IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 28 March 2023, accepted 9 April 2023, date of publication 12 April 2023, date of current version 15 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3266562

APPLIED RESEARCH

Detecting Malware Activities With MalpMiner: A Dynamic Analysis Approach

MUSTAFA F. ABDELWAHED [1,2], **MUSTAFA M. KAMAL** [2], **AND SAMIR G. SAYED** [2,3]

[1] Department of Computers and Systems Engineering, Faculty of Engineering, Helwan University, Helwan, Cairo 11792, Egypt
[2] Egyptian Computer Emergency Readiness Team (EG-CERT), National Telecom Regulatory Authority (NTRA), Cairo 12971, Egypt
[3] Department of Electronics and Communication Engineering, Faculty of Engineering, Helwan University, Helwan, Cairo 11792, Egypt

Corresponding author: Mustafa F. Abdelwahed (mustafa.faisal@h-eng.helwan.edu.eg)

This work was supported by the National Telecom Regulatory Authority (NTRA).

ABSTRACT Day by day, malware as a service becomes more popular and easy to acquire, thus allowing anyone to start an attack without any technical background, which in turn introduces challenges for detecting such attacks. One of those challenges is the detection of malware activities early to prevent harm as much as possible. This paper presents a trusted dynamic analysis approach based on Answer Set Programming (ASP), a logic engine inference named Malware-Logic-Miner (MalpMiner). ASP is a nonmonotonic reasoning engine built on an open-world assumption, which allows MalpMiner to adopt commonsense reasoning when capturing malware activities of any given binary. Furthermore, MalpMiner requires no prior training; therefore, it can scale up quickly to include more malware-attack attributes. Moreover, MalpMiner considers the invoked application programming interfaces' values, resulting in correct malware behaviour modelling. The baseline experiments prove the correctness of MalpMiner related to recognizing malware activities. Moreover, MalpMiner achieved a detection ratio of 99% with a false-positive rate of less than 1% while maintaining low computational costs and explaining the detection decision.

INDEX TERMS Cybersecurity, artificial intelligence, answer set programming, malware behaviour detection, logic programming, emulation.

I. INTRODUCTION

Nowdays, the rapid spread of sophisticated malware samples introduce challenges to anti-malware systems concerning identifying malicious activities in order to mitigate and reduce damages. Moreover, such hurdles keep getting harder since malware selling became a business model named Malware-as-a-Service (MaaS). MaaS allows newbie attackers/individuals to carry out an all-out attack over the internet without any prior knowledge while earning low-risk money. On the one hand, Kaspersky announced that during the year 2020, more than 10% of worldwide computers' users encounter at least one sort of malware attack [1]. On the other hand, some nations' economics got also affected by cybersecurity attacks. For instance, an American oil pipeline system suffered from Darkside's ransomware which is Ransomware-as-a-Service sample, impacting the computerized pipeline equipment managing system, which forced the oil company to pay around $4.4 million worth in bitcoins [2]. Another attack on the international foreign exchange firm Travelex, causing it to shut down its online operations for more than a month leading to undisclosed damage to its bottom line [3]. According to [4], during the "COVID-19" pandemic, the number of newly developed malware samples increased by 15%. Moreover, Interpol stated, in a report published in 2020, that Palo-Alto company witnessed an increase of highly risked new domains that reached more than 40,000 domains slightly, exploiting the fact that many people were searching for info about "COVID-19" [5]. For such events, the malware detection problem receives attention in the research community; through trying to employ different techniques to mitigate malware activities effects. Moreover, it is proven that no algorithm would detect all malware variants according to [6], making the

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

malware problem even more challenging to the researchers. Therefore, research suggested solving this problem by analyzing the malware sample using static and dynamic analysis techniques. The primary difference between those tactics is that static analysis extracts features from the suspicious sample without executing it. In contrast, the dynamic analysis extracts other features when executing this sample in a controlled environment. One popular static analysis technique for solving this problem is through Signature-based methods. Such methods are powerful versus known samples, but they fail against modified/unknown specimens. Therefore, malware authors deploy evasion tactics like code obfuscation, encryption or packing to existing malware samples to avoid getting recognized by signature-based systems [7], [8], [9]. Another approach uses heuristic-based algorithms; those algorithms use rules to overcome the previously mentioned limitation by searching for instructions that symbolize malicious intentions. Moreover, the heuristic-based approaches introduced false-positive detections since such rules may match benign applications [10]. Formal methods are another version of heuristic-based algorithms for recognizing malware samples; they rendered their way in malware detection by employing model checking for malicious movements methodologies [11], [12], [13]. Still, some obfuscation techniques can easily defeat such an approach. Furthermore, heuristics approaches require redevelopments and modifications to identify new families. Other researchers suggested replacing those procedures with machine learning classifiers [14]. Those classifiers can predicate whether a sample is malicious or not based on its training which acts as a knowledge base. Conversely, dynamic analysis approaches execute the suspicious sample in a controlled environment and trace the invoked Application Programming Interfaces (APIs). Next, passes this trace to a machine learning model to recognize any potential malware activities if they exist [15]. Even though dynamic analyses are resilient to code obfuscation, it introduces a potential threat where the samples could escape this controlled environment and infect the host machine.

The significant differences between the static and dynamic analysis methods are computational resources, detection ratio, and code obfuscation sensitivity. The static analysis procedures have low time and memory consumptions, with an average detection ratio and are highly vulnerable to code obfuscation techniques [16], [17], [18], [19]. On the other hand, dynamic analysis methods have higher time and memory exhaustion with a higher detection ratio besides being better resistant to code obfuscation algorithms [20]. For that particular reason, researchers employ a hybrid approach by merging both static and dynamic analysis to solve the malware problem while maintaining reasonable computational resources [21], [22], [23].

Although researchers have achieved notated malware detection ratios using machine learning techniques in static and dynamic analysis [14]. Still, machine learning model development inherits problems like feature selection and extraction, which imposes a challenging responsibility when developing detection models and influences performance and detection ratio [24]. Another dilemma, those techniques need continuous retraining each time new malware or a variant is released. Such training costs an expensive computational time to maintain a robust model precision besides not guaranteeing what the model has learned in this training phase. Moreover, it is challenging to decide which machine learning model to use; besides optimizing its parameters, it can directly affect the detection ratio and computational resources [25], [26]. Finally, machine learning models assume a closed world scenario, the dataset; therefore, they only learn what the dataset contains [27], stripping it from incomplete inferences. Further, models used to recognize sequences are bound to fixed input size, making them sensitive to long-term persistence attacks.

Our motivation in this paper is to present a trusted framework that inherits the same performance compared to machine learning while avoiding acquiring their development challenges. Therefore, we introduce the Malware-Logic-Miner (MalpMiner) framework, which employs logic programming to describe the given executable binary behaviour based on dynamic analysis. Such leverage provides a robust methodology for representing correct binary behaviours through a human-readable representation, which, in turn, achieves a machine learning similar detection ratio. Besides the human-readable representation, MalpMiner uses non-monotonic reasoning to withdraw conclusions upon receiving new information, thus acquiring commonsense reasoning abilities. Finally, the suggested procedure maintains low computational costs concerning time and space complexities. To the best of our knowledge, no researcher has applied logic programming similar to the proposed approach.

We organise this paper by presenting a background knowledge in section II, related work is presented in section III, followed by problem formulation in section IV and the recommended approach in section V. Section VI describes the conducted experiments trails' results, while section VII holds the discussion of the trails' results. Finally, section VIII concludes the current work and discusses potential future work.

II. BACKGROUND

A. MITRE ATT&CK–MALWARE BEHAVIORS

MITRE ATT&CK is a knowledge database, founded in Sep 2013, in which it describes cyber adversary behaviour [28]. MITRE ATT&CK defines a set of generic tactics in which any malicious software exhibits at least one of those tactics. This paper shows that MalpMiner abilities to recognize behaviours like process injection, evidence removal, service stop, persistence, and anti-analysis techniques deployed on an MS Windows machine, adopting a similar selection scheme as [29]. However, the suggested model can be easily opened to include more behaviours and is not limited to the stated

ones; besides, those behaviours act as a proof of concept for using logic programming for malware activities detection.

MITRE defines the *process injection* behaviour as executing arbitrary code in the address space of a separate live process; the malware has to target another process and then gain access to its memory. To perform such action the malware may use kernel APIs like *OpenProcess* API to gain access to a selected process, as for *evidence removal*; one definition from MITRE is removing left behind artefacts after the intrusion activity. Moreover, it defines *service stop* as disabling or stopping services through killing or suspending their processes. Concerning *persistence*, MITRE explains it as the actions took to keep live access across restarts, changed credentials, and other interruptions that could cut off their access. Regarding anti-analysis techniques, MITRE categorizes them into several classes based on their performed actions. Debugger detection is a well-known anti-analysis technique where the malware checks if it is being debugged or not. One implementation may check the return of a kernel API named *IsDebuggerPresent*. Based on the return of this function, the malware learns whether it is being analyzed.

B. SANDBOXES VS EMULATORS

In order to extract a program description dynamically, the program must be executed in a controlled environment to avoid harming the victims' machines. There exist two methodologies for this task. The first strategy is to imitate a natural environment with slight changes to control the execution flow to defeat anti-analysis techniques while extracting the needed program description values. Such an approach is a heavyweight approach since it executes complete OS functionalities within a running OS. On the other hand, Emulators emulate what a binary needs to complete its execution but with no actual code execution. Therefore, emulation has a low-cost computation with more flexibility in controlling the environment (i.e. controlling the return values for a function call) than performing virtual machines. However, such low-cost computation is associated with another hidden cost: developing an enormous number of the OS's kernel APIs to prevent the executing binary from crashing.

C. LOGIC PROGRAMMING

A Logic program defines any given problem's description by a set of rules and constraints, while facts are a problem instance. In other words, Logic programming uses symbolic logic to describe and solve problems. A logic program considers a problem as a theorem from a different perspective, and the problem instance is some axioms, and logic programming solvers are theorem provers. In turn, those solvers try to prove the problem's hypothesis (i.e. description) using those axioms (i.e. facts), and in the case of the solvers, they were able to provide proof rather than, in turn, finds the problem's solution.

This research's strategy employs logic programming to represent a given program by a set of detected behaviours through Answer Set Programming (ASP) programs [30]. ASP is a knowledge representation language with roots in logic and provides nonmonotonic reasoning capabilities (i.e. allowing the removal of assumptions or conclusions), making it ideal for commonsense reasoning. The chief construction element in an ASP program is an atom/rule, expressed in the form *Head* $\leftarrow$ *Body*, stating that the *Head* holds if the *Body* holds. Furthermore, ASP programs are case-sensitive. Consequently, variables start with uppercase while constants begin with lower cases.

Regarding solution finding, ASP program implementation follows a generate-and-test methodology. In the first step, a grounder algorithm generates all possible variables values; afterwards, the solver utilizes the available constraints to generate a set of correct solutions for a given query. It is encouraged to refer to [31], [32], and [33]; for a deeper understanding of the ASP workflow.

In order to clarify how ASP works, the graph colouring problem will be solved using ASP. First, the graph colouring problem is quoted as "Given n colors, find a coloring distribution for the vertices of a graph such that no two adjacent vertices are colored using the same color". Logic program.1 started defining the graph coloring problem with the colors themselves (line:1) followed by a coloring rule asserting that a node can only have one color (lines:2-4) and ending with a constraint stating that no two adjacent nodes can have the same color (line:5).

Logic program 1 $\pi_{Graph-Coloring}$

```
1: color(green). color(red).
   color(blue).
2: coloring(X, green) :- node(X), not
   coloring(X, red), not coloring(X,
   blue).
3: coloring(X, red) :- node(X), not
   coloring(X, green), not coloring(X,
   blue).
4: coloring(X, blue) :- node(X), not
   coloring(X, green), not coloring(X,
   red).
5: :-coloring(X1,C), coloring(X2,C),
   edge(X1,X2).
```

After defining the problem's domain, ASP solvers can receive problem instances to solve like a logic program.2 starting with number of nodes (line:1) and the graph structure (line:2).

Logic program 2 $\pi_{Graph-Coloring-Instance}$

```
1: node (1..4).
2: edge(1, 2). edge(1, 3). edge(3, 2).
   edge(3, 4).
```

Finally, the ASP solver will generate at least one solution set, if it exists, for the passed problem in the following format:

Output: 3 $\pi_{Graph-Coloring} \cup \pi_{Graph-Coloring-Instance}$

```
node(1) node(2) node(3) node(4)
edge(1,2) edge(1,3) edge(3,2)
edge(3,4)
color(green) color(red) color(blue)
coloring(1,red) coloring(2,blue)
coloring(3,green) coloring(4,blue)
```

D. HEURISTIC-BASED SCANNERS

The primary approach for antiviruses to identify malicious specimens is malware analysts' signatures; Therefore, several marks may identify the same sample. VirusTotal presented YARA, a tool used to recognize malicious samples [34], leaving the analysts' leading problem. A rule describes a sample. Each rule consists of metadata, strings, or binary to look for and conditions that must be satisfied to trigger a rule match, making it a perfect tool for capturing samples, not a defined behaviour. On the other hand, FireEye's Capa-Rule tool is another rule-based detector that analyses event logs then maps them to the MITRE matrix [35].

III. RELATED WORK

References of [36], [37], [38], and [39] presented tremendous research emerged to detect malicious behaviours, though according to our best knowledge, logic programming did not receive much attention when solving the malware problem. Therefore, this section presents relatively related research directions covering the malware detection approaches and logic programming developments.

A. RELATED WORK ON MALWARE DETECTION

Authors in [40] suggested an n-gram approach for detecting malware samples; even though such an approach is intuitive, yet it computational costs are unbearable. Thus, most researchers utilize machine learning classifiers to detect malicious software by encoding the given sample into a feature vector and passing it to a trained model. Therefore, the feature selection and extraction introduces a challenging task since it directly influences the model's performance concerning detection ratio and response time under the assumption that the optimal machine learning model is selected for the task.

From the static analysis perspective, [41] suggested modelling malware using a graph-based approach through creating control flow graphs then match them with known malicious files' graphs. According to the authors, such an approach was able to handle some of the basic obfuscation routines. However, such a technique is defenceless against zero-day malware samples. On the other hand, [42] proposed using text-based pattern matching techniques to search for possible malware actions by generating a subset of signatures for scanned malicious files. Those signatures are generated after applying a feed-forward bloom filter on the suspicious sample, followed by a verification process to reduce false-positives effects from the filter. Such an approach enabled handling large-sized databases of malicious files, yet it still did not provide vital results on false-positive rates [37]. Reference [43] proposed extracting the executable binary opcodes and calculated its frequency to grant such an approach higher resistance to dummy injections and changes. Though, such an approach blocks direct inference of interactions between a given binary and the system. Reference [44] listed a vast number of publications related to image process and malware detection. All listed approaches share a common representation: interpreting the executable binary as a raw image then developing dissimilarly machine learning architectures to detect malicious samples. Such a trending approach saves on computation cost, yet it may suffer from packed code.

From the dynamic analysis perspective, [45] proposed decompiling the given specimen into assembly language then extract a feature vector encoding information that reflects API calls and bytes code. Even though such an approach is intuitive, it is sensitive to obfuscation techniques in addition to the high computational cost required for decompilation and feature extraction. Reference [46] suggested using Q-learning for feature selection by providing features driven by binary's format and byte sequences, then the trained agent searches for the optimal set of features. According to their flow, the training algorithm received a 4000 vector size and reduced it to a vector of 204 elements. Still, the selected features count is significant to be used for training machine learning classifiers. Reference [47] proposed logging the API calls sequence by encoding each API into a number form, then use this sequence to teach a two-layer LSTM to capture malicious executions and by using such a model; the authors reached an accuracy of around 98%. However, they ignored the APIs arguments values since considering them will lead to exploring the input space size and may result in misinterpreting the behaviour. References of [47] and [48] presented the usage of API calls in capturing the binaries' behaviours. Nevertheless, API call extraction results in massive data collection, introducing challenges in indexing and querying such data. Moreover, some submitted to the idea of monitoring the whole system's behaviour as proposed by [49]; they claim that capturing abnormal system behaviours can ease the detection of several varieties of malware like zero-day, metamorphic, and polymorphic. A similar approach was advised by [50]; their paper presents a real-time anomaly behaviour detector using directed acyclic graphs. Their framework assumes and uses a secured data provenance as an information source to build its graph incrementally; afterwards, it learns about the host's execution behaviour and encodes it into a model similar to Markov Chain Model. Then uses the learned model as a reference to detect irregular activities; one problem with such a solution is that it assumes that no infection will occur in the training period.

B. RELATED WORK ON LOGIC PROGRAMMING

Most research uses logic programming for model checking, widely used in software engineering requirements checking.

However, some researchers utilized logic formulations to detect malware activities. For instance, [51] suggested using temporal logic to model spyware behaviours for the Android platform. Such an approach used mu-calculus to represent a sequence of actions based on extracted APIs. This technique extracted the required information smoothly due to the development of the language's nature. Another research utilized formal methods with temporal logic in order to detect banking malware targeting Android systems [11]. The authors used a similar approach to [51], which extracted the control-flow graph statically and compared it with a formalized model. However, all approaches require decompilation to check for malware activities; it can still suffer from obfuscation tactics.

Therefore, this paper suggests a trusted logic programming based framework for detecting malware activities through modelling any given binary actions into a set of behaviours that can be matched to the MITRE ATT&CK database. Hence our contributions can be summarized as:

- MalpMiner assumes an open-world assumption; such premise decouples the dependency between the model and the utilized dataset. Therefore, the behaviour recognition quality will depend only on the fact generator machine.
- MalpMiner eliminated the model training and validation process since it does not require any training of any kind. Therefore, challenges like choosing the machine learning model and optimizing its parameters are now dropped.
- MalpMiner welcomes scalability since its knowledge base can be expanded with a human-readable format to include new behaviours, making it an up-to-date system for detecting zero-days.
- MalpMiner considers the invoked APIs argument values leading to a correct behaviour modelling, unlike other approaches that ignore the arguments' values to avoid input space explosions.

IV. NOTATION, PROBLEM FORMULATION AND ASSUMPTIONS

A. PRELIMINARIES

This research approach is independent of the engaged computing machine (i.e. sandbox or emulator); therefore, $\mathcal{M}$ symbolize a computing machine. $\mathcal{M} \in \{\mathcal{S}, \mathcal{E}\}$ where $\mathcal{S}$ represents a sandbox environment while $\mathcal{E}$ denotes an emulator environment. Additionally, the machine's hooking functions and translations are denoted by $\mathcal{H}$. $\mathcal{H}$ is a table defining the map between a given API call and its equivalent generated fact(s) while $\mathcal{L}$ denotes the logic program solver. Also, the input binary file is denoted by $\mathcal{F}$ while the behavioural model is expressed as $\mathcal{B}$.

B. MALWARE BEHAVIOUR EXTRACTION PROBLEM FORMULATION

Given a five tuple $\langle \mathcal{M}, \mathcal{L}, \mathcal{B}, \mathcal{H}, \mathcal{F} \rangle$ **find** a subset of behaviours s which descibes $\mathcal{F}$ correctly, where $s \subset \mathcal{B}$.

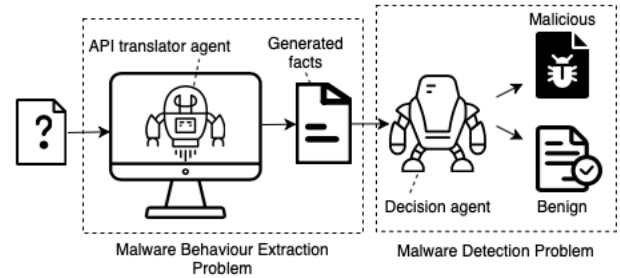


FIGURE 1. The MalpMiner's execution workflow.

C. MALWARE DETECTION PROBLEM FORMULATION

Given an extracted binary behaviour set s, **find** a subset of behaviours x which descibes a malicious behaviour(s).

D. ASSUMPTIONS

It is assumed that the agent who generates facts can resolve API arguments like dereferencing address and handlers values since most of the MS Windows kernel's APIs use handlers rather than the object's unique ID.

V. A BEHAVIOURAL EXTRACTION STRATEGY USING LOGIC-BASED TACTICS

The presented approach shows a novel methodology for using logic programming to recognize executable binaries behaviours to empower the detection of malicious software. Figure.1 shows an abstract for MalpMiner execution workflow. After the MalpMiner receives an executable binary, it executes it in a controlled environment. At the same moment, a monitoring agent translates the invoked APIs into facts, then passed them to a decision agent to decide whether this given binary is malicious or not. Moreover, Algorithm.1 defines the workflow of the suggested approach; it receives the targeted binary file, a computing machine, a behavioural model file, and a hooking map file then returns a set of detected behaviours. First, the machine is loaded into the memory; then, it initializes its APIs' hook to generate the required facts defines in the hooking map (lines:1-2). After concluding the initialization phase, the machine generates a problem instance by executing the passed binary file and creates facts based on the called APIs (line:3). The final phase is when the logic-agent infuses the logic solver in order to describe $\mathcal{F}$'s behaviour correctly (lines:4-5).

The following subsections describe the logic modelling design process for a given behaviour besides the hooking map strategy and a decision-making agent illustration followed by a runtime analysis for the submitted system.

A. BEHAVIOUR MODELLING DESIGN METHODOLOGY

As stated in section II-C, a traditional logic program contains a set of rules, constraints, and facts. In order to model a given binary's behaviour, the Σ alphabet captures the system changes that occurred when executing that binary. Therefore, Σ is determined as { *process, thread, directory,*

Algorithm 1 Behaviour-Extraction LogicAgent

Require: $\mathcal{M}$`::Execution machine,` $\mathcal{B}$`::Behavioural model,` $\mathcal{F}$`::Binary file,` $\mathcal{H}$`::Hooks map`
Ensure: `Returns a set of detected behaviours describing` $\mathcal{F}$
1: $KERNEL(Load, \mathcal{M})$
2: $INITIALISE(\mathcal{M}, \mathcal{H})$
3: $pf \leftarrow EXECUTE(\mathcal{M}, \mathcal{F})$
4: $s \leftarrow \{\mathcal{L}(\mathcal{B}, pf)\}$
5: **return** s

file, filemap, registerykey, service, window, socket, module, driver}. Besides describing the system dynamics, the Λ alphabet arrests the interactions between Σ elements and the table.1 summarises the meaning of each predicate in Λ for simplification.

1) T1055—PROCESS INJECTION BEHAVIOUR

One formal definition for a process injection behaviour is *A process injection occurs when a process P in system S gains write access to process C's memory within the same system THEN P allocates memory and write stream of bytes in C's memory*. Algorithm.2 explains a full injection process by showing how a process gets access to another process's memory. Algorithm.2's target is searching for a process *pt* with a set of properties encoded in the function *sf*; this occurs through asking the OS's kernel to share a set of all current processes in the memory (line:1). Afterwards, it searches for a targeted process by enumerating in all processes available in $\mathcal{P}$ (line:2). Upon finding the necessary process, algorithm.2 asks the kernel for accessing a selected process's memory and after obtaining access it copies the to-inject code *c* into *pv*'s memory (lines:4-7).

Algorithm 2 Process-Injection

Require: `sf::Selection Function, c::To-inject code`
Ensure: **`c`** `is injected into process` **`p`**
1: $\mathcal{P} \leftarrow \{KERNEL(EnumProcesses, ALL)\}$
2: **for** $pin\mathcal{P}$ **do**
3: **if** $sf(p)$ **then**
4: $p_{mem} \leftarrow KERNEL(OpenProcess, p)$
5: $p_{acc} \leftarrow KERNEL(AllocMem, p_{mem}, SIZE(c))$
6: $KERNEL(MemCopy, p_{acc}, c, SIZE(c))$
7: *returnsuccess*
8: **end if**
9: **end for**
10: *returnsuccess*

Based on the provided behaviour description, logic program $\pi_{process-injection}$ can model such behaviour.

Logic program 1 $\pi_{process-injection}$

```
1: targets(PT, PV) ← process(PT) ∧
   process(PV) ∧ allocatedMemory(PT,
   PV) ∧ PT ≠ PV.
2: injectedCode(PT, PV) ← process(PT) ∧
   process(PV) ∧ wroteBytes(PT, PV).
3: injectedCode(PT, PV) ← process(PT)
   ∧ process(PV) ∧ thread(T) ∧ has(T,
   queue_apc), owns(PV, T).
4: injectedCode(PT, PV) ← process(PT) ∧
   process(PV) ∧ createdFileMap(PT, F) ∧
   filemap(F) ∧ mappedFile(PT, F, PV).
5: injectedProcess(PT, PV) ←
   targets(PT, PV) ∧ injectedCode(PT,
   PV).
```

2) T1070—EVIDENCE REMOVAL

One formal definition for the evidence removal behaviour is *A process P deletes its related artefacts after executing its intrusion activities*. Hence, logic program $\pi_{self-deletion}$ forms such a behaviour.

Logic program 2 $\pi_{self-deletion}$

```
1: selfDeletion(PO) ← process(PT) ∧
   process(PO) ∧ file(PROC_FILE) ∧
   owns(PO, PROC_FILE) ∧ deleted(PT,
   PROC_FILE).
2: selfDeletion(PO) ← process(PO) ∧
   file(PROC_FILE) ∧ owns(PO, PROC_FILE)
   ∧ deleted(PO, PROC_FILE).
```

3) T1489—SERVICE STOP

One formal definition for the service stop behaviour is *A process P terminates another process V which does not belong to its subprocesses.* Accordingly, logic program $\pi_{service-stop}$ recognize such behaviour.

Logic program 3 $\pi_{service-stop}$

```
1: related(X,Y) ← owns(X,Y).
2: related(X,Y) ← owns(X,Z) ∧
   owns(Z,Y).
3: ServiceStop(PO,PT) ← terminated(PO,
   PT) ∧ process(PT) ∧ PO ≠ PT ∧ not
   related(PO, PT).
```

4) TA0003—STARTUP PERSISTENCE

One formal definition for the startup persistence behaviour is *A process P guarantees triggering its execution across system restarts by scheduling a task or using the infected system's startup events.* Therefore, logic program $\pi_{service-stop}$ captures such a behaviour.

TABLE 1. The Λ's predicates description.

#	Predicate	Meaning
1	owns(X,Y).	Fact stating that X is the parent/has unique access to Y.
2	allocatedMemory(X,Y).	Fact stating that X has allocated memory inside Y.
3	wroteBytes(X,Y).	Fact stating that X has written bytes in Y's memory.
4	has(X, queue_apc).	Fact stating that X has a user APC queued thread.
5	createdFileMap(X, Y).	Fact stating that X has create a share section/filemap named Y.
6	mappedFile(X,F,Y).	Fact stating that X has mapped a section/file to Y through F.
7	deleted(X,Y).	Fact stating that X has deleted Y.
8	startupfolder(X,Y).	Fact stating that X has placed Y in the startup folder.
9	scheduledtask(X).	Fact stating that X is a scheduled task.
10	checked(X,Y)	Fact stating that X checked on Y.

Logic program 4 $\pi_{service-start}$

```
1: startupExecutionPersistence(PO) ←
   process(PO) ∧ scheduledtask(PO).
2: startupExecutionPersistence(PO) ←
   process(PO) ∧ startupfolder(PO, F) ∧
   directory(F).
```

5) TA0005 - ANTI-ANALYSIS BEHAVIOUR

One formal definition for anti-analysis behaviour is *A process P can protect itself from analysis techniques if it can check whether a debugger is engaged or not*. Still, this definition is not complete since there are vast anti-analysis techniques; algorithm.3 highlights a simplified illustration of the decision process for evading detection belonging to a given malware. Advanced malware would execute its malicious behaviour if it did not detect any analysis tools running like debuggers. Therefore, it asks the kernel if any debuggers are engaged or not. The return value decides to execute its malicious behaviour m or evade detection using behaviours implemented in h.

Algorithm 3 AntiDebug

Require: `h::harmless behaviour, m::malicious behaviour`
Ensure: `The malware runs its malicious behaviour if not being debugged.`
1: **if** $KERNEL(Execute, IsDebuggerPresent)$ **is** TRUE **then**
2: $KERNEL(Execute, h)$
3: **else**
4: $KERNEL(Execute, m)$
5: **end if**

Based on the provided behaviour description, the logic program $\pi_{anti-analysis}$ represents such behaviour.

Logic program 4 $\pi_{anti-analysis}$

```
1: debuggerCheck(PT) ← process(PT) ∧
   checked(PT, debugging).
```

One more benefit of logic programming is merging logic programs which easies further updates/modifications with no additional cost. Moreover, a set of constraints is applied to represent a more realistic environment like a thread solely owned by one process; besides, a process can not be a thread; logic program.5 defines such constraints.

Logic program 5 $\pi_{model-constraints}$

```
1: ← process(P) ∧ thread(P).
2: ← process(P1) ∧ process(P2) ∧
   thread(T) ∧ owns(P1, T) ∧ owns(P2,
   T), P1 ≠ P2.
```

B. BEHAVIOUR EXTRACTION DESIGN METHODOLOGY

State-of-the-art Antivirus systems hook the execution machine's kernel APIs to intercept and prevent malware activities in the runtime. Therefore, the offered strategy followed the same tactic for facts generation by creating a generic map in the following expression: $(API \rightarrow FACT(A), FACT(B), \ldots)$ linking each API with a set of expected facts translation. Table.2 summarises the transformation of the kernel's APIs into the required facts.

C. BEHAVIOUR DECISION DESIGN METHODOLOGY

After splitting the malware detection problem into two subproblems, it introduced flexibility in designing various detection agents. Therefore, this subsection introduces an instance decision-making agent to clarify the malware detection procedure and provide a path for more detection agents. Another logic agent receives the extracted facts by algorithm.1 and tries to infer any malware activity from them. Algorithm.1 formalise the agent's workflow. First, the logic-agent injects the logic solver in order to infer any malware activities described in $\mathcal{D}$ (line:1), then the agent replies with a *malicious* statement in case it detected any malware activities; otherwise, reply with a *benign* declaration.

According to the deploying institute's security standard, the mentioned decision agent infers attacks through the logic program.5.

Logic program.5 classified the anti-analysis attack as a suspicious activity to reduce false positives due to misleading information since legitimate applications may deploy anti-analysis procedures to protect the intellectual property. Another case is that forming a decision-making agent is

TABLE 2. Some MS Windows kernel APIs translations.

#	API	Translation Facts
1	Initialization process	process($uuid_A$). owns($uuid_B$). file($uuid_B$).
2	OpenProcess	process($uuid_A$). process($uuid_B$). targets($uuid_A, uuid_B$).
3	VirtualAllocEx	allocatedMemory($uuid_A, uuid_B$).
4	WriteProcessMemory	wroteBytes($uuid_A, uuid_B$).
5	CreateRemoteThread	process($uuid_A$). thread($uuid_B$). created($uuid_A, uuid_B$). owns($uuid_A, uuid_B$).
6	OpenThread	process($uuid_A$). thread($uuid_B$). owns($uuid_A, uuid_B$).
7	QueueUserAPC	process($uuid_A$). thread($uuid_B$). owns($uuid_A, uuid_B$). has($uuid_B, queue_apc$).
8	NtOpenFile	process($uuid_A$). file($uuid_B$). deleted($uuid_A$, $uuid_B$). systemfile($uuid_B$).
9	NtCreateFile	process($uuid_A$). file($uuid_B$). deleted($uuid_A$, $uuid_B$). systemfile($uuid_B$).
10	NtSetInformationFile	process($uuid_A$). file($uuid_B$). deleted($uuid_A$, $uuid_B$). systemfile($uuid_B$).
11	TerminateProcess	process($uuid_A$). process($uuid_B$). terminated($uuid_A, uuid_B$).
12	NtTerminateProcess	process($uuid_A$). process($uuid_B$). terminated($uuid_A, uuid_B$).
13	IsDebuggerPresent	process($uuid_A$). checks($uuid_A$, debugging).
14	CheckRemoteDebuggerPresent	process($uuid_A$). checks($uuid_A$, debugging).

Algorithm 4 Behaviour-Decision LogicAgent

Require: s::Extracted behaviours, $\mathcal{D}$::Decision model

Ensure: Returns a *malicious* if $\mathcal{D}$ detects at least one malicious behaviour, otherwise *benign*

1: $s \leftarrow \{\mathcal{L}(\mathcal{D}, s)\}$
2: **if** $s \not\subset \{\text{malicious}\}$ **then**
3:     **return** *benign*
4: **end if**
5: **return** *malicious*

Logic program 5 $\pi_{decision-agent}$

```
1: mitre(PT, attack_T1055) ← injectedProcess(PT, PV).
2: mitre(PT, attack_TA0003) ← startupExecutionPersistence(PT).
3: mitre(PT, attack_T1070) ← selfDeletion(PT).
4: mitre(PT, attack_T1489) ← ServiceStop(PT, PV).
5: mitre(PT, attack_TA0005) ← debuggerCheck(PT).
6: malicious(PT) ← mitre(PT, attack_T1055).
7: malicious(PT) ← mitre(PT, attack_TA0003).
8: malicious(PT) ← mitre(PT, attack_T1070).
9: malicious(PT) ← mitre(PT, attack_T1489).
10: suspicious(PT) ← mitre(PT, attack_TA0005).
```

an added complex problem that is out of the scope of this paper. Nevertheless, it is vital to manifest how the submitted malware detector scheme utilizes the MalpMiner framework.

D. RUNTIME ANALYSIS

Regarding runtime analysis, it can be empirically expected that the deployed computing machine's time complexity will be bounded accordingly; this suggested approach has several bottlenecks. The first bottleneck is the used machine model, and the second bottleneck is the developed logic model. Furthermore, we can safely state that the engaged machine bounds the time and space complexity since the developed logic model avoids variables explosion since $\mathcal{B}$ checks the values generated from given rules like *injectedProcess*. Nevertheless, the Ω asymptotic complexity function is utilized to compute the time and space complexities of the offered procedure.

Any given emulator $\mathcal{E}$'s goal is to emulate instruction sets for a selected architecture. Therefore, emulation time depends directly on the number of instructions for a given binary executable. Hence, $\mathcal{E}$'s time complexity is bounded by $\Omega(n)$, where n is the number of instructions to be executed. For that particular reason, emulators are implemented using high-speed languages like C/C++ [52]. Concerning space complexity, $\mathcal{E}$ emulates what the executable binary needs to execute without physically executing it. Therefore, the proposed algorithm's space complexity is bounded by $O(c+m)$, where c is a constant representing $\mathcal{E}$'s memory allocation cost and m denotes the number of loaded modules. It is challenging to compute its cyclomatic complexities about the traditional sandbox environment since it executes complete OS functionality.

TABLE 3. Dike Dataset Statistics.

#	Filetype	Count
1	PE32-EXE	7576
2	PE32-DLL	910
3	PE32-Driver	463
4	PE64-DLL	3
5	DotNet	18
6	Word	1639
7	Power Point	2
8	Excel	108
9	CDFV2	74
10	Rich Text Format	49

VI. EXPERIMENTS AND RESULTS

A. DATASET PREPARATION

This section tested MalpMiner correctness to validate the logic programming utilization using standard libraries that implement malware activities to cover the mentioned before (section:V-A). However, due to paper size limitations, we are showing one correctness test covering process injection techniques through Pinjectra[1] and Al-khaser[2] libraries. After testing the correctness of MalpMiner, DikeDataset[3] was utilized as a benchmarking dataset for the MalpMiner framework and another baselining method. However, DikeDataset was filtered out based on the file type since MalpMiner targets 32-bit portable executable files. Table.3 shows the file type distributions for this dataset.

Afterwards, a dataset was created by extracting the invoked APIs sequences and generated-facts using Binee[4] emulator. The idea behind using Binee is the flexibility to resolve the APIs arguments without maintaining complex data structures, thus leading to a set of 1399 malicious and 965 benign samples completed the execution successfully. Then Kaspersky sandbox[5] generated the MITRE ATT&CK tags for all of those samples to ground-truth their behaviours and make sure that those samples share joint operations like startup persistence, self-deletion behaviours, and process injection.

B. BASELINING

The suggested procedure is baselined against Fireeye's CAPA rules [35] to present the fundamental performance difference between MalpMiner and CAPA. Concerning the ASP inference, MalpMiner used clingo[6] engine to recognize behaviours.

C. RESULTS

Table.4 shows correctness results; this table contains three attributes as follow: *Detected*, *Missed*, and *Crashed*. Detected and Missed attributes imply that a given behaviour is identified or not, while Crashed signifies that a sample started execution though did not finish successfully due to runtime issues.

Moreover, Table.5 shows the MITRE ATT&CK tags distribution for baselining dataset while table.6 shows the baseline tags between CAPA and MalpMiner.

Table.7 shows the MalpMiner performance results.

VII. DISCUSSION

A. OUTCOMES

What differentiates the MalpMiner from any other machine learning approach is the inference scheme. MalpMiner considers the values of the passed arguments for the invoked APIs, thus modelling the executable binary's behaviours correctly. Another ability is that ASP is based on nonmonotonic reasoning, which means MalpMiner can withdraw conclusions upon receiving new facts, thus enabling commonsense reasoning, leading to a similar performance for human expert decisions. Moreover, MalpMiner is not influenced by the utilized datasets since it does not require any training. Besides, expanding the knowledge base to include more behaviours is more straightforward than other methods while guaranteeing no loss in previous behaviours. Such capacities make MalpMiner resilient to new malicious families/variants since it searches for known behaviours, not an API calling sequence.

On the other hand, machine learning models ignore the values of the arguments passed to APIs, thus making feature selection a challenging task. Therefore, feature misselection may result in behaviour recognition misinterpretations, leading to reducing the detection ratio. The reason behind ignoring the arguments is to try to limit and encode as much information without exploding the input space. Another standard limitation across machine learning-based approaches is the adopted dataset's assumption of a closed world scenario. Therefore, researchers split the dataset into train and test sets in order to overcome such a limitation. Nevertheless, such an approach may result in different learning patterns, resulting in varying detection ratios. Accordingly, machine learning models are still vulnerable to new families/variants due to such an assumption (i.e. the utilized dataset). Moreover, adding new families/variants to the machine learning model knowledge will result in a retaining which cannot ensure that there will be information loss.

Concerning heuristic-based approaches, they have a remarkable detection rate when it comes to non-obfuscated samples. Though each rule may cover one or more samples count, not a given behaviour, making it sensitive to metamorphic and polymorphic techniques allowing obfuscation techniques to evade them smoothly. Moreover, such rules may generate a false-positive alarm since it may hit with a legitimate application requiring human intervention to suppress such an alarm. However, without any insurance, such intervention may reduce the detection ratio.

Table.4 proves the correctness of the proposed approach even though some samples have failed to finish execution successfully. As for the missed techniques, the samples crashed

[1] https://github.com/SafeBreach-Labs/pinjectra
[2] https://github.com/LordNoteworthy/al-khaser/tree/master/al-khaser
[3] https://github.com/iosifache/DikeDataset
[4] https://github.com/carbonblack/binee
[5] https://www.kaspersky.com/enterprise-security/malware-sandbox
[6] https://github.com/potassco/clingo

TABLE 4. The process injection correctness test, note: techniques are named after the library's implementation.

Library	#	Technique	Detected	Missed	Crashed
Pinjectra	1	WindowsHook	-	✓	✓
	2	CreateRemoteThread	✓	-	-
	3	CreateRemoteThread	✓	-	-
	4	SuspendThread/SetContext/ResumeThread	✓	-	-
	5	QueueUserAPC	✓	-	-
	6	CtrlInject	✓	-	✓
	7	ALPC	✓	-	-
	8	PROPagate	✓	-	-
	9	SuspendThread/ResumeThread	✓	-	✓
	10	SetWindowLongPtrA	✓	-	-
	11	SuspendThread/ResumeThread	-	✓	✓
	12	ProcessSuspendInjectAndResume	✓	-	-
Alkhaser	1	CreateRemoteThread_Injection	✓	-	-
	2	SetWindowsHooksEx_Injection	✓	-	-
	3	NtCreateThreadEx_Injection	✓	-	-
	4	RtlCreateUserThread_Injection	✓	-	-
	5	QueueUserAPC_Injection	✓	-	-
	6	GetSetThreadContext_Injection	✓	-	-

TABLE 5. Baseline Dataset Tags Statistics.

#	Tag	Count
1	T1203	656
2	T1518.001	16
3	T1112	252
4	T1548.002	15
5	T1546.001	7
6	T1543.003	4
7	T1547.001	1
8	T1562.001	26
9	T1555	396
10	T1055.002	3
11	T1036.005	8
12	T1490	7
13	T1070.004	289

TABLE 6. Targeted Tags Baseline.

#	Tag	Kaspersky	CAPA	MalpMiner
1	T1055	3	0	3
2	TA0003	0	0	1251
3	T1070	289	1	19
4	T1489	0	0	0
5	TA0005	0	0	0

before executing their process injection procedure, unlike mode 6 in the Pinjectra library, which injected the code then crashed. Such a case shows the power behind modelling the behaviour correctly, which considers the invoked APIs values, thus enabling it to recognize malware activities. Another correctness proof is provided by table.6; T1055 indicates the behaviour recognition abilities for MalpMiner when it comes to actual samples by detecting process injection activities. Moreover, MalpMiner was able to detect persistent startup behaviours (TA0003) while Kaspersky sandbox ignored them; the reason behind this is that those samples evaded the sandbox analysis. As for the self deletion techniques (T1070), the difference between Kaspersky and MalpMiner is that MalpMiner translated one technique for self deletion APIs as a proof of concept. However, such a detection rate proves the correctness of MalpMiner. As for CAPA's detections was reduced due to multiple factors like the used samples; most of them were packed. Besides that, CAPA tries to infer Mitre tags based on static analysis, unlike MalpMiner, which monitors the binary's execution while considering the invoked APIs arguments.

Table.7 strengthens the idea of MalpMiner performance concerning false-positive rates and accuracy. The logic program.4 introduced a false-positive detections which appeared in tag TA0003 (FPR in table.7). The analysis showed that 11 samples were legitimate installers that could be allowed, concluding that Λ must include $signed(uuid_a)$ fact indicating if the organization safelists the inspected file or not. Such a scenario shows that MalpMiner detection ratio performance can be improved by expanding its translation table without losing any behaviour techniques. On the other hand, machine learning approaches will need to retain including the new information without guaranteeing that it will not lose any pretrained information.

B. LIMITATIONS AND SUGGESTED SOLUTIONS

MalpMiner methodology may suffer from scalability issues from two primary aspects. One is the need to translate every kernel API into a fact to capture every change in the system. Such scalability is common since most industrial state-of-the-art antiviruses use the same procedure by hooking a set of kernel APIs to monitor system and user changes. However, targeted OS influences such scalability; the kernel APIs count

TABLE 7. KPIs benchmarking scores. Note: F1 denotes the F1 score while FPR and FNR are acronyms for False Positive/Negative Rates.

#	Model	Accuracy	F1	FPR	FNR
1	MalpMiner	0.9943	99.53%	≈ 1%	0%

is limited for a Linux OS. Besides, each API has no internal state machine making its behaviour unchanged based on the inputs, unlike Windows OS since its APIs behaviours depend directly on the passed arguments. Therefore, model assumptions and weak constraints can reduce these scalability issues by enabling the MalpMiner to assume missing information. Another solution is extending the kernel itself; the OS's kernel can enable callback to notify objects like process list data structure upon changes as new process creation/termination.

The second scalability is the creation of behaviour logic programs; each time a new behaviour is noticed, experts need to analyze the possible valid sequences followed by expanding the Λ alphabet in order to recognize brand-new behaviours. Since logic programming has decisive learning criteria due to justifying its inference with a readable format. Therefore, rather than modelling by hand logic programs per behaviour, inductive logic programming can teach MalpMiner using good and bad examples using frameworks like [53].

VIII. CONCLUSION AND FUTURE WORK

This paper introduced a trusted logic-based framework named Malware-Logic-Miner (MalpMiner), which utilizes logic programming as an executable binary's behaviour representation scheme, thus allowing for an explainable human-readable format with an expert-level malicious behaviour detection ratio. The motivation behind using another domain to represent the malware behaviours was driven by [54], who encoded music notes to colours which enabled AI to generate human-level music tracks. Consequently, the MalpMiner can recognize any given binary behaviour by translating its invoked APIs while resolving their arguments to describe what changed in the system allowing for human-expert level detection.

MalpMiner baselined results prove the success of logic programming as a representation scheme that introduces an intuitive domain for capturing binaries executables behaviours. Furthermore, MalpMiner achieved an outstanding performance while avoiding the challenges associated with machine learning model developments. The idea of an open-world assumption, nonmonotonic reasoning, and considering invoked APIs arguments made it possible for MalpMiner to utilize commonsense reasoning to recognize any given binary's activities correctly. Moreover, the no need for training removed the need and the influence of training datasets to control the detection ratio and allowed scaling up the knowledge base to include new behaviours. Nevertheless, after recognizing any given binary, MalpMiner can be easily expanded to notice intentions enabling autonomous agents to devise recovery plans in case of detected outrage activities like the ransomware infliction.

ACKNOWLEDGMENT

The authors would like to thank the EG-CERT for its support by providing resources, such as hardware, software, and data used in this research, and also would like to thank Mohamed A. Abdelmonim, Ahmed B. Sallam, and Mohammed E. Mousa from the EG-CERT's malware analysis team for implementing the required changes in the Binee emulator to integrate it with the logic model.

Its contents are solely the authors' responsibility, and they do not necessarily represent the official views of the NTRA.

APPENDIX

AUXILIARY ALGORITHMS APPENDIX

See Algorithm 6.

Algorithm 6 INITIALISE

Require: `e::Emulator, m::hooks map`
Ensure: `Updates e::emulator with the required hooks.`

1: **for** `h` **in** `m` **do**
2:     *AddHook*(h, e)
3: **end for**

Output: 7 $\pi_{Pinjectra-mode-3}$

```
module(uuid_0 × 2118d000).
module(uuid_0 × 271f4000).
module(uuid_0 × 24444000).
load(uuid_0xfff,uuid_0 × 24444000).
load(uuid_0xfff,uuid_0 × 25b1c000).
created(uuid_0xfff,uuid_0 × 0).
module(uuid_0 × 400000).
load(uuid_0xfff,uuid_0 × 400000).
load(uuid_0xfff,uuid_0 × 271f4000).
createdFileMap(uuid_0xfff,uuid_0 × 0).
targets(uuid_0xfff,uuid_0 × 0).
mappedFile(uuid_0xfff,uuid_0×0,uuid_0×
0).
has(uuid_0 × 0,uuid_0xa000983b).
is(uuid_0 × 0,remotely_created).
load(uuid_0xfff,uuid_0 × 238d8000).
load(uuid_0xfff,uuid_0 × 2118d000).
module(uuid_0 × 25b1c000).
mappedFile(uuid_0xfff,uuid_0 ×
0,uuid_0xfff).
module(uuid_0 × 238d8000).
owns(uuid_0 × 0,uuid_0 × 0).
thread(uuid_0xa000983b).
filemap(uuid_0 × 0).
process(uuid_0 × 0).
process(uuid_0xfff).
```

APPENDIX A
A CASE STUDY

This section presents a complete case study when testing MalpMiner's correctness with Pinjectra. In this case study, we selected one process injection technique, mode 3, to present how MalpMiner operates. In mode 3, Pinjectra injects the payload using *MapViewOfFile* followed by *NtMapViewOfSection* APIs; upon executing the sample with the desired mode, the API translator agent keeps a trace of the invoked APIs then generate facts for them based on their argument values. Output.7 program shows the translation of the API into a fact; note that % indicates a comment in logic programs.

Upon facts generation, the decision agent unions both logic programs.7 and 5 to detect any malware activities as shown in output.8.

Output: 8 $\pi_{Pinjectra-mode-3} \cup \pi_{decision-agent}$

```
injectedProcess(uuid_0xfff,uuid_0×0).
mitre(uuid_0xfff, attack_T1055).
malicious(uuid_0xfff).
```

REFERENCES

[1] (2020). *Kaspersky Security Bulletin 2020. statistics*. [Online]. Available: https://go.kaspersky.com/rs/802-IJN-240/images/KSB_statistics_2020_en.pdf

[2] M. Sparkes, "How do we solve the problem of ransomware?" Tech. Rep., 2021.

[3] R. Reynolds, "The four biggest malware threats to U.K. businesses," *Netw. Secur.*, vol. 2020, no. 3, pp. 6–8, Mar. 2020.

[4] *Deloitte*. [Online]. Available: https://www2.deloitte.com/ch/en/pages/risk/articles/impact-covid-cybersecurity.html

[5] *Interpol*. [Online]. Available: https://www.interpol.int/Crimes/Cybercrime/COVID-19-cyberthreats

[6] F. Cohen, "Computer viruses: Theory and experiments," *Comput. Secur.*, vol. 6, no. 1, pp. 22–35, 1987.

[7] A. A. Mawgoud, H. M. Rady, and B. S. Tawfik, "A malware obfuscation ai technique to evade antivirus detection in counter forensic domain," in *Enabling AI Applications in Data Science*. Springer, 2021, pp. 597–615.

[8] T. Alsmadi and N. Alqudah, "A survey on malware detection techniques," in *Proc. Int. Conf. Inf. Technol. (ICIT)*, Jul. 2021, pp. 371–376.

[9] S. Mahdavifar and A. A. Ghorbani, "Application of deep learning to cybersecurity: A survey," *Neurocomputing*, vol. 347, pp. 149–176, Jun. 2019.

[10] M. Brengel and C. Rossow, "YARIX: Scalable YARA-based malware intelligence," in *USENIX Secur. Symp.*, 2021, pp. 1–19.

[11] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone, "Formal methods for Android banking malware analysis and detection," in *Proc. 6th Int. Conf. Internet Things, Syst., Manage. Secur. (IOTSMS)*, Oct. 2019, pp. 331–336.

[12] E. A. Emerson and C. S. Jutla, "Tree automata, Mu-Calculus and determinacy," in *Proc. 32nd Annu. Symp. Found. Comput. Sci.*, 1991, pp. 368–377.

[13] E. A. Emerson, "Model checking and the Mu-Calculus," *DIMACS Ser. Discrete Math.*, vol. 31, pp. 185–214, Jun. 1997.

[14] Y. Lin and X. Chang, "Towards interpreting ML-based automated malware detection models: A survey," 2021, *arXiv:2101.06232*.

[15] X. Huang, L. Ma, W. Yang, and Y. Zhong, "A method for Windows malware detection based on deep learning," *J. Signal Process. Syst.*, vol. 93, nos. 2–3, pp. 265–273, Mar. 2021.

[16] 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.

[17] B. Bashari Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: From encryption to metamorphism," *Int. J. Comput. Sci. Netw. Secur.*, vol. 12, pp. 74–83, Jan. 2012.

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

[19] Y. Oyama, "Trends of anti-analysis operations of malwares observed in API call logs," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 1, pp. 69–85, Feb. 2018.

[20] K. P. Subedi, D. R. Budhathoki, and D. Dasgupta, "Forensic analysis of ransomware families using static and dynamic analysis," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 180–185.

[21] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *J. Comput. Virology Hacking Techn.*, vol. 13, no. 1, pp. 1–12, 2017.

[22] M. S. I. Sajid, J. Wei, M. R. Alam, E. Aghaei, and E. Al-Shaer, "DodgeTron: Towards autonomous cyber deception using dynamic hybrid analysis of malware," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, Jun. 2020, pp. 1–9.

[23] Y. K. B. M. Yunus and S. B. Ngah, "Review of hybrid analysis technique for malware detection," in *Proc. IOP Conf. Mater. Sci. Eng.*, vol. 769, no. 1. Bristol, U.K.: IOP Publishing, 2020, p. 012075.

[24] R. Ashmore, R. Calinescu, and C. Paterson, "Assuring the machine learning lifecycle: Desiderata, methods, and challenges," *ACM Comput. Surveys*, vol. 54, no. 5, pp. 1–39, Jun. 2022.

[25] C. Rudin, C. Chen, Z. Chen, H. Huang, L. Semenova, and C. Zhong, "Interpretable machine learning: Fundamental principles and 10 grand challenges," 2021, *arXiv:2103.11251*.

[26] C. Esposito, G. A. Landrum, N. Schneider, N. Stiefl, and S. Riniker, "GHOST: Adjusting the decision threshold to handle imbalanced data in machine learning," *J. Chem. Inf. Model.*, vol. 61, no. 6, pp. 2623–2640, Jun. 2021.

[27] G. Fenza, M. Gallo, V. Loia, F. Orciuoli, and E. Herrera-Viedma, "Data set quality in machine learning: Consistency measure based on group decision making," *Appl. Soft Comput.*, vol. 106, Jul. 2021, Art. no. 107366.

[28] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, "MITRE ATT&CK: Design and philosophy," Tech. Rep., 2018.

[29] H. Manocha, A. Srivastava, C. Verma, R. Gupta, and B. Bansal, "Security assessment rating framework for enterprises using MITRE ATT&K matrix," 2021, *arXiv:2108.06559*.

[30] C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge, U.K.: Cambridge Univ. Press, 2003.

[31] V. Lifschitz, *Answer Set Programming*. Springer, 2019.

[32] T. Eiter, G. Ianni, and T. Krennwallner, "Answer set programming: A primer," in *Reasoning Web International Summer School*. Springer, 2009, pp. 40–110.

[33] E. Erdem, M. Gelfond, and N. Leone, "Applications of answer set programming," *AI Mag.*, vol. 37, no. 3, pp. 53–68, 2016.

[34] V. M. Alvarez, "YARA documentation," Tech. Rep., 2020.

[35] FireEye. *Fireeye's Capa-Rule*. [Online]. Available: https://github.com/fireeye/capa-rules

[36] D. Dasgupta, Z. Akhtar, and S. Sen, "Machine learning in cybersecurity: A comprehensive survey," *J. Defense Model. Simul.*, vol. 19, no. 1, 2020, Art. no. 1548512920951275.

[37] J. Singh and J. Singh, "A survey on machine learning-based malware detection in executable files," *J. Syst. Archit.*, vol. 112, Jan. 2021, Art. no. 101861.

[38] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques," *Comput. Sci. Rev.*, vol. 32, pp. 1–23, May 2019.

[39] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Comput. Secur.*, vol. 81, pp. 123–147, Mar. 2018.

[40] M.-J. Lim and Y.-M. Kwon, "Efficient algorithm for malware classification: N-gram MCSC," *Int. J. Comput. Digit. Syst.*, vol. 9, no. 2, pp. 179–185, Jan. 2020.

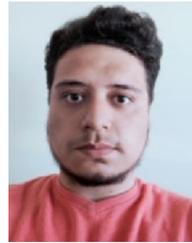
[41] D. Bruschi, L. Martignoni, and M. Monga, "Code normalization for self-mutating malware," *IEEE Security Privacy*, vol. 5, no. 2, pp. 46–54, Mar. 2007.

[42] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "SplitScreen: Enabling efficient, distributed malware detection," *J. Commun. Netw.*, vol. 13, no. 2, pp. 187–200, Apr. 2011.

[43] S. Rezaei, A. Afraz, F. Rezaei, and M. R. Shamani, "Malware detection using opcodes statistical features," in *Proc. 8th Int. Symp. Telecommun. (IST)*, Sep. 2016, pp. 151–155.

[44] R. Komatwar and M. Kokare, "A survey on malware detection and classification," *J. Appl. Secur. Res.*, vol. 16, no. 3, pp. 1–31, 2020.
[45] D. Gibert, C. Mateu, and J. Planes, "HYDRA: A multimodal deep learning framework for malware classification," *Comput. Secur.*, vol. 95, Aug. 2020, Art. no. 101873.
[46] Z. Fang, J. Wang, J. Geng, and X. Kan, "Feature selection for malware detection based on reinforcement learning," *IEEE Access*, vol. 7, pp. 176177–176187, 2019.
[47] F. O. Catak, A. F. Yazı, O. Elezaj, and J. Ahmed, "Deep learning based sequential model for malware analysis using windows exe API calls," *PeerJ Comput. Sci.*, vol. 6, p. e285, Jul. 2020, doi: 10.7717/peerj-cs.285.
[48] C.-M. Chen, G.-H. Lai, T.-C. Chang, and B. Lee, "Detecting pe-infection based malware," in *Proc. Future Inf. Commun. Conf.* Cham, Switzerland: Springer, 2020, pp. 774–781.
[49] M. E. Ahmed, S. Nepal, and H. Kim, "MEDUSA: Malware detection using statistical analysis of system's behavior," in *Proc. IEEE 4th Int. Conf. Collaboration Internet Comput. (CIC)*, Oct. 2018, pp. 272–278.
[50] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "UNICORN: Runtime provenance-based detector for advanced persistent threats," 2020, *arXiv:2001.01525*.
[51] F. Fasano, F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone, "Spyware detection using temporal logic," in *Proc. 5th Int. Conf. Inf. Syst. Secur. Privacy*, 2019, pp. 690–699.
[52] N. A. Quynh and D. H. Vu, "Unicorn: Next generation cpu emulator framework," *BlackHat*. 2015.
[53] A. Cropper, "Efficiently learning efficient programs," Ph.D. dissertation, Imperial College London, London, U.K., 2017.
[54] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, and I. Sutskever, "Jukebox: A generative model for music," 2020, *arXiv:2005.00341*.

MUSTAFA F. ABDELWAHED is currently a Teaching Associate in computers and systems engineering with the Faculty of Engineering, Helwan University, besides with the EG-CERT. He started his career working in embedded systems then switched to artificial intelligence. His research interests include AI, multi-agents, behavioral modeling, and autonomous agents.



MUSTAFA M. KAMAL is currently pursuing the degree with the Faculty of Petroleum and Mining Engineering, Suez University. His enthusiasm drove him to self-learn binary analysis, reverse engineering, and software engineering, leading him to join the EG-CERT as a Malware Analyst.

SAMIR G. SAYED received the B.Sc. degree from the Electronics, Communication, and Computer Engineering Department, Faculty of Engineering, Helwan University, in 1996, the M.Sc. degree, in 2003, and the Ph.D. degree in electronics and electrical engineering from UCL, U.K., in 2010. He is currently an Associate Professor in electronics and communication engineering with the Faculty of Engineering, Helwan University. He is also the Executive Director of Cyber-Attacks Detection and Early Warning Systems with the EG-CERT. His research interests include AI and ML applications related to malware analysis, intrusion detections, and wireless communication systems security (5G and beyond).