio-inspired for features optimization and malware detection. Arabian Journal for Science and Engineering 43(12), 6963–6979 (2018)
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. Aksu, D., Aydin, M.A.: Mga-ids: Optimal feature subset selection for anomaly detection framework on in-vehicle networks-can bus based on genetic algorithm and intrusion detection approach. Computers & Security 118, 102717 (2022)
4. Altaher, A., Barukab, O.M.: Intelligent hybrid approach for android malware detection based on permissions and api calls. International Journal of Advanced Computer Science and Applications 8(6), 60–67 (2017)
5. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss. vol. 14, pp. 23–26 (2014)
6. Aslan, Ö., Ozkan-Okay, M., Gupta, D.: Intelligent behavior-based malware detection system on cloud computing environment. IEEE Access 9, 83252–83271 (2021)

7. 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)
8. Bahtiyar, Ş., Yaman, M.B., Altınığne, C.Y.: A multi-dimensional machine learning approach to predict advanced malware. Computer Networks 160, 118–129 (2019)
9. Bernardi, M.L., Cimitile, M., Distante, D., Martinelli, F., Mercaldo, F.: Dynamic malware detection and phylogeny analysis using process mining. International Journal of Information Security 18(3), 257–284 (2019)
10. Chaabani, A., Bechikh, S., Said, L.B.: A new co-evolutionary decomposition-based algorithm for bi-level combinatorial optimization. Applied Intelligence 48(9), 2847–2872 (2018)
11. Chen, C.M., Lai, G.H., Lin, J.M.: Identifying threat patterns of android applications. In: Information Security (AsiaJCIS), 2017 12th Asia Joint Conference on. pp. 69–74. IEEE (2017)
12. Davis, L.: Handbook of genetic algorithms (1991)
13. D'Angelo, G., Ficco, M., Palmieri, F.: Association rule-based malware classification using common subsequences of api calls. Applied Soft Computing 105, 107234 (2021)
14. 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)
15. Fan, M., Liu, J., Luo, X., Chen, K., Tian, Z., Zheng, Q., Liu, T.: Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Transactions on Information Forensics and Security 13(8), 1890–1905 (2018)
16. Firdaus, A., Anuar, N.B., Karim, A., Ab Razak, M.F.: Discovering optimal features using static analysis and a genetic search based method for android malware detection. Frontiers of Information Technology & Electronic Engineering 19(6), 712–736 (2018)
17. Golberg, D.E.: Genetic algorithms in search, optimization, and machine learning. addion wesley. Reading (1989)
18. Gupta, S.K., Tripathi, M., Grover, J.: Hybrid optimization and deep learning based intrusion detection system. Computers and Electrical Engineering 100, 107876 (2022)
19. Jerbi, M., Dagdia, Z.C., Bechikh, S., Makhlouf, M., Said, L.B.: On the use of artificial malicious patterns for android malware detection. Computers & Security p. 101743 (2020)
20. Kapare, C.S., Joshi, O.S., Rumao, M.V.: Droiddetector: An android application based on contrasting permission patterns
21. 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)
22. Koh, A.: A metaheuristic framework for bi-level programming problems with multi-disciplinary applications. In: Metaheuristics for Bi-level Optimization, pp. 153–187. Springer (2013)
23. Kolstad, C.D.: A review of the literature on bi-level mathematical programming. Tech. rep., Los Alamos National Laboratory Los Alamos, NM (1985)
24. Lashkari, A.H., Kadir, A.F.A., Gonzalez, H., Mbah, K.F., Ghorbani, A.A.: Towards a network-based framework for android malware detection and characterization. In: 2017 15th Annual conference on privacy, security and trust (PST). pp. 233–23309. IEEE (2017)

25. Legillon, F., Liefooghe, A., Talbi, E.G.: Cobra: A cooperative coevolutionary algorithm for bi-level optimization. In: Evolutionary Computation (CEC), 2012 IEEE Congress on. pp. 1–8. IEEE (2012)
26. Martín, A., Menéndez, H.D., Camacho, D.: Mocdroid: multi-objective evolutionary classifier for android malware detection. Soft Computing 21(24), 7405–7415 (2017)
27. Meng, G., Xue, Y., Mahinthan, C., Narayanan, A., Liu, Y., Zhang, J., Chen, T.: Mystique: Evolving android malware for auditing anti-malware tools. In: Proceedings of the 11th ACM on Asia conference on computer and communications security. pp. 365–376 (2016)
28. Nanni, L., Lumini, A.: Generalized needleman–wunsch algorithm for the recognition of t-cell epitopes. Expert Systems with Applications 35(3), 1463–1467 (2008)
29. 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)
30. Ping, X., Xiaofeng, W., Wenjia, N., Tianqing, Z., Gang, L.: Android malware detection with contrasting permission patterns. China Communications 11(8), 1–14 (2014)
31. Pitolli, G., Laurenza, G., Aniello, L., Querzoni, L., Baldoni, R.: Malfamaware: automatic family identification and malware classification through online clustering. International Journal of Information Security 20(3), 371–386 (2021)
32. 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)
33. Rebai, S., Kessentini, M., Wang, H., Maxim, B.: Web service design defects detection: A bi-level multi-objective approach. Information and Software Technology p. 106255 (2020)
34. Ribeiro, J., Saghezchi, F.B., Mantas, G., Rodriguez, J., Shepherd, S.J., Abd-Alhameed, R.A.: An autonomous host-based intrusion detection system for android mobile devices. Mobile Networks and Applications 25(1), 164–172 (2020)
35. Rodriguez, J.J., Kuncheva, L.I., Alonso, C.J.: Rotation forest: A new classifier ensemble method. IEEE transactions on pattern analysis and machine intelligence 28(10), 1619–1630 (2006)
36. Sahin, D., Kessentini, M., Bechikh, S., Deb, K.: Code-smell detection as a bilevel problem. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(1), 1–44 (2014)
37. de los Santos, S., Guzmán, A., Torrano, C.: Android malware pattern recognition for fraud detection and attribution: A case study. Encyclopedia of Social Network Analysis and Mining pp. 1–9 (2017)
38. Sen, S., Aydogan, E., Aysan, A.I.: Coevolution of mobile malware and anti-malware. IEEE Transactions on Information Forensics and Security 13(10), 2563–2574 (2018)
39. Shlens, J.: A tutorial on principal component analysis. arXiv preprint arXiv:1404.1100 (2014)
40. Sinha, A., Malo, P., Deb, K.: Efficient evolutionary algorithm for single-objective bilevel optimization. arXiv preprint arXiv:1303.3901 (2013)
41. Sinha, A., Malo, P., Deb, K.: A review on bilevel optimization: from classical to evolutionary approaches and applications. IEEE Transactions on Evolutionary Computation 22(2), 276–295 (2017)
42. Sinha, A., Malo, P., Frantsev, A., Deb, K.: Multi-objective stackelberg game between a regulating authority and a mining company: A case study in environmen-

tal economics. In: Evolutionary Computation (CEC), 2013 IEEE Congress on. pp. 478–485. IEEE (2013)
43. Sujithra, M., Padmavathi, G.: Research article enhanced permission based malware detection in mobile devices using optimized random forest classifier with pso-ga. Research Journal of Applied Sciences, Engineering and Technology 12(7), 732–741 (2016)
44. Tong, F., Yan, Z.: A hybrid approach of mobile malware detection in android. Journal of Parallel and Distributed Computing 103, 22–31 (2017)
45. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: Gpu-assisted malware. International Journal of Information Security 14(3), 289–297 (2015)
46. 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)
47. Xiaofeng, L., Fangshuo, J., Xiao, Z., Shengwei, Y., Jing, S., Lio, P.: Assca: Api sequence and statistics features combined architecture for malware detection. Computer Networks 157, 99–111 (2019)
48. Xue, Y., Meng, G., Liu, Y., Tan, T.H., Chen, H., Sun, J., Zhang, J.: Auditing anti-malware tools by evolving android malware and dynamic loading technique. IEEE Transactions on Information Forensics and Security 12(7), 1529–1544 (2017)
49. 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)
50. Zhu, H.J., You, Z.H., Zhu, Z.X., Shi, W.L., Chen, X., Cheng, L.: Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing 272, 638–646 (2018)
51. 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)