IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 2, 2021, accepted April 28, 2021, date of publication May 4, 2021, date of current version May 17, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077295

A Novel Method for Detecting Future Generations of Targeted and Metamorphic Malware Based on Genetic Algorithm

DANIAL JAVAHERI[1], (Member, IEEE), POOIA LALBAKHSH[2], AND MEHDI HOSSEINZADEH[3,4]

[1]Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran 1477893855, Iran
[2]Euler Capital, Drysdale, VIC 3222, Australia
[3]Institute of Research and Development, Duy Tan University, Da Nang 550000, Vietnam
[4]Department of Computer Science, University of Human Development, Al Sulaimaniyah 0778-6, Iraq

Corresponding author: Mehdi Hosseinzadeh (mehdihosseinzadeh@duytan.edu.vn)

ABSTRACT This paper presents a novel solution for detecting rare and mutating malware programs and provides a strategy to address the scarcity of datasets for modeling these types of malware. To provide sufficient training data for malware behavioral modeling, genetic algorithms are used together with an optimization strategy that selectively creates generations of mutated elite malware samples. In our unique method, a sequence of system API calls is extracted using tracker filter drivers in a sandbox environment. The most obfuscated and metamorphic malware are chosen by an elite selection method. The behavioral chromosomes are formed by mapping extracted APIs to genes using linear regression. Our analysis system includes an Internet simulator and a human emulator to deceive intelligent classes of malware to successfully execute themselves and prevent system halting. The evolution process is performed through crossover and permutation of genes, which are encoded based on the addresses of the kernel-level system functions. An objective function has been defined to optimize the vital indicators of malignancy and tracking rate with a linear time complexity. This guarantees that new generations of malware are more destructive and stealthy than their parents. J48 and deep neural networks were employed in our experiments as they are two popular modeling and classification strategies in the area of behavioral malware detection. Real-world malware samples from valid references were used for the performance evaluation of our approach. Comprehensive scenarios were involved in the experiments to evaluate the performance of our proposed strategy. The results demonstrate significant improvement in detection accuracy - up to 5% considering rare and metamorphic malware. The results also demonstrated a considerable enhancement in true positive rate for the proposed deep-learning algorithm.

INDEX TERMS Malware detection, malware unpacking, genetic algorithm, metamorphism, obfuscation, data mining, cyber security.

I. INTRODUCTION

The number of malware attacks has considerably increased in recent years. More than 1.1 billion pieces of malware were released in 2020 alone, of which more than 89 million were created specifically for the Microsoft Windows platform [1]. Figure 1 represents the number of released malware programs in the last decade for three major Operating Systems (OS); Microsoft Windows, Android, and Apple macOS, according to the recent reference data. This chart illustrates that the number of malware programs produced for the Microsoft Windows OS is approximately thirty times greater than the Android OS, and one hundred and thirty times greater than the Mac OS. This has been brought about by the popularity of Microsoft Windows which means this platform is a large target for malware attacks. The consequences of malware attacks in business are considerable. For example, the damage inflicted on business organizations by ransomware attacks in 2019 was estimated in the hundreds of millions of dollars [2]. The increasing growth of malware activity has always been a core concern for researchers in the field of cyber security.

The associate editor coordinating the review of this manuscript and approving it for publication was Frederico Guimarães.

One of the important lessons learned from the recent increase in malware attacks is that many of the new malware programs are duplicates of existing programs but with changes in their executable code made to deceive and circumvent updates to anti-malware tools and malware detection modules. The proliferation of malware programs is partly brought about by the advances in areas such as automated code generation tools, novel code protection methods [3], obfuscation engines, and packers. These advances have been used maliciously to develop novel intelligent malware with polymorphic and metamorphic characteristics. The existence of these tools and technologies has created an opportunity for malicious programmers to harness the power of self-modification and obfuscation already provided in ready-to-use engines to release hundreds of executable versions of a basic malware. All these malware programs are pursuing a single malicious goal - but from different channels. This is why obfuscation and metamorphic engines play a critical role in the development of a large number of perilous malware programs in today's IT environment [4]. Consequently, there is a never-ending battle between malware developers and security analyzers, which is evolving as rapidly as the complexity of malware advances [5].

The focus of our study is on malware programs with three main characteristics. They are: (a) rare (b) created with customized packers (c) extremely obfuscated and intelligent. These types of malware programs have not been discussed in most related studies in the area of malware behavioral modeling and detection. Hence, this paper is focused on (a) unpacking and executing the malware (b) extracting necessary features (c) evolving possible mutations to train a classifier that would be able to accurately detect such malware. Therefore, this paper argues that an effective malware detection strategy can be defined using a combination of a concise feature extraction method and creation of optimized datasets using a modified genetic algorithm.

This paper is organized as follows. The main problem and underlying challenges of malware modeling and malware detection are described in Section 2. Section 2 also provides reviews of relevant key concepts and technologies. Section 3 includes malware behavior analysis and a review of related works. Section 4 provides a detailed description of the authors proposed method for extracting features, and generating and optimizing required datasets for training a malware detection model to detect activities related to metamorphic malware. Section 5 evaluates and discusses the performance of our proposed method by estimating the quality of the generated and optimized dataset used for the data mining process. Finally, Section 6 forms the conclusion.

II. STATEMENT OF THE PROBLEM

Besides the significant growth of malware, malware have been created to be more intelligent, complex, targeted, and shrewd [6]. They have been equipped with a wide range of techniques to deceive and evade any antivirus scanners and mislead analyzers. The underlying reasons for these abilities are described next.

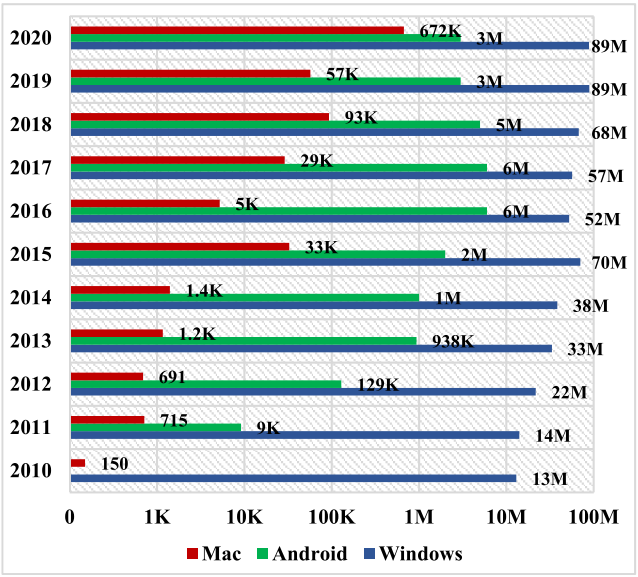

FIGURE 1. The amount of malware produced for different platforms in the last decade.

A. PACKING

Packing is a popular method malware producers may use to conceal malicious activities. Packing malware conceals most of the software attributes in a portable executable (PE) structure that may be targeted in the static anti-malware analysis. This may include API names, names and attributes of the imported libraries, and the control flow graph (CFG). Hence, static anti-malware scanners which ignore the execution phase and only check the source code [7] would not recognize the true nature of the file, therefore the malware passes the detection phase. Figure 2 shows a targeted malware sample where the names of API calls in the import address table (IAT) were encrypted by an (unknown) packer. The structural information of this malware has been extracted using the open-source tool Detect It Easy.[1]

The packing status of a piece of malware can be obtained by measuring the entropy of the code. A drastic change in the entropy of machine code created by standard compilers indicates that the file has been packed. Many types of packers such as UPX,[2] FSG, Yoda's,[3] ExeStealth,[4] PETite,[5] ASPack,[6] UPack, and VMProtect[7] can be detected by looking for a unique sequence of byte codes [8], though the unpacking patterns and decryption keys are not detectable using this method. The complexity and ambiguity of malware

[1] https://github.com/horsicq/Detect-It-Easy
[2] https://upx.github.io/
[3] https://sourceforge.net/projects/yodap/
[4] http://www.webtoolmaster.com/exestealth.htm
[5] https://www.un4seen.com/petite/
[6] http://www.aspack.com
[7] https://github.com/eaglx/VMPROTECT

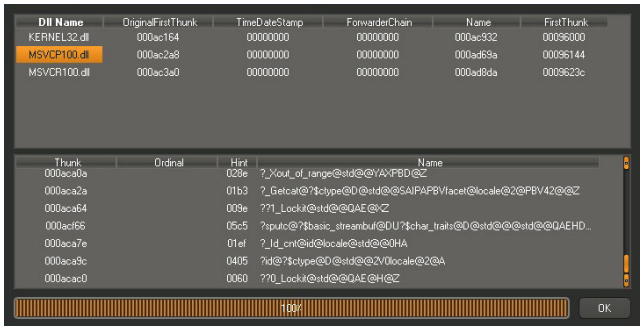

FIGURE 2. Encryption of system API calls of obfuscated malware extracted by detect it easy.

activities prompts analysts towards dynamically monitoring and analyzing malware's behavior in order to discover the true nature of obfuscated files. No matter how complicated the obfuscation behavior coded into the malware to conceal its malicious behavior during its static phase (stored on a disk), it has to unpack and decrypt what is hidden inside in the execution phase to be able to use system resources and deliver its requests to the OS. Therefore, it is possible to determine the behavior of an anonymous file based on its interactions with the OS by closely monitoring the allocation of system resources during the program's execution phase. To prevent the malware from doing damage, the anonymous file is executed inside a risk-free, isolated environment such as a sandbox. Here system API calls are monitored during the actual execution of the program. However, the difficulty of implementing a sandbox environment with proper virtualization and resource simulation is the major challenge of this method [9]. Symbolic execution is another strategy that can be employed. This method symbolically executes an anonymous file under the environment of a processor emulator and abstractly interprets the behavior of the file through symbol analysis [10]. The majority of dynamic behavior analysis methods are based on the two techniques just described. Our proposed solution is based on dynamic behavioral analysis using a risk-free sandbox to extract necessary features.

B. OBFUSCATION AND METAMORPHISM

Depending on the obfuscation technique used by a given malware, various challenges need to be addressed after the unpacking process. Various obfuscation techniques could be used including random dead code and unconditional branch insertion [11], dynamic code loading [12], identifier renaming, instruction reordering [13], runtime code generation (RTCG) and integration [14]. Each of these techniques should be addressed in a different way.

Metamorphic engine technology provides a platform to create new generations of one or a set of basic malware programs. It provides an easy platform for creating new instances of malware. One solution to deal with this threat is to identify the class of malware generated by well-known existing metamorphic engines. Although metamorphic engines try to create generations with characteristics different from the original, they cannot prevent passing down commonalities to new generations. Therefore, the similarity between generations will never reach zero. Some of the most famous engines of this sort are PS-MPC, G2, MPCGEN, NGVCK, and VCL32 [15]. P. Desai conducted a study showing that the minimum, maximum and average similarity level in the structure of a malware program compared to a benign application is 14%, 93%, and 35% respectively [16]. In order to deceive malware detection methods, metamorphic engines also try to increase the similarity of the newly generated malware with benign entities. By examining the malware produced by these engines and comparing them with basic samples, e.g. samples that have not been affected by the metamorphic process, it is possible to detect the utilized metamorphic pattern to some extent and provide solutions for the normalization of the malware produced by such engines. This is mainly made possible by using pattern recognition algorithms. However, the drawback of using these engines is the size of the binary code, which is increased due to the injection of junk code with false conditions, permutation of instructions, and/or the redundancy of a metamorphic stub in the generated malware's file. As an example, in [17] a code emulator tool has been specifically designed to detect the inserted dead code in every malware file. The resulted data from the code emulator is used to improve HMM-based detection algorithm for detecting metamorphic viruses produced by such engines.

Polymorphic engines transform the structure of malware program to a new version. This generates a new behavioral signature every time the malware executes. Therefore, it follows a different path to achieve its purpose. This relates to the fact that any detection strategy should expect new unknown behavior from newly generated malware programs. To make necessary changes, malware needs to execute the self-modifying stub embedded in the body of its executable file during each run. The existence of the stub is detectable by calculating the starting address of the original entry point (OEP) of the executable code, which can be obtained by adding two values: the image base address and a variable value retrieved from the OS version. Standard compilers put the OEP address at the beginning of the *.text* section, while the malware must execute its own metamorphic stub before executing its original code. Therefore, the OEP value in such malware is reset to the first line of the metamorphic stub code [18]. The complexity here is that the mere detection of the existence of a metamorphic engine cannot properly detect the type of malware generated by that engine. Intelligent malware programs also utilize artificial intelligence strategies such as the method described in [19]. These methods use deep and reinforcement learning through interaction with anti-malware programs. The ability to predict and detect future mutations of the malware is a necessity to create robust and reliable malware detection modules. This challenge has been precisely addressed in our proposed method by producing a dataset containing potential mutations of malware.

C. SCARCENESS OF TARGETED MALWARE

Creating high-end targeted malware programs with the mentioned capabilities requires proper funding and a high-level of technological support. These malware are designed and developed in a very complex way to sabotage specific targets such as modern weapons of cyber warfare [20]. This is why these types of malware are almost impossibly difficult to detect. The small number of high-end targeted malware always has been a big challenge when attempting to study their behavior and when tailoring defense and detection strategies against them. Implementing intelligent diagnostic models to detect them, particularly model training, would not be possible when there is not a sufficient dataset. This argument is valid for different categories of malware programs such as rootkits, exploiters, evaders, file-less malware, and particular classes of spyware and malware that are specifically designed to work at the level of the master boot record (MBR) and hypervisor.

Bootkit malware is capable of infecting the MBR loads prior to the OS startup process to control the operating system and modify drivers before anti-malware scanners start running. This is the most potent method for attackers to silently infiltrate a targeted machine [21]. Another rare malware is Blocker ransomware which disrupts the communication between the OS and input peripherals at the kernel level by hooking interrupt service routines (ISRs) [6]. This type of hook modifies the addresses in the interrupt descriptor table (IDT). The kernel-level rootkit malware family is another example, and is designed to hide traces of other malware, including running processes, files stored on the hard disk, registry keys, and the names of the registered services. Exploiter malware is another category that is programmed to be aware of OS zero-day (unpublished) vulnerabilities and uses them to bypass security mechanisms in the OS including kernel path protection (KPP) and user access control (UAC), to undertake their malicious activities. The File-less malware class is stored in the main memory space and is disposable. Its main target is servers in data and service centers, which run constantly, for a long period of time. Evader malware programs are able to assess their environment and recognize analysis environments, such as sandboxes and virtual machines (VM), and also decoy environment like a honeypot. Collecting this information helps them to avoid the execution phase in such monitoring and/or analysis environments. Finally, there are Malware programs that function at the ring-1 (hypervisor) level and have the ability for performing Hyperjacking attacks in the IaaS cloud environment, like a 'blue pill'. The most important classes of rare malware, which are studied in this paper, have been tabulated in Table 1.

Malware programs can be very deceptive and stealthy. In order to illustrate this we will outline a sample scenario of a spyware attack that has been designed and built quite purposefully. The spyware is transmitted to the victim's system through a known zero-day vulnerability or a backdoor created by a piece of malware that has already infected the victim's system. The malware silently launches itself by finding the right condition through monitoring the environment. The existence of this malware is concealed by a rootkit malware during its runtime so that anti-malware scanners or analysts are not able to detect it. The malware is also able to assess the environment to determine if a monitoring tool is running so it does not reveal its malicious behavior. It waits for a suitable time to send the stolen data to its command and control (C&C) center. C&C is a node dominated by a malicious attacker to send commands to the compromised computer by malware. It also receives the hijacked information from the victim's system through a covert channel using authorized network protocols such as ICMP, The data is combined with regular network traffic in order to avoid detection. To prevent any forensic analysis, the malware then receives the desired commands from the C&C to remove its footsteps and commit suicide after completing its mission.

TABLE 1. The rare classes of malware.

Class	Main Behavior	Ref.
Exploiter	Combine zero-day exploits with malware or ransomware	[22]
Stealth Spyware	Stealing valuable data in the presence of AVs and knowing unpublished vulnerabilities	[6]
Kernel Rootkit	Employing direct kernel object manipulation (DKOM) and IRPs to hiding other malware	[23]
Injector	Compelling other running process to execute its malicious commands by code/dll injection	[24]
Blocker	Halting the OS by hooking ISRs	[6]
Bootkit	Infecting MBR to load before the OS and control OS services	[25]
File-less	Having no executable file and rest just in the main memory space of the OS	[26]
Evader	Applies anti-debugging techniques and ignore executing in VM/Sandbox/ Honeypot	[9]
Hyperjacking	Hijacking hypervisor or VMM at ring -1	[27]

Undoubtedly, designing such a multifaceted malware, e.g. a malware program that consists of more than an executable file and running process simultaneously, requires plenty of financial support and time. Therefore, accessing their executable files for behavior extraction is quite difficult and expensive. Researchers continually face restrictions in developing strategies to detect and deal with these types of malware. Extracting features of these classes of malware and training a classifier to detect them, and predicting the next future mutations are some important contributions of this paper. This will be outlined in the following.

D. CONTRIBUTION OF THE PAPER

Data scarcity of targeted malware, which is a severe restriction when machine learning and data mining algorithms are adopted for malware detection, is not the only challenge.

As described, the relevant executable files are extremely obfuscated and encrypted using anonymous obfuscators and customized packers. This makes the unpacking process almost impossible using conventional methods [28].

Our proposed method provides an architecture for unpacking and extracting necessary features from targeted malware. After the feature extraction phase, a genetic algorithm is used to generate the essential dataset and then optimizes it to train a behavioral model for detecting rare classes of malware as well as predicting their potential mutations in the future. The contribution and innovations of our proposed method can be summarized as follows:

- ✓ Extracting features from extremely obfuscated and intelligent malware.
- ✓ Solving the problem of sample scarcity for training models, particularly when rare classes of malware are considered, through generating new and qualified populations.
- ✓ Higher prediction accuracy of the behavior of metamorphic malware in future mutations using the optimized dataset through a deep learning algorithm.
- ✓ Understanding the behavior of the compound malware, which is formed as a crossover between different components, each belonging to a specific class.
- ✓ Reducing the difficulty of detecting malware programs that have been created with various metamorphic technologies through crossover between the chromosomes of old-released and newly released malware.
- ✓ Maintaining a balance between the classes of rare malware and other common malware in model training which results in a more robust and accurate model.

III. RELATED WORKS

The most important relevant studies on malware detection are reviewed next, based on the behavior analysis strategy. Then, details of our proposed strategy will be provided.

Over recent years, dynamic behavior analysis has become a key strategy in detecting new and obfuscated malware programs during their execution phase. Machine learning and data mining techniques are popular methods for malicious behavior modeling and malware detection according to the literature [29]. However, dynamic strategies are empowered by static methods to improve their detection capabilities [30]. Modeling malware activities is conducted based on several features such as system API calls, values of registry keys, opcodes, and power consumption. Artificial intelligence and machine learning techniques are employed. These include neural networks [31], deep learning [32], image processing [33], reinforcement learning [34], and ontology [35]. Trivial classifiers such as k-nearest neighbor (KNN), naive Bayes (NB), decision tree (DT), support vector machines (SVM), and hidden Markov model (HMM) were also widely utilized in the modeling stage.

Extracting influential features for training a model is another critical part of a malware detection software. Different studies used alternate methods for feature extraction as well as various classification algorithms. The first part of this section provides details of studies that have utilized the malware analysis procedure to extract features at first, then have attempted to address the problem of malware classification. The process of investigating malware programs to realize their functionality, discover the source of propagation and possible impacts is described as malware analysis. This process is vital for any infrastructure in order to respond to cybersecurity attacks and incidents [5]. The second part of this section includes related works that have only focused on malware classification and have not been engaged in malware analysis in order to extract behavioral features. These works have used datasets that contain malware behavior in the form of PNG images or CSV files. Obviously, the first category is more challenging since various obstacles need to be tackled when facing packed, obfuscated, and metamorphic malware in the real world. Further, a precise report of malware behavior during execution is required to develop a removal tool in order to eliminate malware files and disinfect the OS by rolling back all malware activities. Our study is placed in the first of the two categories.

A. MALWARE ANALYSIS AND CLASSIFICATION

Mohaisen *et al.* [36] developed a malware detection method using dynamic behavior analysis. The authors also considered the possibility of manual detection of unknown malware. Four system resources, e.g. registry, memory, network, and file system, were used to extract relevant information for behavior modeling. The model was trained on 4,000 samples and tested on 115,000 samples to prove the scalability of their tool for implementing in the edge of the Internet for enterprise and industry. The best result for recall and precision of the model was reported as 99.6% and 99.5%, respectively.

Imran *et al.* [37] established a hybrid method of sequence classification based on HMM and similarity-based methods for training and classification of malware. Their hybrid method creates a symbolic reference to 120 system API calls from 20 categories. Evaluation of different classifiers in the similarity-based method demonstrated that a random forest classifier performs better than other classifiers on malware similarity vectors by a precision of 88%.

Liu *et al.* [38] presented a new malware detection method based on machine learning. Using the n-gram pattern of the malware opcode, they created black-and-white images, which were applied to extract features for clustering the anonymous malware based on the shared nearest neighbor (SSN) algorithm. In order to assess their model, the authors used a dataset of 20,000 samples, including a mix of malware samples and benign files. Their results showed that the best accuracy for classifying anonymous malware was 98.9%. Also, the average accuracy of their method for the detection of modern malware samples was reported as 86.7%.

Javaheri *et al.* [6] developed a method to detect and remove spyware, including keyloggers, screen recorders, and blockers. This method introduced an efficient architecture for kernel level tracking and dynamic behavior analysis.

The reported performance evaluation demonstrated that their method accurately detected more than 92% of the malware and could successfully remove 81% from the OS. According to the literature, this reported study is the only one that provides a method for disinfection of malware on top of detection and classification.

Khan *et al.* [39] used GoogleNet and ResNet models from two different platforms in their work to identify new obscure classes of malware by using deep learning techniques through image processing. Microsoft datasets, including nine classes of such malware with a dataset of benign files, were used to train and validate the model. The precision was evaluated at more than 74% and 88% on GoogleNet and ResNet models respectively. The experiments indicated that ResNet152 has the highest accuracy while GoogleNet has the best execution time.

Image processing techniques were also used in the study conducted by Vinayakumar *et al.* [32]. The authors proposed a hybrid method using deep learning to detect zero-day malware. The authors claimed that their introduced architecture worked better than traditional learning algorithms. The innovation of their approach is in combining machine vision and deep learning algorithms. The proposed detector can also be used with big data for real-time malware detection.

Wang *et al.* [40] present a malware detection method called LSCDroid for realizing the malware's intention through behavioral analysis for the Android platform. The proposed method used sequences of local sensitive API invocation and function-call graphs to recognize the behavioral pattern of malware through manually static code analysis. A machine learning algorithm was then used for malware classification. The results indicated that the precision of this method for malware classification was more than 96%.

Makkar *et al.* have proposed a machine learning framework [41] using five different models based on refined input features to address security concerns in IoT devices, including spam detection. The authors used the REFIT smart home dataset to validate their proposed method. The evaluation indicated that the accuracy of the proposed method was between 79.8% and 91.8% for five different learning models.

B. MALWARE CLASSIFICATION

Kalash *et al.* [31] proposed a deep learning framework for malware classification rather than shallow learning algorithms. The authors adopted a deep learning approach using convolutional neural networks (CNN) to classify malware programs. Performance evaluation of this strategy showed that this method resulted in 98.52% and 99.97% accuracy for the Malimg and Microsoft datasets respectively. The authors claimed that there is a higher detection ability for deep learning when compared to traditional shallow learners such as SVM.

Roseline *et al.* [33] presented a method for visualizing malware within 2D images. They used a random forest feature selection strategy to detect dominant features and then employed deep learning for the classification phase. The accuracy of malware detection and classification tasks for this method was estimated between 97.2% and 98.6% for Malimg, BIG 2015, and MaleVis malware datasets respectively.

A deep learning method using convolutional neural network was employed by Kumar *et al.* at [42] to identify unknown malware programs. The authors visualized malicious codes in the form of gray-scale images to address the challenge of malware identification and classification. Dataset of Vision Research Lab, including more than nine hundred samples from twenty-five groups in addition to three hundred benign files, were used in this study. The experiments demonstrated an accuracy of 98%.

C. COMPARISION OF RELATED WORKS

Table 2 compares related studies in terms of the type of detection strategies and other attributes such as platform, the type of the behavioral analysis and feature selection (if utilized). The table also includes the dataset, scale, classification algorithm and also the accuracy of the model. According to the literature, targeted samples and datasets of malware obfuscated by anonymous obfuscators and packed by customized packers are not easily available, and the majority of the related studies merely focused on available datasets in their model generation and performance evaluation. This shows that limited access to rare classes of malware has led to neglecting these classes in the area of malware detection. This is because model training with a limited number of samples is not robust, accurate, and reliable; models create by limited data cannot appropriately fit the problem search space and cover all possible ways that a malicious programmer may use to implement the malware.

IV. THE PROPOSED METHOD

This section describes our proposed method for detecting rare classes of malware as well as possible and potential future mutations caused by metamorphic engines. Many of the malware classes we focused on are extremely rare and obfuscated. We were able to collect and acquire some relevant data for these classes from Adminus [43], VirusSign [44], and VirusShare [45] malware datasets.

To create the initial population, we randomly chose samples from seven classes of rare malware, including stealth spyware, kernel rootkit, injector, blocker, bootkit, evader, and file-less. All samples from the early population malware were extremely packed and protected by robust packers mentioned in Subsection A of Section 2, in addition to several unknown packers to prevent detection and disclosure of their action mechanism. The size of the initial population is not large enough to enable accurate and robust classification. This problem was addressed by generating a new dataset so that the size and quality of the population would become sufficient for reliable training. In the following subsection, we explain how to extract behavioral features for the modeling stage.

TABLE 2. Comparison of related works.

Reference | Year	Malware Analysis			Malware Classification			
	Platform	Behavioral Analysis Type	Feature Selection	Dataset	Scale	Algorithms	Accuracy
[36] - 2015	Windows	Dynamic	Automatic system resources	AutoMal	4K-119K	SVM, DT, Regres., Perceptron, KNN	86-99%
[37] - 2016	Windows	Dynamic	Automatic system calls	Malheur	9K	HMM, RF, J48, SVM, NB, KNN	88%
[38] - 2017	Windows	Dynamic	Automatic Opcode & import functions	ESET NOD32 Anubis	20K	SNN	86.7%
[31] - 2018	N/A*			Malimg Microsoft	9K	CNN	98-99%
[6] - 2018	Windows	Dynamic	Automatic API & sys calls	Adminus VirusSign	8K	Linear Regression, JRip, J48	93%
[42] - 2018	N/A			Malimg	12K	CNN	98%
[39] - 2019	Windows	Hybrid	Opcode	Microsoft	10K	CNN	74-88%
[32] - 2019	Windows Linux Android	Hybrid	?	Malimg	9K	DNN	90-91%
				VirusSign VirusShare	15K	CNN	93-96%
[33] - 2020	N/A			Malimg MaleVis Microsoft	9K	Random Forest	97-98%
[40] - 2020	Android	Static	Manual API & FCG	Andro-dumpsys Algenome	7K	J48, RF, SVM, Logistic Regression	96-98%
				FalDroid	8K		
[41] - 2021	IoT	Dynamic	15 features	REFIT	100K	Bag, bayesglm, BstLm, xgbLinear, glm-StepAIC	79-91%
The proposed method - 2021	Windows	Hybrid	Automatic API & sys calls	Adminus VirusShare VirusSign	12K	Genetic Algorithm, JRip, J48, DNN	94-96%

* Malware analysis was not discussed in these references, they have merely focused on malware classification.

A. MALWARE BEHAVIORAL FEATURE EXTRACTION

In this work, we extracted behavioral features and modeled the behavior of highly protected and obfuscated malware by dynamically unpacking them and tracking their system API calls at the kernel-level for Microsoft Windows OS. This process was performed by installing tracker hooks. Dynamic unpacking was performed by deceiving and executing malware in an isolated environment and eventually dumping the process memory at an appropriate time and rewriting it in the form of an executable file - based on the method explained in [28]. This reference provides an accurate method for dynamic unpacking based on kernel-level memory dumping. Success in the unpacking process is essential for extracting behavioral features since it has a direct effect on the accuracy of model training for detecting metamorphic malware. After finalizing the unpacking process, attributes necessary to model the malware behavior - including the names of the API calls and libraries - are extracted by parsing the PE header. It should be noted that the names of the API calls stored in the header of the PE file alone do not provide enough information to infer the actual behavior of this class of malware. The reason is that many function names may be unrealistic and might have been inserted in the IAT or the export address table (EAT) of the PE file to confuse scanners to be passed. These names do not necessarily refer to the functions invoked by a given malware in the execution phase. Moreover, some novel intelligent malware programs use dynamic programming and runtime code generation methods, making it impossible to extract their behavior without real execution. Thus, the malware executable files are executed in a virtual machine environment equipped with tracker hooks. In this environment, the chains of the relevant system API calls are also recorded. Tracker hooks are created by installing a Kernel-Mode Driver Framework (KMDF) filter driver [46] in the I/O stack space of the OS with group ordering and a very precise offset from the base address to track the malware input/output request packet (IRP).

Our strategy was designed in such a way that by following a logical and structured procedure, the precision of the analysis and the degree of transparency of the analysis environment are maximized. This is because the performance of detecting intelligent malware is critically dependent on extracting behavioral features [47]. The architecture and workflow of our proposed method are illustrated in Figure 3. As shown, the process begins by delivering a malware file collected by a honeypot, or from a valid source, to a static analysis module. In the first step, the necessary information for behavior analysis, such as section properties and the

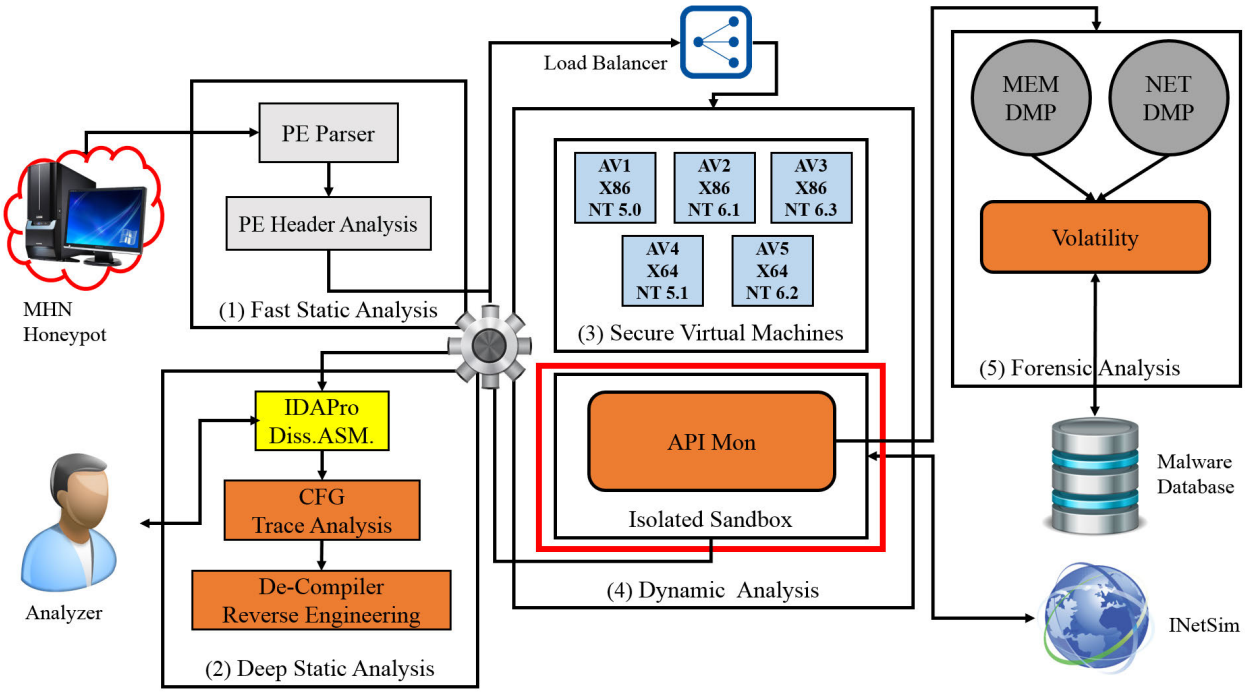

FIGURE 3. The architecture of the proposed method for the malware behavior extraction.

information related to the IAT and EAT tables, are extracted using a PE structure analysis tool. The packing status of the malware file is determined by calculating the entropy for each section of the file. If the file is packed, it is not possible to extract many properties, and this step must be repeated after the unpacking process. In the second step, a deep statistical analysis is performed on the malware file. At this stage, the relevant assembly binary code is extracted from the executable file and then a corresponding control flow graph is generated using disassembler tools or human analysts, if required. This step concludes by translating the assembly code into a high-level language such as pseudo-C using a decompiler. The extracted information from step one and step two, including PE header properties, CFG, and the pseudo-C code, is then stored in a database, and the malware is transferred to the third and fourth steps in order to finalize the dynamic behavior analysis. In the third step, a multi-scanner system scans the malware file to detect and approve their predicted class. If the malware was merely detected and not classified correctly, it is not possible to clarify that the result is a true positive or a false positive. Our multi-scanner consists of twenty different antivirus programs, all installed on VMs running different versions of the Microsoft Windows OS kernels. Table 7 includes the configuration of our multi-scanner system used for labeling malware samples for conducting a supervised learning process. In the fourth step, the malware file is delivered to an isolated VM-based sandbox environment in which the system routine observer hooks are installed. The malware file is automatically executed by a launcher, and its dynamic behavior log file is created and recorded. This step of the analysis was automated as it is a time-consuming process, especially for a large number of malware samples. In order to accelerate the analysis, 30 VMs were used for parallelization, where a load balancer was responsible for dispatching tasks and managing load balancing among VMs. Eventually, in the fifth and last step of malware analysis, the malware's dedicated memory was extracted and dumped three times using the method described in [28], and the values were rewritten and aggregated in a new file which was further used an input for a Volatility 2.5[8] memory forensic tool. The new malware file contained the necessary unpacked parts, including PE header, Data Directories, *.text* section, IAT, and EAT tables, used for a statistical analysis. In the case of file-less malware, for which only the memory dump is available, the process of analyzing and extracting the behavior begins from this step.

The need for communicating and interacting with the outside world (for example, interaction with the C&C) through the network when the malware is running inside the virtual machine environment is an important point that must be carefully addressed. Using the open-source tool INetSim, a simulated Internet network was provided to the malware so that its requests for a network connection can be handled in a controlled manner. Another important issue is the malware's need to interact with the user through peripheral devices such as a mouse or keyboard (for example, a user entering a password using a keyboard). Given the fact that the analysis process for today's large amount of malware is performed

[8] https://www.volatilityfoundation.org/releases

automatically, a tool mimicking human behavior (Monkey) was designed and embedded in the VMs in order to address this challenge.

B. GENETIC ALGORITHMS FOR METAMORPHISM

The Genetic Algorithm (GA) is one of the most popular evolutionary algorithms to generate high-quality solutions for optimization problems. GA was inspired by the Darwinian evolutionary theory where every solution maps to a chromosome, and each parameter indicates a gene. GA evaluates the quality of each generation using fitness functions to improve iterative solutions [48]. GA has been utilized in a wide range of domains and applications such as disease diagnosis, cancer prognosis, abnormality detection in medical images, stock price prediction, particle swarm optimization, predicting energy consumption, spatial modeling of climate data, fault diagnosis [49], and other problems in the area of optimization algorithms.

Over the recent years, variants of genetic algorithms have been adopted to address a wide range of problems such as hierarchic genetic scheduler (HGS), smart genetic algorithm (SGA), panmictic genetic algorithm (PGA), interval genetic algorithm (IGA), and hybrid genetic algorithm (HGA) for scheduling tasks workflow, especially in the cloud computing [50] or quantum genetic algorithm (QGA) for quantum computing [51].

This work utilizes GA to generate and optimize a dataset of rare and metamorphic malware in order to increase detection accuracy.

C. MALWARE BEHAVIORAL MODELING

After the initial population was formed and their behavioral features were extracted, the system API calls and their parameters were stored in a database. This database contains the log file of each malware behavioral features. In our proposed method, each chromosome represented the behavior of malware in which each system API call is mapped to a gene together with its effective parameters. Each genome is a sequence of corresponding system API calls that represent a specific malicious behavior. Genes are sorted based on the time sequence in which their corresponding system functions are called and arranged in a chromosome. Therefore, the length of each chromosome is equal to the number of invoked system functions. The time sequence of calls is determined using a timestamp. The encoding modes in the proposed method are performed in a 10-bit binary system. A total number of 1024 genes or 1024 corresponding system calls were defined, which is sufficient for the main kernel-level system calls of the Microsoft Windows OS (NT 6.1+) (which has about 700 system calls available) [52]. The major challenge in gene encoding is to find the addresses of system functions. For this purpose, the system service descriptor table (SSDT) at the OS kernel mode was used. This table holds the addresses of 400 system functions [33]. The addresses of the remaining functions, mainly used for graphical operating system interactions, are stored in the SSDT shadow table. The addresses of the system functions at the kernel level were determined using Equation 1.

$$Original\ Add. = ServiceTableBase + SysEnterAdd. \times 4 \quad (1)$$

where *ServiceTableBase* is the beginning address of the SSDT table and *SysEnterAdd* indicates the address of the *SysEnter* command, which maps the user-level APIs onto kernel-level APIs in Microsoft Windows NT OS Family [53]. The encoding is performed based on addresses of the functions in the OS kernel, therefore, their assembly equivalence can be searched in the unpacked version of the malware file. It is important to note that the addresses differ in different versions of the OS kernel. In addition, because of the randomization of the addresses by the address space layout randomization (ASRL) and KPP security mechanisms, a fixed address for some functions cannot be found (mainly the functions of the sensitive modules, which start with *Nt*) using the tools provided with the OS, such as the *GetProcessAddress* routine call. In order to resolve this problem, a unique sequence from the body of the desired function, known as function ID was looked up. It is possible to find an ID by debugging the OS kernel with tools such as Windbg and Ollydbg. Refer to [52], where some of these IDs are provided. Table 3 contains the encoding of the proposed method for 45 different functions. These are the functions with significant effects on malware's behavior. This is considered one of the key findings of this study.

In our proposed method, system functions (mainly the kernel-level system functions) were used to model the behavior of malware programs. The reason for this choice is that, in most cases, a certain number of user-level system functions are mapped to a smaller number of equivalent functions at the kernel level. Similarly, kernel-level routines are mapped to a smaller number of corresponding IRPs; the IRPs are mapped to a smaller number of (ISRs), and they are further mapped to a smaller number of relevant interrupts [28]. Therefore, from the user level to the kernel level and from the kernel level to the hardware, the abstraction of functions increases. It therefore becomes more difficult to interpret the resulting behavior as well as the implementation of tracker hooks. As for the user-level functions, although these functions are less abstract and easier to implement, due to the large number of functions of the state space, the problem of finding optimal solutions becomes an instance of a NP-hard problem. Generally, the kernel-level system functions can be considered as middle-state functions, so they create the most appropriate zone for modeling the behavior of a given malware.

After extracting the names of the system functions used by a given malware, the malicious behavior based on the order of the system API calls and their call frequencies can be modeled, and the corresponding chromosomes can be formed. We used linear regression to model both the behavior and chromosome formation. The chromosome of malicious behavior is formed according to:

$$Chr._1 = A \times X_A + B \times X_B + C \times X_C + \ldots \quad (2)$$

TABLE 3. Encoding for chromosome formation in the proposed method.

API Name	Module Name (OS Version)	Memory ID (ring 3)	Access at Kernel (ring 0)
NtAdjustPrivilegesToken	ntoskrnl.exe (6.1)	0x000c	__asm{mov eax, 0x82C2785}
NtAlertResumeThread		0x000d	__asm{mov eax, 0x82CF6979}
NtAllocateVirtualMemory		0x0013	__asm{mov eax, 0x82C65E0F}
NtClose		0x0032	__asm{mov eax, 0x82C6109C}
NtCreateEvent		0x0040	__asm{mov eax, 0x82C7B5D9}
NtCreateFile		0x0042	__asm{mov eax, 0x82C64E82}
NtCreateProcess		0x004f	__asm{mov eax, 0x82CF4E07}
NtCreateThread		0x0056	__asm{mov eax, 0x82CF4C0E}
NtGetNextProcess		0x008b	__asm{mov eax, 0x82CF6B70}
NtGetNextThread		0x008c	__asm{mov eax, 0x82CADA82}
NtLoadDriver		0x009b	__asm{mov eax, 0x82BBB279}
NtOpenFile		0x00b3	__asm{mov eax, 0x82C945C4}
NtOpenProcess		0x00be	__asm{mov eax, 0x82C9BF31}
NtOpenSection		0x00c2	__asm{mov eax, 0X82C991BA}
NtOpenThread		0x00c6	__asm{mov eax, 0X82C99E88}
NtQueryAttributesFile		0x00d9	__asm{mov eax, 0x82C7CB48}
NtQueryDirectoryFile		0x00df	__asm{mov eax, 0x82C9462F}
NtQueryDirectoryObject		0x00e0	__asm{mov eax, 0x82CA287A}
NtQueryObject		0x00f8	__asm{mov eax, 0x82C26CB1}
NtQueryVirtualMemory		0x010b	__asm{mov eax, 0x82C93003}
NtReadFile		0x0111	__asm{mov eax, 0x82C69858}
NtReadFileScatter		0x0112	__asm{mov eax, 0x82BCF286}
NtReadVirtualMemory		0x0115	__asm{mov eax, 0x82C9DB79}
NtResumeProcess		0x012f	__asm{mov eax, 0x82CF6913}
NtResumeThread		0x0130	__asm{mov eax, 0x82C8BFCF}
NtSuspendProcess		0x016e	__asm{mov eax, 0x82CF68B3}
NtSuspendThread		0x016f	__asm{mov eax, 0x82CB3650}
NtTerminateProcess		0x0172	__asm{mov eax, 0x82C7BB3D}
NtTerminateThread		0x0172	__asm{mov eax, 0x82C8E8E4}
NtWriteFile		0x018c	__asm{mov eax, 0x82C580B4}
NtWriteFileGather		0x018d	__asm{mov eax, 0x82BCF7DE}
NtWriteVirtualMemory		0x018f	__asm{mov eax, 0x82CA15B5}
NtUserBlockInput	win32k.sys (6.1)	0x1141	__asm{mov eax, 0x925B5C7E}
NtUserGetAsyncKeyState		0x1192	__asm{mov eax, 0x924DA1DB}
NtUserGetDC		0x11a4	__asm{mov eax, 0x9251CBEB}
NtUserGetKeyBoardState		0x11b2	__asm{mov eax, 0x925BC7C4}
NtUserGetKeyState		0x11b4	__asm{mov eax, 0x924E6291}
NtUserGetRawInputData		0x11c0	__asm{mov eax, 0x925C8253}
NtUserGetDCEx		0x11a5	__asm{mov eax, 0x924EA7DA}
NtUserLockWorkStation		0x1201 - 0x1204	__asm{mov eax, 0x92554759}
NtUserRegisterHotKey		0x1273 - 0x1276	__asm{mov eax, 0x924D356D}
NtUserSendInput		0x1266 - 0x1269	__asm{mov eax, 0x925B9C9C}
NtUserSetWindowsHook		0x1230 - 0x1233	__asm{mov eax, 0x925C5759}
NtUserSetWindowsHookEx		0x122f - 0x1232	__asm{mov eax, 0x924F8513}
NtGdiBitBlt		0x100e	__asm{mov eax, 0x92534E07}

where A, B, and C are the system API calls, and X_A, X_B, and X_C are their repetition frequencies. $Chr._1$ is a behavioral chromosome formed based on a chain of system API calls and their repetition frequency. Two malicious chromosomes extracted from the code injector and self-propagating classes of malware are shown in Figure 4.

Chromosomes shown in this figure represent two types of malicious behavior: code injection and self-propagation. In the first chromosome, a sequence of nine genes represents the malicious behavior of injecting code into the memory of a running process. This behavior is used mainly for obtaining control of the victim process, including the victim's access privileges to misuse its signature for UAC. In the second chromosome, a sequence of seven genes indicates a self-replication behavior for another sample of the malware.

D. MUTATION IN THE MALWARE BEHAVIOR TO CREATE MALWARE DATASETS

In this section, we describe how the dataset was created containing malware with more stealth and destructive qualities. Through data mining, we conducted a careful examination of the structure and parameters of system functions in several chromosomes extracted from a primary population of malware samples. These samples were extracted from Adminus, VirusSign, and VirusShare malware datasets. We found that it was possible to replace some genes with other corresponding

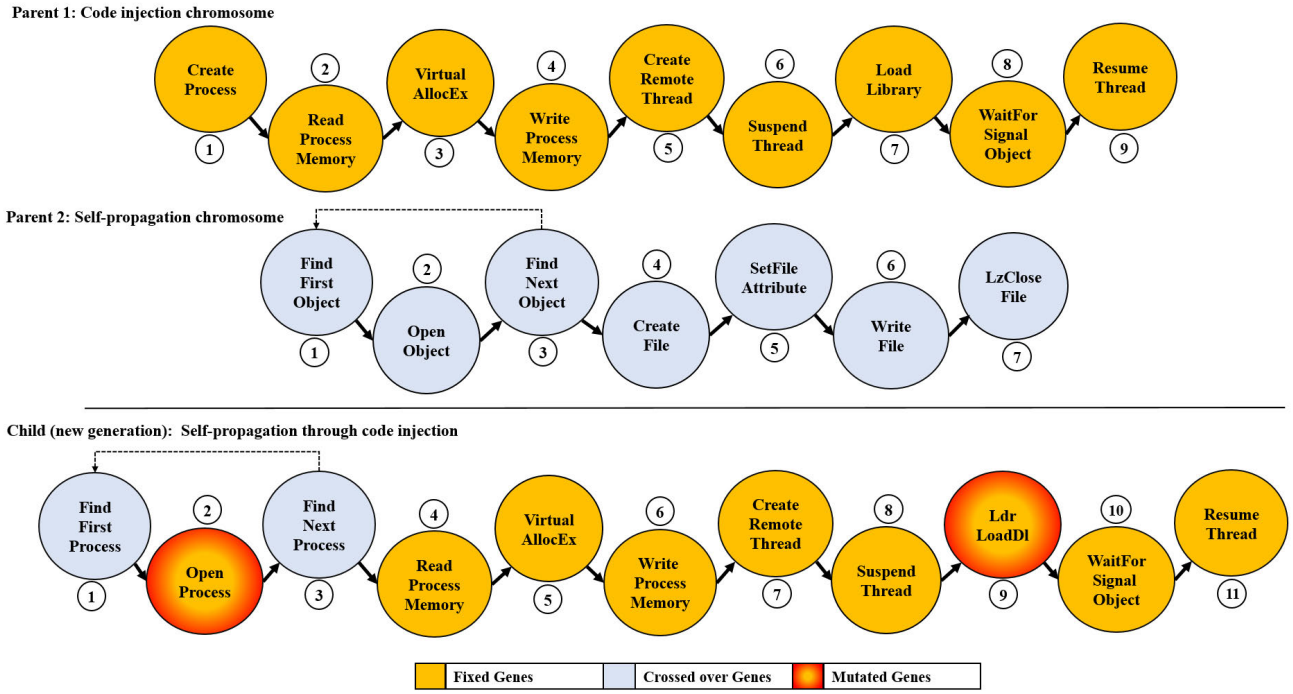

FIGURE 4. Chromosomes of malicious behavior for two samples of parent malware and their mutant child.

system functions to cause mutations. For instance, in the first gene of a chromosome, code injection could create a handle to a running process using the *CreateProcess* system function instead of the *OpenProcess* function, which is considered as a rule in changing the malware gene and inducing mutation. The other mutation could replace the *LdrLoadDll* function with the seventh gene in the same chromosome, i.e. *LoadLibrary*, which is used to load a binary file, usually a Dynamic Link Library (DLL), into the memory of the victim process. Also, for the self-propagating behavior chromosome, the second gene, i.e. *OpenObject*, can be mutated by substituting the *CreateFileA/W* and *WriteFileA/W* system functions. A new chromosome is thus created with mutated genes that pursue the same purpose as previously but in a slightly different manner. The mutations induced by changing the function codes also changed the malware signature. Again, it was possible to induce a newer mutation within the new generation chromosome in the gene corresponding to the *ReadFile* and *WriteFile* functions by replacing them with the *NtReadFileScatter* and *NtWriteFileGather* functions, respectively. This process may continue as long as the rules of the knowledge base (KB) can find the possibility of non-repetitive mutations in malware chromosomes. This KB includes all possible states for gene substitution. This KB supports a rule-based module based on JRip algorithm to define and manage the cross over function between chromosomes [54]. It should be noted that these functions have not been documented by the manufacturer (the Microsoft Corporation) as they are highly sensitive. The structure of this function and some other undocumented system functions were derived from [55].

The crossover process in malware genes is more complicated. In this case, at least two basic samples of malware were required for gene crossovers. For instance, in extracted chromosomes, a crossover operation can be defined as an insertion of the first and third genes of self-propagation behavior into the genes for the code injection behavior. This created a new compound malware that exhibited its self-propagation behavior by code injection. Therefore, the new generation of malware emerged from the crossover of the two chromosomes that inherited features from each parent. This new malware can self-propagate by injecting code into the memory of the victim processes. The original purpose - i.e. self-propagation - was clearly achieved, but in a different manner. In either case the new generation behaved in a much more complex way than its parents. Moreover, its chain of calls is now longer as two genes are crossed over. The crossover and mutation process for the encoded genes of the mentioned samples along with the related flowchart are illustrated in Figure 5.

In the proposed method, the candidates for producing a new generation were chosen by an elite selection method from the initial population. Elite Selection is performed by maximizing the malignancy rate (MR) and minimizing the tracking rate (TR). The MR component indicates the malignancy of the malware behavior and is determined according to the method described in [56]. In our study, the malware was executed in a very advanced sandbox environment, and after assessing its behavior, the sandbox provided a complete report and accurate rate of malware malignancy based on the damage malware afflicted the system resources (according to YARA and Sigma rules). TR is the detection rate of the

scanners showing the percentage of the correctly classified malware classes. The malware file was scanned by over 70 anti-malware scanners available in [57] simultaneously. The TR was computed using the following formula:

$$TR = \frac{\sum_{i=1}^{N} AV\ (Classified\ Correctly) \times W}{\sum_{i=1}^{N} AV \times W} \quad (3)$$

As shown in Equation 3, TR is the ratio of the number of scanners that detect the malware and classify it correctly to the total number of scanners and their weights. N is the number of scanners and W indicates the weight of each scanner calculated according to the ranking list released by AV-Test in 2020 [58]. The list has been sorted by the measure of protection obtained during testing scanners by AV-Test.

In other words, an elite malware has the most malicious behavior when it is detected by the least number of anti-malware tools. Elites in each class of malware were selected as candidates for evolution and reproduction. Possible states according to rules of the KB were then applied to mutate and reproduce a new generation by substitution and one-point/multi-point crossover in the parents' chromosome genes. The quality of the new generation was calculated by a fitness function which was then compared with the parents' quality. The fitness function described in Equation 4 maximizes MR and minimizes TR.

$$\forall Class_i(Fit.Func._i) = Max\ (MR) \cap Min(TR) \quad (4)$$

If the MR value is more than those of the parents and the TR value less than those of the parents, the child was accepted and added to the new population. Otherwise, the child was removed and a new state of possible substitutions and crossovers created and tested. Consequently, each child chromosome is a behavior indicator of a new sample of malware, which is more destructive and more secretive than its parents. The termination condition in the proposed method was to achieve a certain rate of quality or to produce a certain number of entity samples. If all states for crossover or mutation in the malware genes are traversed, the algorithm also terminates. The following pseudocode shows how new generations are created according to our method.

The input for Algorithm 1 includes two samples of rare malware. On these, gene substitution and the crossover function were triggered to create new generations. Next, the fitness function measured the malignancy and tracking rate of new generations to determine if they qualify for selection and keeping. The result is a dataset with more stealth and therefore more destructive samples of malware.

Previous works have not discussed detecting rare and targeted malware equipped with complex packing and obfuscation techniques, nor the lack of sufficient training samples in any format (such as binary, assembly, source code, and image). Hence, this paper focusses on addressing this challenge. It is noteworthy that the method proposed in this paper was designed to produce new, high-quality entity samples to include in training and classification in order to accurately detect complex and rare classes of malware.

Algorithm 1 Generating and Optimizing Malware Dataset

Input: two samples of rare malware
Output: more destructive and stealth generations

```
1.  i = 0;
2.  k = 0;
3.  DataSet = 0;
4.  M = Rare Malware Dataset;
5.  Initialize (Pi (EliteSelection(M)));
                -Choosing parent 1
6.  Initialize (Pk (EliteSelection(M)));
                -Choosing parent 2
7.  While KBisNotTerminated() do
8.    Mutation (Pi);
9.    Mutation (Pk);
10.   Child ← Crossover(Pi, Pk);
                        -Generating new child
11.     FitnessFunction(Child, Pi);
                        -Comparing with parent 1
12.       If (MRChild >MRPi & TRChild <TRPi)
13.         FitnessFunction(Child, Pk);
                        -Comparing with parent 2
14.           If (MRChild >MRPk & TRChild <TRPk)
15.             DataSet ← Child
                                -Adding child to the dataset
16.   i = i+1;
17.   k = k+1;
18.   Next;
19. End.
```

V. EVALUATION AND DISCUSSION OF THE RESULTS

In this section, we discuss the performance of our method in terms of the quality of the generated and optimized dataset. The accuracy of the trained models for detecting rare classes of malware and its future mutations are also addressed.

A. EVALUATION OF THE QUALITY OF THE NEW GENERATIONS

In our proposed method, modification of API calls after each mutation changes the child's structural and behavioral signatures. This change in the signature can be confirmed by performing MD5 and SHA-1 hashes on new generations to compare them with their parents' signatures.

According to Equation 4, the mutant malware should behave more destructively when compared to its parents, and be more secretive than them. Figure 6 shows the malignancy rate that occurred for new generations produced by our proposed method.

As the diagram in Figure 6 illustrates, the new mutant generations in all seven classes described in this work behaved far more destructively than their parents. The values for TR for new generations are provided in Table 4. As shown, new generations were far more stealthy compared to their parents. To improve data clarity, TR values have been provided as a line chart in Figure 7.

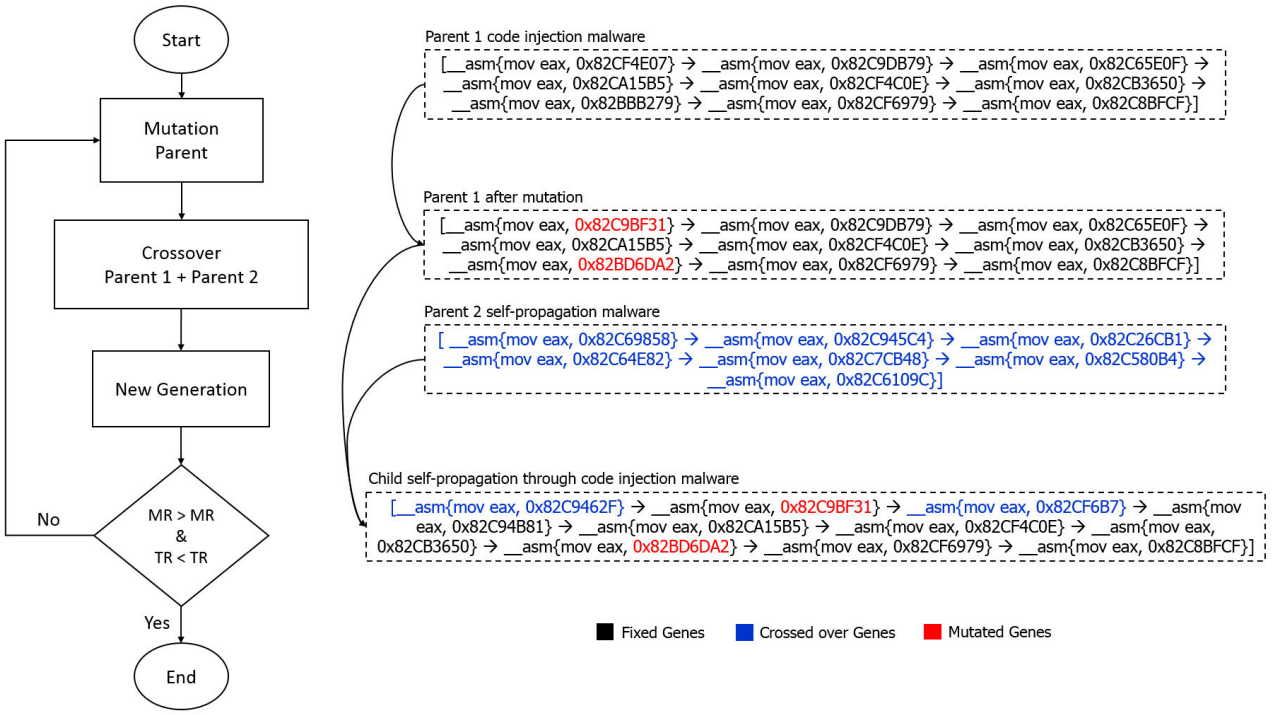

FIGURE 5. Crossover and mutation of the malware encoded genes in the proposed method.

TABLE 4. The tracking rate for new generations of seven rare classes of malware.

Gen / TR	St Spy	Blckr	FileL	Injec	RootK	BootK	Evdr
Parent 1	31%	49%	37%	41%	40%	37%	33%
Parent 2	29%	46%	33%	38%	35%	31%	27%
1st Gen.	23%	44%	29%	36%	33%	28%	25%
2nd Gen.	21%	39%	27%	30%	27%	25%	21%
3rd Gen.	19%	30%	23%	28%	25%	21%	18%
4th Gen.	12%	28%	19%	24%	22%	17%	15%
5th Gen.	6%	25%	13%	20%	19%	10%	13%
6th Gen.	6%	23%	11%	17%	16%	7%	10%
7th Gen.	6%	23%	8%	13%	11%	7%	10%

As Table 4 and Figure 7 show, the mutant generations in each class were detectable by fewer anti-malware scanners. This illustrates that the behavior of the new generations was more secretive than their parents.

From both Figure 6 and 7, we can infer that the increase in the malignancy rate of malware samples as well as the decrease in their tracking rate, were higher in the first generations compared to their parents, and as the generations evolve, the rate diminished so that after a multiple generations mutation evolution moves to a steady state, in which MR and TR can no longer be improved. In our study, a steady state was met after the fifth generation for exploiter spyware and after the sixth generation for the injector classes of malware. Gene substitution could not improve the figures for both MR and TR beyond this point.

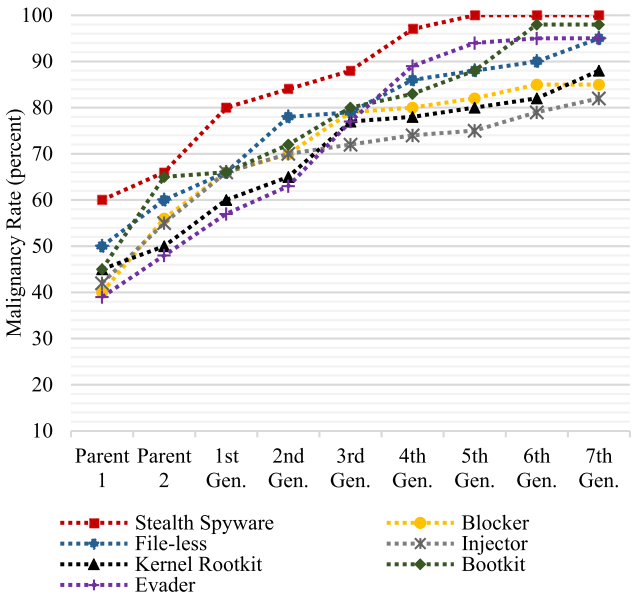

FIGURE 6. The graph of malignancy rate in mutant generations.

B. MODEL TRAINING

In this subsection, we trained two classifiers, a popular one from traditional machine learning approaches and another from new deep learning techniques. We used Weka UI[9] version 3.7.4 and H2O-3[10] to perform classification in our proposed method. A comprehensive evaluation and comparison

[9] https://www.cs.waikato.ac.nz/ml/weka/
[10] https://github.com/h2oai/h2o-3

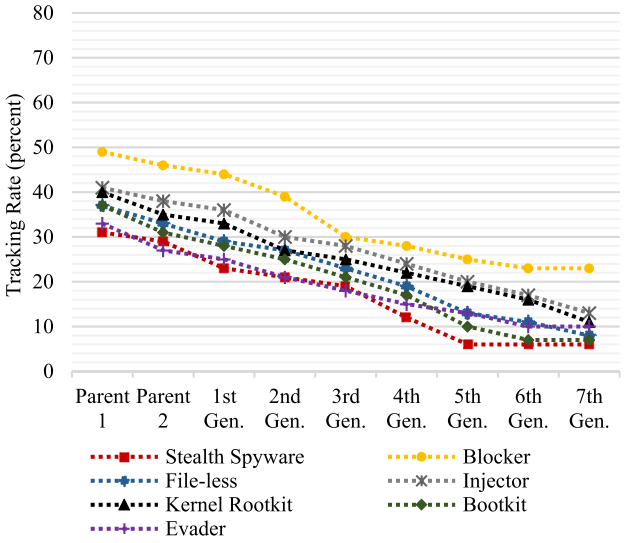

FIGURE 7. The graph of tracking rate in mutant generations.

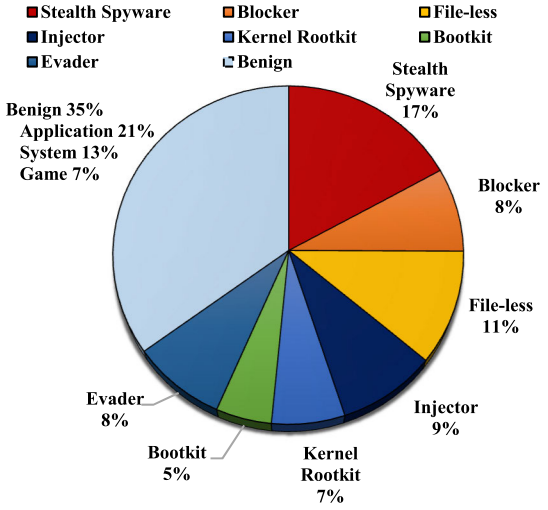

FIGURE 8. The frequency of dataset classes used to train the model.

have been provided to prove the effectiveness of our proposed method.

1) DATASETS

As previously mentioned, Adminus [43], VirusSign [44], and VirusShare [45] datasets were used in our work. These datasets have been collected during the seven years between 2013 and 2020. We chose 66% of our data as training data and the remaining 33% as testing data. The total population of the datasets used for training the classifiers includes 11632 records, of which 7563 samples were seven rare classes mentioned in Table 4, and 4069 samples of benign files from the three safe classes. Table 5 indicates the number of samples for each dataset.

TABLE 5. The number of samples for each dataset.

#	Dataset	Num. of samples
1	Adminus	1530
2	VirusSign	1800
3	VirusShare	1163
4	Optimized	3070
5	Benigns	4069

The distribution of malware and benign classes for training data have been illustrated as a pie chart in Figure 8. Benign files have been collected from Windows system files, software applications, and games.

It is worth mentioning that the proposed GA needed real binary samples of malware to generate and optimize the dataset of possible future mutations of malware. Other existing datasets such as Microsoft Big 2015, Malimg, Malheur,[11] and MaleVis[12] that do not contain malware binaries cannot be used for the proposed methods. Microsoft Big 2015 and Malheur 2016 recorded malware behavior as CSV and XML files, and MaleVis 2015 and Malimg 2011 have consisted of PNG images. Malicia[13] was another malware dataset for 2013 – but its project is discontinued now, so its distribution was stopped by the provider due to aging. It is not possible to detect unknown and zero-day malware if the model has not been trained by the latest releases of the dataset. This issue indicates how essential model updating is. The datasets used in our work were updated daily by the relevant vendors.

Creating a newly generated dataset that contains malware samples with higher MR and lower TR led to a significant enhancement in the accuracy of the predictor model since the model was trained with a better-quality dataset in our approach. The behavioral model not only detects rare and unknown malware, but it is also able to detect changes caused by metamorphic and polymorphic engines. The data presented in Table 4 shows that without any mutation in the stealth spyware class, 70% of the scanners were not able to detect malware programs, while with the resulting mutations, this figure reached 94%, so that only 6% of anti-malware programs could detect malware programs correctly. It should be noted that maximizing the malignancy rate and minimizing the tracking rate in new generations have led to high-quality data being produced for better training, and consequently a better model. This is more obvious when the model is used to detect rare, polymorphic, and metamorphic classes of malware. Some characteristics of the dataset used to train these models include:

- The comprehensiveness of the training data. All classes and subclasses of the malware are included in the relevant domain as much as possible.
- Data mining is performed on an appropriate amount of malware samples. Definitely, it is not applicable to include all classes of malware in our training dataset.

[11] https://www.sec.cs.tu-bs.de/data/malheur/
[12] https://web.cs.hacettepe.edu.tr/~selman/malevis/
[13] http://malicia-project.com/dataset.html

Modeling must be done on an optimal scale and should be scalable and reliable.

- The model was created according to a balanced dataset. Obviously, unbalanced training has adverse effects on the total accuracy of the classification.
- Appropriate time distribution was considered. Malware programs behave differently depending on the technologies available at the time of their construction. Therefore, if the statistical population for training is limited to a particular timeframe, then older or newer malware might not be detectable by the resulted model. However, if malware programs with quite different release times are sampled, the accuracy of the model will decrease because of the outliers. Hence, we performed sampling at an appropriate interval of malware release time, which should be constantly updated.

2) SHALLOW MACHINE LEARNING ALGORITHMS

The J48 decision tree algorithm provided by Weka was used as the classifier to train our model based on supervised learning. The reason for choosing a decision tree (DT) is its popularity for modeling cybersecurity attacks [59]. DT algorithms are also consistent with rule-based KBs and are able to perform high-speed file scanning tasks, particularly when the KB is structured as a rule-based system.

3) DEEP NEURAL NETWORKS

We have used both machine learning algorithm and deep neural network (DNN) for training and testing the malware detection model based on our optimized dataset. According to the literature, J48 is a popular classification algorithm in the category of traditional machine learning strategies. It should be noted that DNN is a modern algorithm that can handle higher dimensions which results in more accurate and robust models.

A DNN is a feedforward artificial neural network (ANN) with many hidden layers between the input and output layers [60]. A deep neural network needs to have more than three layers, in which the number of hidden layers indicates the depth of the network. Neural networks have been widely used in various domains such as machine vision, natural language processing, speech recognition, handwriting recognition, sentiment analysis, and medical image analysis [49], [60]. Here we have utilized DNN for training a malware detection and classification model. The structure of the DNN is illustrated in Figure 9 [61].

As Figure 9 shows, 'a' demonstrates neurons, 'x' is neuron input, 'y' is neuron output, and 'W' indicates the weight matrix.

In order to make the optimized dataset compatible for training by DNN, we transformed all malware and benign binaries into grayscale images. There were two methods for this conversion. The first method converts malware binary files into 8-bit vectors consisting of a string of zeros and ones, then converts vectors to grayscale or RGB images [31]–[33]. The second method disassembles malware binary files into

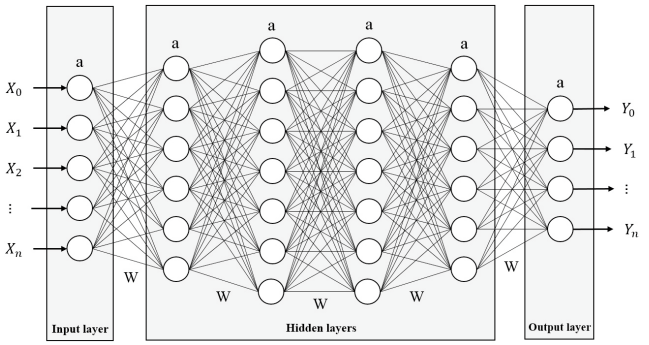

FIGURE 9. The structure of the deep neural network [61].

assembly code and then converts the assembly code into a grayscale image, much like the method described in [38]. The latter method has increased challenges as it needs to reverse the malware binary. There are also some obstacles regarding reversing obfuscated and packed malware, including packer identification and decrypting ciphered sections [8], normalizing metamorphic codes [13], and executing time-limited malware - malware that only runs on a specific point or period of time. However, its efficiency was proven compared to other methods, and the amount of data required for creating the corresponding images is smaller than the former method. In our work, the disassembling malware binary files method was used since assembly codes for each malware binary were already available at step 2 of our analysis architecture, e.g. deep static analysis through disassembling by IDA Pro in Figure 3. We converted each ASM code to a 512 × 512 grayscale image, and each pixel was valued at 0 to 255. This resolution makes it possible to convert ASM files with the maximum size of 256 KB, which is sufficient for different malware types with the minimum data loss. Next, a deep neural network algorithm was utilized to classify samples using H2O-3 platform. H2O-3 is an open-source platform to provide an integrated and scalable framework for machine learning and deep learning [62]. It can be installed and used locally or online over well-known clouds such as Amazon[14] and Aquarium[15] - through a web-based user interface.

We performed supervised learning in algorithm training mode running on ml.c4.8xlarge instance type in H2O-3. Supervised learning was possible through labeling data using our multi-scanner - this architecture has been described in Table 7. H2O-3 provides various functions and a wide range of automatic and manual options for tuning deep learning hyperparameters to reach the optimal set of hyperparameters including learning rate (step size), batch size, momentum, and weight decay. Gradient descent was used as an optimization technique and Softmax as an activator function.

[14]https://amzn.to/3bMUI7l
[15] https://www.h2o.ai/test-drive/

TABLE 6. Comparison of accuracy, Error rate, TPR, and FPR between models trained by different datasets.

Classifier	Class \ DS	Adminus				VirusSign				VirusShare				Optimized			
		Acc	Err	TPR	FPR	Acc	Err	TPR	FPR	Acc	Err	TPR	FPR	Acc	Err	TPR	FPR
J48	Stealth Spyware	0.87	0.13	0.61	0.07	0.90	0.10	0.71	0.06	0.89	0.11	0.70	0.10	0.96	0.04	0.91	0.02
	Blocker	0.90	0.10	0.62	0.07	0.93	0.07	0.65	0.04	0.87	0.13	0.62	0.11	0.95	0.05	0.81	0.03
	File-less	0.87	0.13	0.69	0.11	0.90	0.10	0.66	0.08	0.86	0.14	0.61	0.12	0.94	0.06	0.79	0.04
	Injector	0.88	0.12	0.61	0.09	0.91	0.09	0.74	0.07	0.90	0.10	0.61	0.07	0.96	0.04	0.79	0.04
	Kernel Rootkit	0.85	0.15	0.56	0.13	0.88	0.12	0.63	0.10	0.91	0.09	0.70	0.06	0.92	0.08	0.79	0.07
	Bootkit	0.90	0.10	0.68	0.09	0.89	0.11	0.60	0.11	0.90	0.10	0.63	0.07	0.90	0.10	0.81	0.09
	Evader	0.88	0.12	0.60	0.10	0.90	0.10	0.72	0.09	0.91	0.09	0.61	0.05	0.93	0.07	0.78	0.06
	Benign	0.92	0.08	0.91	0.07	0.94	0.06	0.96	0.07	0.93	0.07	0.95	0.09	0.96	0.04	0.97	0.05
	Average	**0.88**	**0.12**	**0.66**	**0.09**	**0.91**	**0.09**	**0.71**	**0.08**	**0.90**	**0.10**	**0.68**	**0.08**	**0.94**	**0.06**	**0.83**	**0.05**
DNN	Stealth Spyware	0.89	0.11	0.67	0.05	0.92	0.08	0.75	0.04	0.91	0.09	0.72	0.04	0.98	0.02	0.94	0.01
	Blocker	0.93	0.07	0.70	0.05	0.93	0.07	0.72	0.04	0.93	0.07	0.71	0.04	0.96	0.03	0.84	0.01
	File-less	0.92	0.08	0.78	0.06	0.94	0.06	0.86	0.05	0.94	0.06	0.87	0.05	0.96	0.04	0.85	0.03
	Injector	0.91	0.09	0.69	0.07	0.93	0.07	0.81	0.06	0.88	0.12	0.69	0.10	0.98	0.02	0.95	0.01
	Kernel Rootkit	0.88	0.12	0.61	0.11	0.89	0.11	0.70	0.09	0.92	0.08	0.74	0.06	0.93	0.07	0.84	0.07
	Bootkit	0.91	0.09	0.87	0.09	0.90	0.10	0.74	0.09	0.90	0.10	0.72	0.09	0.94	0.06	0.91	0.05
	Evader	0.91	0.09	0.81	0.08	0.92	0.08	0.81	0.07	0.91	0.09	0.84	0.08	0.95	0.05	0.91	0.05
	Benign	0.94	0.06	0.94	0.05	0.92	0.08	0.75	0.04	0.94	0.06	0.96	0.07	0.98	0.02	0.99	0.03
	Average	**0.91**	**0.09**	**0.76**	**0.07**	**0.92**	**0.08**	**0.77**	**0.06**	**0.92**	**0.08**	**0.78**	**0.07**	**0.96**	**0.04**	**0.90**	**0.03**

C. MODEL VALIDATION

We used 10-fold cross-validation to train and validate the model in the experiments. And a grid search with cross-validation was used for deep learning to search for the best configuration during cyclical learning rate between 0.80 to 0.85 to avoid overfitting and cyclical momentum in boundaries of 0.99 down to 0.90. Weight decay remained constant without causing instabilities. We set a cutoff up to 56 for the number of iterations and a max depth of {5, 10, 15, 20}; dimensions of the multiplications matrix, batch size, and the number of epochs have been automatically tuned by the H2O-3 in order to achieve the optimal state and the highest possible accuracy.

Both J48 and DNN classifiers were trained on three datasets and were compared to the classifiers trained by our optimized dataset in eight separate experiments. The obtained results demonstrate the effectiveness of the proposed GA in detecting rare malware. In the experiments, 3781 real-world malware samples and 2035 benign files were used as test data. This was necessary for determining how accurate the trained model was in detecting unknown and zero-day malware. Footprints of more than 150 metamorphic and obfuscator engines were found in the test data samples by searching the n-gram of the remaining unique subsequences, indicating all samples are packed and obfuscated. All experiments were repeated three times to avoid environmental errors such as VM crashes during the scanning process, or any mistakes that may have been made by the analyzer while recording results. The results of the performance evaluation are presented in Table 6. This shows accuracy, error (mis-classification) rate, true positive rate, and false positive rate.

The values in Table 6 were created by calculating the accuracy and classification error according to Equations 5 and 6 [63].

$$Acc. = \frac{TP + TN}{TP + TN + FP + FN} \quad Avg.Acc. = \frac{\sum_{i=0}^{N} Acc._i}{N} \tag{5}$$

$$Err. = \frac{FP + FN}{TP + TN + FP + FN} \quad Avg.Err. = \frac{\sum_{i=0}^{N} Err._i}{N} \tag{6}$$

where *Acc.* and *Err.* indicate the accuracy and the error rate when detecting each class of malware. N demonstrates the total number of classes used in model training, *TP* (True Positive) and *TN* (True Negative) are correctly classified malware samples and benign files, respectively. *FN* (False Negative) indicates the number of malware samples mistakenly classified as benign, while *FP* (False Positive) indicates the number of benign files mistakenly recognized as malware. *Avg. Acc.* and *Avg. Err.* refer to the average accuracy and the average error rate for the total N classes, respectively. True Positive Rate (TPR), also called sensitivity, and False Positive Rate (FPR), also called specificity, were calculated according to Equation 7.

$$TPR = \frac{TP}{TP + FN} \quad FPR = \frac{FP}{FP + TN} \tag{7}$$

D. DISCUSSION

As it can be understood from Table 6, after utilizing our proposed method, there was an improvement in detecting all malware classes. However, the rate of improvement is variable for each class. The most improvement was for stealth spyware and injector classes, and the least improvement was for bootkit and kernel-level rootkit classes. This is because the detection accuracy is correlated tightly with the quality of samples that existed in the training dataset, as well as their quantity. Also, it depended on how successful we were in unpacking malware executables and normalizing their obfuscated behavior in order to extract essential features for training. Some classes that are more significant and costly for their developers were equipped with dedicated facilities, such as customized packers and anonyms obfuscator engines, to prevent any malware analysis and unpacking process. Therefore, since real-world malware samples have been studied in our approach the effects of a successful or unsuccessful unpacking on the accuracy is inevitable.

The averaged values obtained from the successful trials are provided in Figure 10. This shows that the average accuracy for the classification of rare and metamorphic malware for both J48 and DNN classifiers was higher when the models were trained on the optimized dataset. Also, the average error rate was less for our method.

Figure 10 also shows that there was between 4% and 5% improvement in the average accuracy for our method when the DNN classifier was employed compared to other datasets. There was also a 3% to 5% improvement when J48 was utilized. These results also show that TPR and FPR were significantly improved in our approach. The average TPR of seven classes was increased between 12% and 14% in comparison with other datasets for DNN Classifier, and between 12% and 17% for the J48 Classifier. Additionally, the average FPR decreased between 3% and 4% for both classifiers.

Low accuracy in detecting rare malware classes was due to the lack of relevant samples in a dataset since vulnerabilities for such malware classes are unpublished and not easily accessible, so they cannot be hunted by AVs or Honeypots. The proposed GA has effectively addressed these challenges by increasing the population of such classes and enhancing their stealth as well. The following subsection compares the accuracy of the classifiers used in our experiments.

E. COMPARISON OF THE CREATED MODELS

In our experiments, two classifiers were trained on four datasets. A Receiver Operating Characteristics (ROC) and Area under the Curve (AUC) were conducted to provide a comprehensive comparison between the classifiers trained on each dataset.

A ROC curve is plotted against TPR and FPR for different thresholds in [60]. By illustrating a ROC curve, it is possible to make a trade-off between TPR and FPR rates that are already obtained from Equation 7. The area under the ROC curve (AUC) is calculated using Equation 8. AUC values close to 1 indicates a higher stability for the classifier.

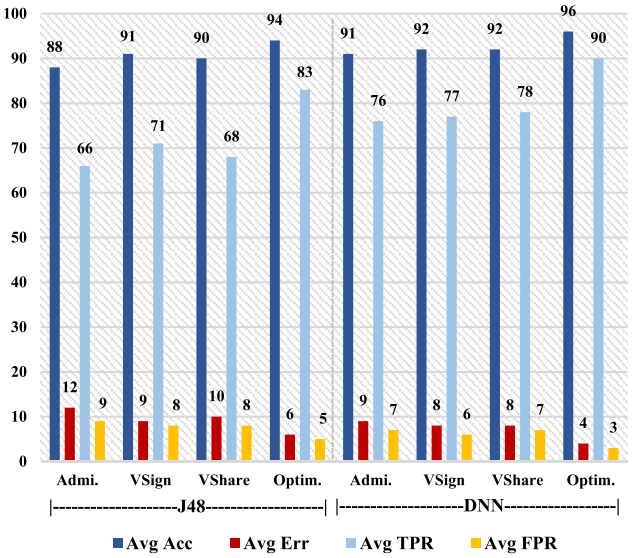

FIGURE 10. The comparison between classifiers and datasets.

$$AUC = \int_0^1 \frac{TP}{TP + FN} d \frac{FP}{TN + FP} \tag{8}$$

ROC curve and AUC for each classifier have been calculated during all eight experiments (Figure 11).

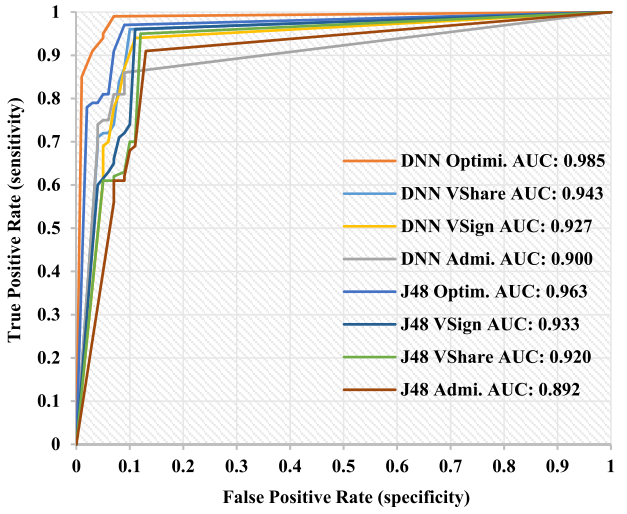

FIGURE 11. ROC curve and AUC analysis for each classifier.

As demonstrated in Figure 11, the value of AUC for the DNN method with the optimized dataset was larger than all other classifiers. This illustrates that deep learning generates a more discriminative model considering all datasets, except VirusSign in which J48 had a better performance. Adminus had the smallest value of AUC for both J48 and deep learning classifiers, which indicates that the quality of samples in this dataset is lower than other datasets while the quality of our evolved dataset is better than others.

TABLE 7. The configuration of our multi-scanner system.

Virtualization Technology	Hypervisor	System Architecture	Host Machine	Guest OS Kernel Version	AV Scanner
Intel VT	VMware ESXi	x86	Ubuntu 12.04	NT 5.2 (Win XP, 2003)	Kaspersky
				NT 6.1 (Win 7, 2008R2)	Avira
				NT 6.2 (Win 8, 2012)	Eset-Nod32
				NT 6.3 (Win 8.1, 2012R2)	Sophos
				NT 10 (Win 10, 2016)	McAfee
	VMware ESXi	x64	Ubuntu 14.04	NT 5.2 (Win XP, 2003)	ClamAV
				NT 6.1 (Win 7, 2008R2)	Dr.Web
				NT 6.2 (Win 8, 2012)	TrendMicro
				NT 6.3 (Win 8.1, 2012R2)	F-Secure
				NT 10 (Win 10, 2016)	G-Data
AMD-V	VMware ESXi	x86	Debian 7	NT 5.2 (Win XP, 2003)	Avest
				NT 6.1 (Win 7, 2008R2)	Malwarebytes
				NT 6.2 (Win 8, 2012)	ViRobot
				NT 6.3 (Win 8.1, 2012R2)	AVG
				NT 10 (Win 10, 2016)	Comodo
	VMware ESXi	x64	Debian 8	NT 5.2 (Win XP, 2003)	FireEye
				NT 6.1 (Win 7, 2008R2)	Ikarus
				NT 6.2 (Win 8, 2012)	TrendMicro
				NT 6.3 (Win 8.1, 2012R2)	Zillya
				NT 10 (Win 10, 2016)	Emsisoft

F. TIME COMPLEXITY AND SYSTEM CONFIGURATION

The time complexity for the genetic algorithm section of our method can be calculated as $O(g.(n.m + n.m + n))$ where g is the number of generations, n is the population size, and m is the sample size. Hence, the time complexity can be considered as an order of $O(g.n.m)$.

In the case of classification of new generations of modern malware, mostly created using metamorphic engines, an optimal population should be considered statistically, otherwise, handling a large set of metamorphic and polymorphic malware can push the problem into an infinite state that practically cannot be solved with optimization algorithms having linear time complexity. The importance of time complexity is further evident when detecting malicious behavior during run-time.

Model training for all experiments in this work was conducted on a machine with a CPU with 24 cores, 48 logical processors. The machine had 128GB of RAM and 10TB of SAS HDD. The configuration of our multi-scanner system is presented in Table 7.

G. LIMITATIONS AND CONTINUATION OF THE RESEARCH

It should be noted that the proposed method for the crossover of chromosomes requires at least two samples of malware. Where only one sample of basic malware is available, the crossover operation would not be plausible, and the mutation of chromosomes will be only possible by substituting functions. Moreover, the process of monitoring malware capable of Hyperjacking requires installing hooks at ring-3, which is not supported by the proposed architecture due to the lack of sufficient hardware requirements.

As discussed at the start of this paper, due to the popularity of Microsoft Windows OS, most malware programs are created for this platform. For this reason, this article focused on PE type malware in x86 and x64 architectures. Our proposed method is also consistent with and applicable to detecting new generations of malware on other platforms such as ART on Android and ELF on Linux.

VI. CONCLUSION

The detection of rare malware programs that were specifically and purposefully created has always been a severe challenge due to the lack of samples required to properly model relevant malware detection systems. This paper defined a novel malware detection strategy that used a combination of a succinct feature extraction method and creating an optimized dataset utilizing a modified genetic algorithm. The method proposed in this paper used genetic algorithms to conduct the evolution of rare malware through crossover and mutation processes in behavioral genes to generate a suitable dataset to train the model for malware detection. In the evolution process, the quality of new generations of malware was increased and the malignancy rate and tracking rate were measured by influential indicators. Having an evolved dataset that was created according to the extracted behavior of obfuscated and compound malware samples provided a conceptual guideline for the trained model to better detect future mutations. This thesis increased the accuracy of the model and decreased classification errors for rare and metamorphic malware.

ACKNOWLEDGMENT

The authors would like to acknowledge the assistance of the head of cyber security research center of SRBIAU and APA specialized center of IUMS in providing them with commercial datasets and tools. They also wish to thank David Venema for his patience in proof-reading and offering editorial advice.

REFERENCES

[1] *AV-Test Report*. Accessed: 2020. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[2] A. Zimba and M. Chishimba, "On the economic impact of crypto-ransomware attacks: The state of the art on enterprise systems," *Eur. J. Secur. Res.*, vol. 4, no. 1, pp. 3–31, Apr. 2019.

[3] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. Sutter, and K. Bosschere, "Software protection through dynamic code mutation," in *Proc. 6th Int. Conf. Inform. Secur. Appl.*, 2006, pp. 194–206.

[4] A. G. Kakisim, M. Nar, and I. Sogukpinar, "Metamorphic malware identification using engine-specific patterns based on co-opcode graphs," *Comput. Standards Interface*, vol. 71, Aug. 2020, Art. no. 103443.

[5] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *J. Netw. Comput. Appl.*, vol. 153, Mar. 2020, Art. no. 102526.

[6] D. Javaheri, M. Hosseinzadeh, and A. M. Rahmani, "Detection and elimination of spyware and ransomware by intercepting kernel-level system routines," *IEEE Access*, vol. 6, pp. 78321–78332, 2018.

[7] I. Ismail, M. N. Marsono, B. M. Khammas, and S. M. Nor, "Incorporating known malware signatures to classify new malware variants in network traffic," *Int. J. Netw. Manage.*, vol. 25, no. 6, pp. 471–489, Nov. 2015.

[8] B. Jung, S. I. Bae, C. Choi, and E. G. Im, "Packer identification method based on byte sequences," *Concurrency Comput., Pract. Exper.*, vol. 32, no. 8, Apr. 2020, Art. no. e5082.

[9] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 1009–1024.

[10] N. Namani and A. Khan, "Symbolic execution based feature extraction for detection of malware," in *Proc. 5th Int. Conf. Comput., Commun. Secur. (ICCCS)*, Oct. 2020, pp. 1–6.

[11] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Proc. Int. Conf. Broadband, Wireless Comput., Commun. Appl.*, Nov. 2010, pp. 297–300.

[12] S. Sun, X. Fu, H. Ruan, X. Du, B. Luo, and M. Guizani, "Real-time behavior analysis and identification for Android application," *IEEE Access*, vol. 6, pp. 38041–38051, 2018.

[13] S. Jagsir and J. Singh, "Challenges of malware analysis: Obfuscation techniques," *Int. J. Inf. Secur. Sci.*, vol. 7, no. 3, pp. 100–110, 2018.

[14] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, and A. Tonda, "Towards automated malware creation: Code generation and code integration," in *Proc. 29th Annu. ACM Symp. Appl. Comput.*, Mar. 2014, pp. 157–160.

[15] W. Wong and M. Stamp, "Hunting for metamorphic engines," Dept. Comput. Sci., San Jose State Univ., San Jose, CA, USA, 2006.

[16] P. Desai, "A highly metamorphic virus generator," *Int. J. Multimedia Intell. Secur.*, vol. 1, pp. 402–427, Jan. 2010.

[17] S. Priyadarshi, "Metamorphic detection via emulation," M.S. thesis, Dept. Comput. Sci., San Jose State Univ., San Jose, CA, USA, 2011.

[18] A. Mohanta and A. Saldanha, "Malware packers," in *Malware Analysis and Detection Engineering*. Berkeley, CA, USA: Apress, 2020, pp. 640–642.

[19] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang, "Evading anti-malware engines with deep reinforcement learning," *IEEE Access*, vol. 7, pp. 48867–48879, 2019.

[20] C. Easttom, "An examination of the operational requirements of weaponised malware," *J. Inf. Warfare*, vol. 17, no. 2, pp. 1–15, 2018.

[21] B. Grill, C. Platzer, and J. Eckel, "A practical approach for generic bootkit detection and prevention," in *Proc. 7th Eur. Workshop Syst. Secur. (EuroSec)*, 2014, pp. 1–6.

[22] A. Fagioli, "Zero-day recovery: The key to mitigating the ransomware threat," *Comput. Fraud Secur.*, vol. 2019, no. 1, pp. 6–9, Jan. 2019.

[23] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 483–493.

[24] D. Korczynski and H. Yin, "Capturing malware propagations with code injections and code-reuse attacks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1691–1708.

[25] *Kaspersky Encyclopedia*. Accessed: 2021. [Online]. Available: https://encyclopedia.kaspersky.com/glossary/bootkit/

[26] *McAfee Website*. Accessed: 2021. [Online]. Available: https://www.mcafee.com/enterprise/en-us/security-awareness/ransomware/what-is-fileless-malware.html

[27] N. Rakotondravony, B. Taubmann, W. Mandarawi, E. Weishäupl, P. Xu, B. Kolosnjaji, M. Protsenko, H. de Meer, and H. P. Reiser, "Classifying malware attacks in IaaS cloud environments," *J. Cloud Comput.*, vol. 6, no. 1, pp. 1–12, Dec. 2017.

[28] D. Javaheri and M. Hosseinzadeh, "A framework for recognition and confronting of obfuscated malwares based on memory dumping and filter drivers," *Wireless Pers. Commun.*, vol. 98, no. 1, pp. 119–137, Jan. 2018.

[29] M. Sewak, S. K. Sahay, and H. Rathore, "Comparison of deep learning and the classical machine learning algorithm for the malware detection," in *Proc. 19th IEEE/ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Jun. 2018, pp. 293–296.

[30] S. Venkatraman, M. Alazab, and R. Vinayakumar, "A hybrid deep learning image-based analysis for effective malware detection," *J. Inf. Secur. Appl.*, vol. 47, pp. 377–389, Aug. 2019.

[31] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware classification with deep convolutional neural networks," in *Proc. 9th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Feb. 2018, pp. 1–5.

[32] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, pp. 46717–46738, 2019.

[33] S. A. Roseline, S. Geetha, S. Kadry, and Y. Nam, "Intelligent vision-based malware detection and classification using deep random forest paradigm," *IEEE Access*, vol. 8, pp. 206303–206324, 2020.

[34] X. Wan, G. Sheng, Y. Li, L. Xiao, and X. Du, "Reinforcement learning based mobile offloading for cloud-based malware detection," in *Proc. GLOBECOM IEEE Global Commun. Conf.*, Dec. 2017, pp. 1–6.

[35] Y. Ding, R. Wu, and X. Zhang, "Ontology-based knowledge representation for malware individuals and families," *Comput. Secur.*, vol. 87, Nov. 2019, Art. no. 101574.

[36] A. Mohaisen, O. Alrawi, and M. Mohaisen, "AMAL: High-fidelity, behavior-based automated malware analysis and classification," *Comput. Secur.*, vol. 52, pp. 251–266, Jul. 2015.

[37] M. Imran, M. T. Afzal, and M. A. Qadir, "Malware classification using dynamic features and hidden Markov model," *J. Intell. Fuzzy Syst.*, vol. 31, no. 2, pp. 837–847, Jul. 2016.

[38] L. Liu, B.-S. Wang, B. Yu, and Q.-X. Zhong, "Automatic malware classification and new malware detection using machine learning," *Frontiers Inf. Technol. Electron. Eng.*, vol. 18, no. 9, pp. 1336–1347, Sep. 2017.

[39] R. U. Khan, X. Zhang, and R. Kumar, "Analysis of ResNet and GoogleNet models for malware detection," *J. Comput. Virol. Hacking Techn.*, vol. 15, no. 1, pp. 29–37, Mar. 2019.

[40] W. Wang, J. Wei, S. Zhang, and X. Luo, "LSCDroid: Malware detection based on local sensitive API invocation sequences," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 174–187, Mar. 2020.

[41] A. Makkar, S. Garg, N. Kumar, M. S. Hossain, A. Ghoneim, and M. Alrashoud, "An efficient spam detection technique for IoT devices using machine learning," *IEEE Trans. Ind. Informat.*, vol. 17, no. 2, pp. 903–912, Feb. 2021.

[42] R. Kumar, Z. Xiaosong, R. U. Khan, I. Ahad, and J. Kumar, "Malicious code detection based on image processing using deep learning," in *Proc. Int. Conf. Comput. Artif. Intell. (ICCAI)*, 2018, pp. 81–85.

[43] *Adminus Malware Dataset 2016-18*. Accessed: 2018. [Online]. Available: http://www.adminus.net

[44] *VirusSign Malware Dataset 2013-20*. Accessed: 2020. [Online]. Available: http://www.virussign.com

[45] *VirusShare Malware Dataset 2016-20*. Accessed: 2020. [Online]. Available: http://www.virusshare.com

[46] R. D. Reeves, *Windows 7 Device Driver*. Boston, MA, USA: Addison-Wesley Publisher, 2010, pp. 135–136.

[47] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Comput. Surveys*, vol. 50, no. 3, pp. 1–40, Oct. 2017.

[48] S. Mirjalili, "Genetic algorithm," in *Evolutionary Algorithms and Neural Networks, Studies in Computational Intelligence*, vol. 780. Cham, Switzerland: Switzerland, 2019, pp. 43–55.

[49] A. J. Kulkarni, and S. C. Satapathy, *Optimization in Machine Learning and Applications*. Singapore: Springer, 2020.
[50] B. M. Varghese and R. J. S. Raj, "A survey on variants of genetic algorithm for scheduling workflow of tasks," in *Proc. 2nd Int. Conf. Sci. Technol. Eng. Manage. (ICONSTEM)*, Mar. 2016, pp. 489–492.
[51] C. H. Kumar, S. H. Prakash, T. K. Gupta, and D. P. Sahu, "Variant of genetic algorithm and its applications," *Int. J. Artif. Intell. Neural Netw.*, vol. 4, no. 4, pp. 8–12, 2014.
[52] M. Jurczyk. *Windows System Call Tables*. Accessed: 2020. [Online]. Available: https://github.com/j00ru/windows-syscalls
[53] M. Russinovich, D. Solomon, and A. Ionescu, *Windows Internals Part 1*, 6th ed. Redmond, WA, USA: Microsoft Press, 2012, pp. 133–138.
[54] P. Cichosz, *Data Mining Algorithms: Explained Using R* 1st ed. Hoboken, NJ, USA: Wiley, 2015.
[55] B. Schreiber, *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Boston, MA, USA: Addison Wesley Longman Publishing Co., 2001.
[56] (2014). *Joe Sandbox*. Accessed: 2020. [Online]. Available: https://www.joesandbox.com
[57] (2014). *Virus Total*. Accessed: 2020. [Online]. Available: https://www.virustotal.com
[58] *AV-Test Ranking List*. Accessed: 2020. [Online]. Available: https://www.av-test.org/en/antivirus/home-windows/
[59] I. H. Sarker, Y. B. Abushark, F. Alsolami, and A. I. Khan, "IntruDTree: A machine learning based cyber security intrusion detection," *Model. Symmetry*, vol. 12, no. 5, pp. 754–769, 2020.
[60] R. Kumars, M. Alazab, and W. Wang, "A survey of intelligent techniques for Android malware detection,"in *Malware Analysis Using Artificial Intelligence and Deep Learning*. Cham, Switzerland: Springer, 2021.
[61] N. Yuvaraj, R. A. Raja, N.V. Kousik, P. Johri, and M. J. Diván, *Analysis on the Prediction of Central Line-Associated Bloodstream Infections (CLABSI) Using Deep Neural Network Classification, Computational Intelligence and its Applications in Healthcare*. New York, NY, USA: Academic, 2020, pp. 229–244.
[62] *H2O AI Hybrid Cloud*. Accessed: 2021. [Online]. Available: https://www.h2o.ai/
[63] A. Honig, and M. Sikorski, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA, USA: No Starch Press, 2012, pp. 221–224.

DANIAL JAVAHERI (Member, IEEE) received the B.Sc., M.Sc. (Hons.), and Ph.D. (Hons.) degrees in computer software engineering from Islamic Azad University (IAU), in 2012, 2014, and 2018, respectively. In 2015, he joined the Cyber Security Research Center, IAU, and the APA Specialized Center, as a Security Specialist and has maintained his role as a university lecturer with a number of top universities in Iran. Over the past decade, he has conducted extensive research in the field of software security, network forensics, and malware analysis, and has published several articles and books in relevant areas. Due to his exemplary performance in education, he has been awarded the Top Student and the Top Graduated by the Iran's National Elites Foundation (INEF), from 2017 to 2018, including being awarded several honors by the university.

POOIA LALBAKHSH received the Ph.D. degree in computer science and computer engineering from La Trobe University, Melbourne, Australia, in 2017. He has more than 15 years of experience in academia and industry, working in the areas of decision support systems, simulation modeling, machine learning, swarm intelligence, and universal AI. He has completed several research and industrial projects in the areas of capacity analysis, adaptive intelligent trading, virtual therapy, financial distress analysis, network monitoring, adaptive routing, and capital market prediction. He has collaborated with several multidisciplinary teams from La Trobe University, RMIT University, The Australian National University (ANU), and the Defense Science and Technology Group (DSTG), Australia. He is currently working as an Artificial Intelligence Scientist with Euler Capital, Australia. He also focuses on intelligent trading, detecting market regimes, and predicting anomalies and arbitrage.

MEHDI HOSSEINZADEH received the B.Sc. degree in computer hardware engineering from the Islamic Azad University of Dezful, Iran, in 2003, and the M.Sc. and Ph.D. degrees in computer system architecture from the Science and Research Branch, Islamic Azad University, Tehran, Iran, in 2005 and 2008, respectively. He is currently an Associate Professor with the Iran University of Medical Sciences, Tehran. He has made a significant contribution to the advancement of knowledge in his area of expertise, with more than 150 publications and 4000 citations. His research interests include information technology, data mining, big data analytics, e-commerce, e-marketing, and social networks. He is also an associate editor for internationally reputed journals.