YAMME: a YAra-byte-signatures Metamorphic Mutation Engine

Antonio Coscia, Vincenzo Dentamaro, *Member, IEEE*, Stefano Galantucci, Antonio Maci, and Giuseppe Pirlo, *Senior Member, IEEE*

***Abstract*— Recognition of known malicious patterns through signature-based systems is unsuccessful against malware for which no known signature exists to identify them. These include not only zero-day but also known malicious software able to self-replicate rewriting its own code leaving unaffected its execution, namely metamorphic malware. YARA is a popular malware analysis tool that uses the so-called YARA-rules, which are built to match malicious contents within files or network packets analyzed by an Anti-Virus engine. Sometimes such content is expressed in the form of a byte-signature, i.e., a sequence of operational machine-level code. However, these can be bypassed since malware obfuscation techniques can change these sequences, rewriting them in several equivalent forms. This paper presents YAMME, a YARA-byte-signatures Metamorphic Mutation Engine to strengthen rules against some malware obfuscation techniques deployed in metamorphic mutation engines. First, it rewrites YARA-bye-signatures in several equivalent ways, as a metamorphic mutation engine would do. Second, an optimization phase exploits the YARA-rules syntax constructs to provide several rules formats, making them suitable for different real-world application requirements. YAMME rules have been evaluated on MWOR, G2, NGVCK, and MetaNG datasets, resulting in a better detection rate than that achieved by YARA-rules generated through AutoYara. Furthermore, an analysis of computational overhead required by different YAMME rules formats validates the low impact introduced by the mutation engine at the YARA-rules level.**

***Index Terms*— YARA, metamorphism, malware obfuscation, metamorphic malware detection, AutoYara, metamorphic mutation engine.**

I. Introduction

MALWARE represents the most significant cybersecurity threat due to the exponential growth of cyberinfections caused by their wide spread [1]. Some examples of popular malware are: Ransomware that encrypts victim data so that they cannot be accessed until a ransom amount is paid [2]; Hardware-Trojan that can execute malicious actions in the background [3]; Worm that can self-replicate to infect the highest possible number of assets in a computer network [4]. Therefore, developing defense measures capable of efficiently analyzing malicious samples is critical [5]. Malware analysis is usually divided into two main categories, i.e., static and dynamic analysis strategies [6]. During dynamic analysis the behavior exhibiting by the malware is observed [7]. However, this approach is not applicable in fast communication computer networks due to the latency introduced by the time required to perform the analysis itself. As a consequence, static analysis tools such as Anti-Virus (AV) are typically used since these can provide quick feedback. An AV engine typically employs a scanner that analyzes a file or the payload of a network packet. It uses a set of rules capable of intercepting malware based on text strings or binary patterns that it contains [8]. Yet Another Recursive Acronym (YARA) [9] is a static analysis tool widely used by several commercial AV and intrusion detection or prevention systems [10]. This led the scientific community to propose several innovative mechanisms to generate YARA-rules [11], [12], [13], [14], [15] and optimize YARA-scanner performance [16]. Moreover, according to [17], well-crafted YARA-rules can intercept malware obfuscated using anti-static analysis techniques [18]. A YARA-rule uses a simple C-based syntax and consists of two main sections: *strings,* i.e., a list of ASCII or hexadecimal (HEX) patterns or regular expressions, searched by the AV scanner into the analyzed item; *condition*, i.e., the relationship between the strings to be met to trigger the rule.

Static analysis methods, such as YARA, are very accurate in intercepting known malicious patterns, as shown in several studies [19], [20]. On the other hand, the same are not very effective to detect zero-day [21]. Furthermore, AV employs signature-based methods having low tolerance to malware variability. Therefore, a new malware signature must be developed to detect malware variants [22]. This weakness is exploited by malware writers, who develop mutation engines able to evade known AV signatures. Among the different techniques used by malware authors for generating variants, a metamorphic mutation engine rewrites the malware assembly (ASM) code using some obfuscation techniques, without altering the sample malicious scope [23]. In this regard, let

Manuscript received 23 September 2022; revised 31 May 2023; accepted 4 July 2023. Date of publication 10 July 2023; date of current version 31 July 2023. This work was supported in part by the Fondo Europeo di Sviluppo Regionale Puglia Programma Operativo Regionale (POR) Puglia 2014–2020-Axis I-Specific Objective 1a-Action 1.1 (Research and Development)-Project Title: CyberSecurity and Security Operation Center (SOC) Product Suite by BV TECH S.p.A., under Grant CUP/CIG B93G18000040007. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Ghassan Karame. *(Corresponding author: Stefano Galantucci.)*

Antonio Coscia and Antonio Maci are with the Cybersecurity Laboratory, BV TECH S.p.A., 20123 Milano, Italy (e-mail: a.coscia@bv-tech.it; a.maci@bv-tech.it).

Vincenzo Dentamaro, Stefano Galantucci, and Giuseppe Pirlo are with the Department of Computer Science, University of Bari Aldo Moro, 70125 Bari, Italy (e-mail: vincenzo.dentamaro@uniba.it; stefano.galantucci@uniba.it; giuseppe.pirlo@uniba.it).

Digital Object Identifier 10.1109/TIFS.2023.3294059

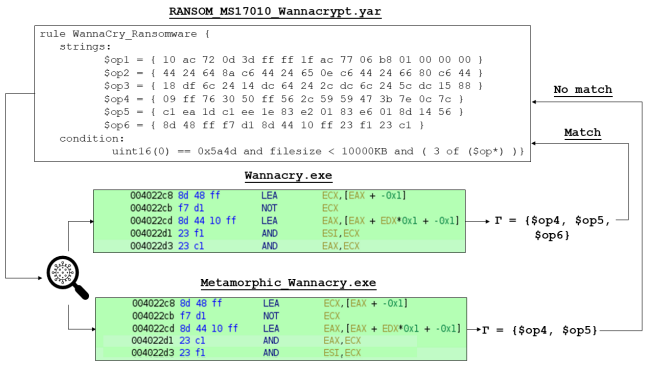


Fig. 1. YARA-rule failure on a metamorphic variant of WannaCry malware.

Figure 1 be considered. The rule `Wannacry_Ransomware` in `RANSOM_MS17-010_Wannacrypt.yar` of the YARA-rules repository [24] has been considered. In detail, the Hexadecimal (HEX) strings that represent byte-patterns, i.e., ASM code instructions, are taken into account. The aforementioned YARA-rule is tested on a Wannacry sample downloaded from Virushare [25]. Moreover, a metamorphic variant of WannaCry has been generated to show the differences in the results obtained by scanning both malware with the same YARA-rule. Such results are denoted with Γ to indicate the set of binary patterns found. According to its condition, the rule matches if $|\Gamma| = 3$. As a consequence, the metamorphic variant of malware can bypass the YARA-rule by applying a simple malware obfuscation technique, such as ASM code instruction permutation [26] on the HEX string identified by $\$op6$. Therefore, a YARA-byte signature can be vulnerable to such general obfuscation techniques, and it is verified for both public rulesets [24] and those obtained by automatic generation algorithms, such as AutoYara [13]. In fact, according to [27], automatically generated YARA-rules can require post-processing actions to make them suitable for addressing a specific security problem.

This paper presents YAra-byte-signatures Metamorphic Mutation Engine (YAMME), an algorithm employing general metamorphic obfuscation techniques to strengthen YARA-rules against evasion attempts. Therefore, the proposed contribution comes as a defense-as-attack mechanism that perturbs known YARA-byte-signatures using the same techniques that the generic metamorphic mutation engine would use as a self-protect detection method. However, metamorphic engines often combine several techniques to make AV detection difficult. The application of sequential obfuscation methods results in several malware mutations. As a consequence, the resulting perturbed YARA-byte-signature will be composed of the same number of HEX strings. However, low computational overhead is mandatory regardless of the post-processing technique introduced to improve the effectiveness of the YARA-rules [27]. Therefore, to compact the strings produced, YAMME embeds three optimization algorithms, which differ in the result produced in terms of optimized rule readability and computational overhead required.

The main contributions provided by this paper are:

- it presents YAMME, an innovative metamorphic mutation engine for enhancing YARA-byte-signatures against malware obfuscation techniques;
- it provides an analysis of:
  - detection performance achieved by YAMME rules on four different metamorphic malware families;
  - computational performance achieved by YAMME rules during a scanning process performed through the YARA-scanner.

TABLE I
SUMMARY OF MAIN NOTATIONS USED IN THIS PAPER

Symbol	Description
Γ	The set of patterns matched by a YARA-rule.
Σ	A nonempty finite charset, i.e., an alphabet.
e	A regular expression.
$L(e)$	The regular language represented by e.
Υ_{INIT}	The YARA-rule passed as input into YAMME.
$\mathcal{B}$	The set of byte-signatures extracted from Υ_{INIT}.
$\mathcal{V}$	The vector of the numeric encoded instruction.
p_{ADM}	The number of admissible permutations.
CPM	The Constrained Permutation Matrix.
Υ_{PERT}	The output of the YAMME perturbation phase.
Ω	The set of HEX strings contemplated by Υ_{PERT}.
m	The number of replaceable instructions.
r	The number of replacements.
t	The number of overall transformations.
$\Upsilon_{LD_{OPT}}$	The YAMME rule obtained from optimization approach described in Section IV-B1.
$U_{LD_{OPT}}$	The set of undesired strings introduced by the approach described in Section IV-B1.
$\mathcal{O}_{LD_{OPT}}$	The set of HEX strings contemplated by $\Upsilon_{LD_{OPT}}$.
$\Upsilon_{LCS_{OPT}}$	The YAMME rule obtained from optimization approach described in Section IV-B2.
$\mathcal{O}_{LCS_{OPT}}$	The set of HEX strings contemplated by $\Upsilon_{LCS_{OPT}}$.
$U_{LCS_{OPT}}$	The set of undesired strings introduced by the approach described in Section IV-B2.
$\Upsilon_{FSM\_TO\_RX_{OPT}}$	The YAMME rule obtained from optimization approach described in Section IV-B3.
z	The malware index associated with each variant introduced during experiment V-A.
η	The number of variants generated in experiment V-A.
$\mathcal{T}$	The number of transformations introduced for each z-th variant.
$\mathcal{G}$	The set of HEX strings contemplated by a YARA-rule examined in experiment V-C.
$\mathcal{CPU}$	The average percentage of CPU consumption.
$\mathcal{R}$	The average percentage of RAM consumption.
$\mathcal{D}$	The disk space required.
$\mathcal{S}$	The time required by the scanning process.

The remainder of this paper is organized as follows. Section II provides a literature review on metamorphic malware generation and detection techniques, and YARA; furthermore, it defines the motivation of this paper. In Section III, the theoretical framework used for the proposed contribution is presented. Section IV illustrates YAMME, describing how perturbation and optimization phases work. Section V provides the experimental settings, that is, the description of methods and materials used to evaluate YAMME. Then, the results and their discussion are reported in Section VI. Section VII discusses the feasibility of the proposed technique. Section VIII concludes the paper and draws possible future works. Table I summarizes the main notations used in this paper.

II. Related Work and Motivation

A. Metamorphism

1) Metamorphic Malware Generation Methods: The randomness in malware evolution is due to the tools employed by the malware authors to implement several functions according to the malicious scope of the malware family to which the sample belongs. Generally, malware variants can be automatically generated using polymorphism or metamorphism [28]. Polymorphic malware embeds a compiler capable of decompiling or decrypting the sample to mute. Then, the viral patterns searched by static analysis tools are encrypted using a new random encryption before recompiling the malware variant. However, the viral code is decrypted in memory, which favors detection methods [29]. On the other hand, a metamorphic malware is able to rewrite its ASM code in an equivalent way by applying obfuscation methods, such as instruction permutations, garbage code insertion, or equivalent instruction replacement [30]. This is done using the so-called metamorphic mutation engine [29]. In [31], the Genetic Algorithm (GA) is used to create a metamorphic malware variant, inserting a set of instructions into opportune ASM listing blocks to change the code representation, i.e., to bypass static analysis based on byte-pattern matching. Evolutionary algorithms as metamorphic mutation engines are also evaluated in [32] and [33]. In the latter case, each individual created at the end of a generation, is tested on a virtual environment to determine if it represents a potential malware variant, and is scanned using widespread AV engines. In [34], a novel framework called AMVG generates malware variants using a modifier block function that applies a GA or a machine learning model to introduce random perturbations. The modification schema is defined according to the input file that can be the malware ASM code. In this case, the above cited transformations are applied. The sample is then validated through behavior analysis to determine that a new metamorphic malware variant has been obtained. Furthermore, the Reinforcement Learning (RL) paradigm has recently been explored for generating malware variants [35]. In [36], *ADVERSARIALuscator* exploits a Deep RL agent such as Proximal Policy Optimization (PPO) to create feasible obfuscations at ASM code level. Analogously, PPO is used in [37] and [38] to generate metamorphic malware variants. In MERLIN [39], REINFORCE and Deep Q-Network are used to train a mutation system capable of evading AV engines and machine learning classifiers, considering metamorphic malware, obfuscation techniques. In [40], a metamorphic worm engine is presented that implements garbage code insertion and ASM instruction replacement focusing on MOV and XOR substitutions. The effectiveness of XOR-based instruction replacement technique has been recently discussed in [41]. In [42], a phylogeny model built on code permutations technique is presented to point out the similarity between malware samples when permuted variants of programs are compared.

2) Metamorphic Malware Detection Methods: Detecting malware obfuscated using metamorphic mutation techniques represents a fascinating research topic. In [43] , an algorithm called MetaAware for identifying metamorphic malware is presented. This system detects malware variants through similarity comparison between disassembled samples. This comparison is performed considering the function calls and system libraries used by the two programs. In [44], an approach called MEDUSA addresses metamorphic malware detection by creating a signature based on Application Programming Interface (API) call sequences extracted from behavioral analysis. In [45], the metamorphic malware detection problem has been tackled by computing the structural entropy similarity between malware variants. This method was developed in [46] and consists of two main phases. First, the analyzed files are segmented using entropy and wavelet analyses. The second stage computes a similarity metric between the obtained objects. The results obtained show that such a method can effectively detect metamorphic malware. In [47], the metamorphic malware detection problem is addressed using a graph matching algorithm, which is based on APIs called during program execution. Starting from a database of known malware API call graphs, a distance metric is used to compute the similarity between unknown and known API call graphs. This method achieves promising performance in the detection of various types of metamorphic malware. MOMENTUM [48] represents a metamorphic malware detection methodology based on the use of signatures defined by Multiple Sequence Alignment (MSA) algorithms. The detection and false positive rates obtained are very promising. Reference [49] proposes a metamorphic detection method based on entropy and the number of repeated ASM instructions. First, the examined sample is disassembled; second, the instruction occurrence matrix stores the occurrence frequencies of a generic instruction; and finally, such a matrix is sent to a machine learning algorithm to classify the sample as malicious or not. The results obtained show the effectiveness of this detection method. A. G. Kakisim et al. [23] introduced two methodologies to identify metamorphic malware, both based on co-opcode graphs, i.e., graphical data structures created for a singular malware family. Entropy-based distance measure is used in [50] to determine the degree of metamorphism in four different malware families. This distance is passed to K-Nearest Neighbors to classify metamorphic malware. The results obtained are encouraging. In [51], several works that use a combination of Hidden-Markov-Models (HMMs) with other algorithms are evaluated according to the features considered. This collection reveals how in each case the use of HMM results in encouraging metamorphic malware classification accuracy values. R. Mirzazadeh et al. [52] combine opcode graph similarity with linear discriminant analysis to tackle metamorphism. In [53], a novel framework for metamorphic malware analysis called MARD (presented in [54]) is used. It exploits the Malware Analysis Intermediate Language introduced in [55]. The method uses a combination of control flow graph and sliding window of difference, and control flow weight techniques to build a behavior signature for detecting metamorphic malware in real-time. The Non-negative Matrix Factorization technique has been used in [56] to detect metamorphic malware, leading to better results than previous similar-paradigm techniques. The same technique has been combined with an HMM model in [57] to improve

metamorphic malware detection performance. Wang et al. [58] discuss how metamorphic malware evolves, describing the concept drift introduced by metamorphic mutation engines. A training phase on properly extracted and processed data is required for some of the most effective methods, i.e., Artifical Intelligence-based algorithms, which are able to minimize such concept drift. On the contrary, a signature-based static analysis tool such as YARA needs a proper ruleset covering as many malware variants as possible.

B. YARA

YARA has been studied in several research contributions due to its flexibility and widespread. In [59], a tool called MASSE performs intrusion detection using YARA as a malware analysis tool. In [60], a clustering algorithm aimed at intercepting malware variants is presented. Such an algorithm runs downstream of initial filtering performed by scanning with YARA-rules. Biclustering algorithms are extended in [13] for the automatic generation of YARA-rule given a set of samples sharing similar properties. In particular, a variant of the SpectralCoClustering algorithm has been developed using the Variational Gaussian Mixture Model algorithm, which allows prior knowledge of the number of clusters. In [11], a GA-based process is used to generate YARA-rules. The fitness function is defined according to the detection rate of the produced rule, and the goal is to maximize such a score while generating candidate rules. The generated rules obtained promising detection rate scores. In [14], an automatic YARA-rule generation algorithm has been presented, which uses an executable sample as input and identifies relevant malicious patterns based on a pre-trained Naïve Bayes. The proposed framework results in better detection performances than the compared approaches. M. Belaoued et al. [61] proposed a novel feature selection method working on the generic API call sequence derived from dynamic analysis. In particular, given two sequences, the LCS was used to extract common APIs that are not necessarily contiguous. In addition, the generic API was ranked according to its Term Frequency-Inverse Document Frequency (TF-IDF) score. As a consequence, two different types of YARA-rules are obtained: the first considers API sequences extracted by the LCS; the second takes into account the API having the highest TF-IDF value. The obtained YARA-rules resulted in promising detection accuracy scores. In [15], YaraML models, i.e., Logistic Regression and Random Forest (RF), are leveraged to rank dynamic features, such as the API call sequence and the API parameters, extracted from the sample behavioral analysis, and thus generate a set of YARA-rules. In the case of RF, the YARA-rule is generated considering each single rule (associated to each tree in the forest) and combining them at the condition level through a permutation of multiple trees. The method results in effective malware detection performance against malicious Windows samples. In [62], YARA is combined with import and fuzzy hashing methods to perform ransomware analysis. Based on the performance obtained, fuzzy rulers are embedded with the YARA-rule in [63], [64], and [65] to extend the range of conditions admissible by the YARA-rule. In [27], a fuzzy hashing method is proposed to improve the detection rate of automatically generated YARA-rules. Among all the experiments, the results obtained reveal performance improvement in intercepting four ransomware categories due to the YARA-rules enhancement using the proposed method. In [66], a Bidirectional Encoder Representations from Transformers (BERT) model is exploited to generate YARA-rules. In particular, given an input sample, the rule strings are extracted, and then the most relevant strings are selected by the BERT model to reduce the false positive rate. The BERT-based rules achieve better detection performance than benchmark approaches.

C. Motivation

From the state-of-the-art review, it emerged that recent researches propose strengthening methods for detecting metamorphic malware, which are mainly related to algorithms generalization ability of different Artificial Intelligence-based algorithms. The introduction of a signature generalization mechanism that is not dependent on the learning phase (which could be vulnerable to adversarial attacks capable of altering rules effectiveness, as shown in [67], [68], and [69]) may prove to be a viable alternative. In addition, such a system may prove helpful for enhancing rules manually generated by security analysts, which is still widely used in practice [70]. Furthermore, according to the review of the literature on metamorphic malware detection and YARA usage, and to the best of our knowledge, no work considers enhancing the YARA-rules against metamorphic malware by mutating them using malware obfuscation techniques. This motivates the proposal of this research, that is, the presentation of a defense-as-attack method that provides a mutation engine for YARA-rules replicating common low-level code transformation techniques. In particular, mutations resulting in known OPCODE mutations are considered.

III. Background

A. ASM OPCODE

An operation code (OPCODE) identifies an ASM instruction according to the specific CPU architecture.

These are typically encoded using HEX strings, such as: `CWD` $\rightarrow$ 0 $\times$ 99; `INTO` $\rightarrow$ `0xCE`: `LEA` $\rightarrow$ `0xBD`; `LOCK` $\rightarrow$ `0xF0`; `NOP` $\rightarrow$ 0 $\times$ 90; `POPF` $\rightarrow$ 0 $\times$ `9D`; `PUSHF` $\rightarrow$ 0 $\times$ `9C`; etc.

As described above, a metamorphic mutation engine generates an equivalent malware form by rewriting its ASM code through some methods that leave unaffected the program execution. However, these transformations result in OPCODE mutations, which result in ineffective YARA-byte signatures. In this paper, some metamorphic mutation techniques discussed in [26], [30], and [49] are considered. Note that these represent some general techniques widely deployed by metamorphic mutation engines [29].

B. Metamorphic Malware: Overview of Obfuscation Methods

In this section, a discussion of some malware obfuscation techniques used by malware authors for generating variants

is provided. To better outline YAMME applicability, such techniques are categorized according to whether they result in a priori known OPCODE mutation.

1) Not Known OPCODE Mutations:

- Inlining process: this technique consists of replacing a function call with the corresponding function body. As a consequence, given an HEX string that has a semantic item corresponding to a function call, it is not possible to know a priori the new OPCODE sequence associated to the instructions within the function body.
- Outlining process: this obfuscation technique is the dual of the above one. Therefore, a call instruction replaces function body sequence.

2) Known OPCODE Mutations:

- *Instruction Permutation* (IP): this obfuscation technique swaps the instructions in the listing without altering the correct execution flow. This is achieved by satisfying dependency relationships between the operand registers according to the following statement [26].
  Definition 3.1 (Dependency relationship): Given a listing such as the following: `op1 R1, R2; op2 R3, R4.`
  It can be stated that two instructions are permutable if the following conditions are met: (i) $R1 \neq R2$; (ii) $R1 \neq R4$; (iii) $R2 \neq R3$.
- *Instruction Replacement* (IR): this obfuscation technique substitutes instructions with equivalent ones [49], [71].
- *Register exchange* (RE): this technique consists of replicating an ASM flow by changing the registers used [49]. In fact, an instruction typically contains one or more identifiers for the operands on which the operation is performed.
- *Garbage Code Insertion (GCI)*: this obfuscation technique adds no-effect operations between the instructions in a listing. This does not alter the correct program execution but results in the OPCODE sequence mutation.

Several examples of the effects of such techniques on OPCODE sequences are reported in Table II. YAMME can be adapted to each transformation resulting in a priori known OPCODE mutation. To make detection more challenging, metamorphic engines frequently mix two or more of these methods [49]. Several malware mutations can be obtained by applying these techniques sequentially, along with other obfuscation techniques leading to opcode sequence metamorphoses. As a consequence, the resulting YARA-byte signature will consist of the same number of HEX strings. To reduce this number, the usage of regular expressions has been considered since these are admissible by the YARA syntax.

C. Regular Expressions and Finite State Machines

The following are some basic definitions of formal language theory derived from [72], which will be used to optimize YARA-rules.

Definition 3.2 (Alphabet): An alphabet Σ is a nonempty finite set of symbols (characters).

Since YARA-byte-signatures are taken into account, Σ consists of the set of HEX digits, i.e. $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$.

Definition 3.3 (String or word): Given an alphabet Σ, a string or word is defined as a finite sequence of Σ characters.

Each OPCODE assigned to an ASM instruction, and an OPCODE sequence represents a word in Σ.

Observation 3.1: The set of all strings defined on the alphabet Σ, including the empty string, i.e., the string of length zero, is denoted Σ^*.

Definition 3.4 (Language): A language is a subset of Σ^*. Specifically, a language L over Σ is a set of words in Σ^*

Each possible isolated string obtained through several concatenation operations between strings results in a language L.

Definition 3.5 (Regular Expression): Regular expressions are a method of representing languages. Each regular expression e corresponds to the language that it represents.

Given the above alphabet Σ, the following example of regular expression $e = \{AAD|ABD|ACD\} = \{A(A|B|C)D\}$ can occur. In particular, it represents any of AAD, ABD, and ACD.

Definition 3.6 (Regular language): The language determined by a regular expression e is called Regular language $L(e)$. Therefore, it is a subclass of all possible languages that can be defined given Σ. In particular, given a regular expression e, and a word w, the regular language is given by $L(e) = \{w \in \Sigma^* | w \text{ matches } e\}$.

Chomsky grammars of type 3 (regular grammars) generate regular languages. A Chomsky (formal) grammar represents a set of production rules, i.e., strings replacement from left-hand side to right-hand side, where each side is given by a sequence of: terminal(non-terminal) symbols for which replacement is(not) allowed; a start symbol. Type-3 grammars have the following main properties: (i) a single non-terminal on the left-hand side; (ii) a single terminal or, a single terminal followed by a single non terminal, on the right-hand side; (iii) production rules in the form $A \rightarrow B$ or $A \rightarrow BD$, where A and B are non-terminal symbols and D is a terminal one. According to their structure, regular languages can be modeled using the Finite State Machines (FSMs).

IV. Proposed Contribution: YAMME

The proposed solution consists of two phases. The first is called the perturbation phase since the mutation engine acts on the input YARA-byte-signature to introduce the corresponding variants.

The second phase provides the choice of several optimization algorithms aimed at compacting the strings obtained in the previous phase.

Note that the optimization algorithms are implemented exploiting YARA syntax constructs for the handling of HEX strings.

A. YAMME - Perturbation Phase

YAMME perturbs the input byte-signature, according to the malware obfuscation techniques introduced in Section III-B. This section describes these IP and IR scenarios, but the same process can be extended to other mutation techniques resulting in a priori known OPCODE variation.

TABLE II
EXAMPLE OF OPCODE MUTATIONS DUE TO MALWARE OBFUSCATION METHODS IN SECTION III-B.2

Obfuscation method	Initial ASM Instruction	Initial OPCODE	Equivalent ASM Instruction	OPCODE Mutation
IP	and esi, ecx and esi, ecx; and eax, ecx.	0x23F123C1	and eax, ecx; and esi, ecx.	0x23C123F1
IR	push eax.	0x50	sub esp, 4; mov DWORD PTR SS: [esp], eax.	0x83EC04890424
	pop eax.	0x58	mov eax, DWORD PTR SS: [esp]; add esp, 4.	0x8B042483C404
	inc eax.	0x40	add eax, 1.	0x83C001
	mov eax, edx.	0x89D0	add esp, 4; push edx.	0x5258
	xor eax, eax.	0x31C0	sub eax, eax.	0x29C0
RE	pop eax; pop ecx.	0x5859	pop ebx; pop edx.	0x5B5A
GCI	and esi, ecx; and esi, ecx.	0x23F123C1	and esi, ecx; nop; and esi, ecx.	0x23C19023F1

1) Instruction Permutation: To analyze all the steps performed by the mutation engine, the following YARA-rule Υ_{INIT} generated using AutoYara [13] on the same Wannacry sample used in Figure 1, is taken into account.

```
rule Wannacry {
 strings:
  $x0 = { 42 2A 12 04 BD 9D }
 condition:
        $x0 }
```

Step 4.1 (Read Byte Pattern): First, YAMME reads the input YARA-rule and extracts the appropriate byte-pattern list $\mathcal{B}$. From the above Υ_{INIT}, $\mathcal{B}$ = {{42 2A 12 04 BD 9D}}.

Step 4.2 (Disassembly): Thus, binary patterns $b \in \mathcal{B}$ are disassembled. From the above step, b = {42 2A 12 04 BD 9D} and it is disassembled as follows: 42 → inc edx; 2A 12 → sub dl byte ptr [edx]; 04 BD → add al 0xbd; 9D → popfd.

Step 4.3 (Integer Encoding): The algorithm encodes the instructions using integer numbers as follows: inc edx → 0; sub dl byte ptr [edx] → 1; add al 0xbd → 2; popfd → 3.
As a result, the vector $\mathcal{V} = [0, 1, 2, 3]$ is obtained. All possible permutations that can be generated from $\mathcal{V}$ are equal to $|\mathcal{V}|!$. However, to avoid a loss of flow integrity, a series of n_c constraints between instructions must be defined according to the Definition 3.1. As a consequence, $|\mathcal{V}|!$ will be filtered by an n_c function, resulting in the admissible permutations p_{ADM}.

Step 4.4 (Direct Acyclic Graph Builder): Given a generic pair of instructions $x, y \in \mathcal{V}$, a constraint $[x, y]$ forces instruction x to precede instruction y. Furthermore, constraints are also generated if an operand refers to a memory address, e.g., in the case of a function call. The resulting constraints list can be modeled using a Directed Acyclic Graph (DAG), where nodes represent instructions, while arcs model the constraints to be imposed between the nodes. Given the vector $\mathcal{V}$ obtained in Step 4.3, the resulting DAG consists of: (i) a set of nodes, that is {0, 1}; (ii) the constraint [0, 1].

Step 4.5 (Compute p_{ADM}): The algorithm builds the admissible permutations matrix through the application of the topological sorting algorithm in [73]. This matrix is called the Constrained Permutations Matrix (CPM) $\in \mathbb{N}^{p_{ADM} \times |\mathcal{V}|}$.

Step 4.6 (Inverse Integer Encoding): The algorithm assigns to each CPM_{ij} item, with $i = 1, \ldots, p_{ADM}$ and $j = 1, \ldots, |\mathcal{V}|$, the corresponding instruction (in the opposite way to what was done in Step 4.3).

Step 4.7 (Assembly): Then, the algorithm assembles the instructions through the dual operation of the one performed in Step 4.2, getting the list of the new HEX strings Ω, such that $\mathcal{B} \subseteq \Omega$.

Step 4.8 (Write Signature): The algorithm outputs the perturbed form of Υ_{INIT}, using as strings the patterns in Ω. The condition is *any of them* since all strings are equivalents. As a result, the YARA-rule Υ_{PERT} is obtained:

```
rule Wannacry_Pert_IP{
 strings:
  $op00 = { 42 2A 12 04 BD 9D }
  $op01 = { 42 2A 12 9D 04 BD }
  $op02 = { 42 04 BD 2A 12 9D }
  $op03 = { 42 04 BD 9D 2A 12 }
  $op04 = { 42 9D 2A 12 04 BD }
  $op05 = { 42 9D 04 BD 2A 12 }
  $op06 = { 04 BD 42 2A 12 9D }
  $op07 = { 04 BD 42 9D 2A 12 }
  $op08 = { 04 BD 9D 42 2A 12 }
  $op09 = { 9D 42 2A 12 04 BD }
  $op10 = { 9D 42 04 BD 2A 12 }
  $op11 = { 9D 04 BD 42 2A 12 }
 condition:
  any of them }
```

2) Instruction Replacement: Considering the IR technique, given m applicable substitutions, the number of feasible mutations is given by $r = 2^m - 1$. For example, consider $m = 3$, that is, the trio of replacements $< r_A, r_B, r_C >$, such that each can be applied (1) or not (0); thus, given the initial byte-signature s_0, the remaining combinations are: $< 0, 0, 1 >$; $< 0, 1, 0 >$; $< 0, 1, 1 >$; $< 1, 0, 0 >$; $< 1, 0, 1 >$; $< 1, 1, 0 >$; $< 1, 1, 1 >$. Each of them denotes a new byte-signature.

Example 4.1 (Instruction Replacement): Consider the following listing: pop eax → 58; inc eax → 40; and eax, exc → 21 C8.

In this case, the first two instructions are considered replaceable. Therefore, since $m = 2$, the feasible mutations are: s_0 = {584021C8}; s_1 = {5883C00121C8};

$s_2 = \{8B042483C4044021C8\}$; $s_3 = \{8B042483C40483C00121C8\}$.

The recursive application of the algorithm ensures that all possible substitutions are applied for each existing permutation. Then, it is possible to compute the overall number of transformations of the starting YARA-rule:

$$t = \begin{cases} p_{ADM} \cdot r, & \text{if } m \geq 1 \\ p_{ADM}, & \text{otherwise} \end{cases} \quad (1)$$

Note that t increases with the number of malware obfuscation techniques contemplated by YAMME and applicable to the specific HEX string to perturb.

The aforementioned phases are described considering IP and IR to clearly describe how YAMME works. However, the proposed contribution embeds the metamorphic mutation techniques defined in Section III-B.2. In particular, both RE and GCI are handled using YARA wildcards, such as the placeholder character represented by the question mark.

B. YAMME - Optimization Phase

As discussed above, to preserve the computational resources, an optimization step is introduced. Without the optimization step, in all cases, a single scan will result in a single binary pattern match among all t generated. Furthermore, reducing the number of strings by compacting them through the optimization step makes the YARA-rule compliant for other AV engines, such as ClamAV [74].

In this section, several optimization algorithms are presented, with the goal of producing the smallest number of strings, but capable of covering all the t mutations.

To simplify the discussion, the following optimization algorithms are applied to Υ_{PERT} obtained from the IP mutation introduced in Section IV-A.1.

1) Logical Disjunction Based Algorithm: As described in Section IV-A.1, the instruction permutation algorithm is implemented taking into account all possible permutations of $\mathcal{V}$, satisfying the corresponding DAG. Given the CPM, it can be observed which distinct elements can occur at the $j-$th position by crossing the same matrix by columns, as highlighted in (2).

$$CPM = \begin{bmatrix} \mathbf{0} & \mathit{1} & \underline{2} & 3 \\ \mathbf{0} & \mathit{1} & 3 & \underline{2} \\ \mathbf{0} & \underline{2} & \mathit{1} & 3 \\ \mathbf{0} & \underline{2} & 3 & \mathit{1} \\ \mathbf{0} & 3 & \underline{2} & \mathit{1} \\ \mathbf{0} & 3 & \mathit{1} & \underline{2} \\ \underline{2} & \mathbf{0} & \mathit{1} & 3 \\ \underline{2} & \mathbf{0} & 3 & \mathit{1} \\ \underline{2} & 3 & \mathbf{0} & \mathit{1} \\ 3 & \mathbf{0} & \mathit{1} & \underline{2} \\ 3 & \underline{2} & \mathbf{0} & \mathit{1} \\ 3 & \mathbf{0} & \underline{2} & \mathit{1} \end{bmatrix} \quad (2)$$

C_j is defined as the set of elements that can occur at the j-th column. In this case, since $|\mathcal{V}| = 4$, the following sets are obtained: $C_1 = \{0, 2, 3\} = \{42, 04BD, 9D\}$; $C_2 = C_3 = \{0, 1, 2, 3\} = \{42, 2A12, 04BD, 9D\}$; $C_4 = \{1, 2, 3\} = \{2A12, 04BD, 9D\}$. At this stage, each obtained element is placed in a logical disjunction relation. As a result, only one string will be obtained for all strings in Ω. The optimized YARA-rule $\Upsilon_{LD_{OPT}}$ follows:

```
rule LD_OPT{
    strings:
        $opt={(42|04BD|9D)(2A12|04BD|9D
            |42)(04BD|9D|2A12|42)(9D|04BD
            |2A12)}
    condition:
        $opt }
```

$\Upsilon_{LD_{OPT}}$ covers all the cases in Υ_{PERT}, gaining simplicity and readability of the YARA-rule, but introduces many undesired HEX strings, which defines the set:

$$\mathcal{U}_{LD_{OPT}} = \{u_s : u_s \in \mathcal{O}_{LD_{OPT}} \wedge u_s \notin \Omega\} \quad (3)$$

where $\mathcal{O}_{LD_{OPT}}$ is the set of all HEX strings contemplated by $\Upsilon_{LD_{OPT}}$ strings. To adequately quantify undesired strings, the CPM dimensional structure must be considered. More precisely, let $\mathcal{V}$, and let the quantity of distinct elements that can occur starting from the second column; thus multiplying them up to the $|\mathcal{V}|$-th column. From this product, the admissible perturbations must be subtracted, resulting in the following equation:

$$|\mathcal{U}_{LD_{OPT}}| = \prod_{j=2}^{|\mathcal{V}|} |C_j| - p_{ADM} \quad (4)$$

2) LCS Based Algorithm: An alternative optimization algorithm is based on the prior use of the Longest Common Subsequence (LCS) strategy [75], which is one of the most widely used algorithms to extract common contiguity characteristics between two or more strings [76].

Each variant obtained is compared with all others using the LCS algorithm. Among the possible substrings obtained, those of greatest length deprived of the semantic element having the shortest length are selected. If more than one exists, the LCS that minimizes the number of strings obtained is chosen.

Example 4.2 (LCS between two Byte Strings): Let two pairs of strings ($\$op00$, $\$op01$) and ($\$op02$, $\$op03$) extracted from Υ_{INIT}. Applying the LCS algorithm results in $LCS(\$op00, \$op01) = 422A1204BD$ and in $LCS(\$op02, \$op03) = 4204BD2A12$, denoting the absence of $9D$ with respect to the starting strings in both cases.

The subtracted semantic element is associated with a placeholder, and its possible placement within the chosen LCS, is determined by comparing it with the initial HEX strings in Ω. If possible, the remaining semantic elements will be placed between placeholders in logical disjunction. Then, a string is built for each possible semantic element placement. A number of strings equal to the number of possible placeholder placements will be obtained. Therefore, Example 4.2 results in the two strings $\$temp\_opt01 = 42(2A12|04BD)9D(2A12|04BD)$ e $\$temp\_opt02 = 42(2A12|04BD)$

$(2A12|04BD)9D$. Given Υ_{PERT} this strategy results in the following $\Upsilon_{LCS_{OPT}}$:

```
rule LCS_OPT{
 strings:
  $opt01 = {9D(42|04BD)(42|2A12|04BD)(04
      BD|2A12)}
  $opt02 = {(42|04BD)9D(42|2A12|04BD)(04
      BD|2A12)}
  $opt03 = {(42|04BD)(42|2A12|04BD)9D(04
      BD|2A12)}
  $opt04 = {(42|04BD)(42|2A12|04BD)(04BD
      |2A12)9D}
 condition:
  any of them }
```

However, concatenating the disjunction of semantic elements introduces undesired strings, which can be quantified as follows:

$$|\mathcal{U}_{LCS_{OPT}}| = n_p \cdot \prod_{k=2}^{n_p-1} \gamma_k - p_{ADM} \tag{5}$$

where n_p represents the number of possible placements and γ is the number of distinct semantic elements in the possible alternatives, except for the first , and the set of undesired strings is computed as follows:

$$\mathcal{U}_{LCS_{OPT}} = \{u_s : u_s \in \mathcal{O}_{LCS_{OPT}} \wedge u_s \notin \Omega\} \tag{6}$$

In real-world applications, the optimized rules that embed undesired matching can be used in the first step of a multi-step scan. In such a scenario, the first step involves the execution of the command `yara -s` $\Upsilon_{LD_{OPT}}(\Upsilon_{LCS_{OPT}})$ `sample.exe`, through which the string $\delta \in \Gamma$ for which matching occurred is returned. During the second step, a search for δ within KMP, where KMP is the overall list of known malicious patterns (such that $\Omega \subset KMP$), is performed. If $\delta \notin KMP$, then δ is an unwanted match.

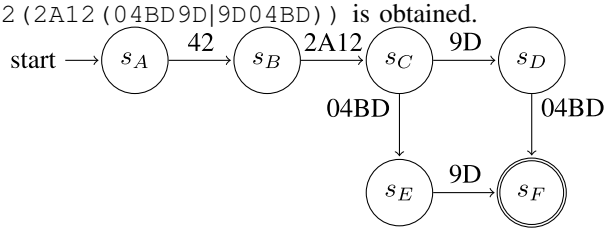
Example 4.3 (Multi-step scanning): From an implementation perspective, the multi-step scan is realized by combining $\Upsilon_{LD_{OPT}}$ or $\Upsilon_{LCS_{OPT}}$ with Υ_{PERT}; the condition will relate the optimized strings and all strings in Ω, as follows:

```
rule Multi_Step_Scanner{
 strings:
  $opt01 = {9D(42|04BD)(42|2A12|04BD)(04
      BD|2A12)}
  $opt02 = {(42|04BD)9D(42|2A12|04BD)(04
      BD|2A12)}
  $opt03 = {(42|04BD)(42|2A12|04BD)9D(04
      BD|2A12)}
  $opt04 = {(42|04BD)(42|2A12|04BD)(04BD
      |2A12)9D}
 condition:
  1~of ($opt*) and Wannacry_Pert_IP }
```

3) FSM to Regular Expression Based Algorithm: All the aforementioned optimization algorithms use regular expressions, but the derived regular languages are supersets of Ω. To avoid such a scenario and according to the definitions in Section III-C, the desired regular language can be modeled using an FSM. Therefore, given the FSM derived from the CPM, this approach aims at obtaining the regular expression e resulting in $L(e) = \Omega$.

Example 4.4 (From HEX strings to FSM): Consider the pair of strings HEX_1 = `422A1204BD9D` and HEX_2 = `422A129D04BD` representing $\$op00$ and $\$op01$ in rule Υ_{PERT}. Since Yara-bytye signatures have been considered, the alphabet Σ is the set of HEX digits. Then, from Step 4.2, it is possible to derive the word of interest given symbols in Σ, i.e. {42, 2A12, 04BD, 9D}.

Starting from an initial state s_A, the possible state transitionsare obtained by adequately concatenating the previous words until HEX_1 and HEX_2 are completed. In addition, to obtain the minimal FSM, the complete strings converge to a single final state s_F. For example, in HEX_1 and HEX_2 dcases, starting from s_A, it is possible to transit in the following states: s_B = {42}; s_C = {422A12}; s_D = {422A129D}; s_E = {422A1204BD}; s_F = {{422A1204BD9D}, {422A129D04BD}}. This can be seen by the following FSM, from which e = `42(2A12(04BD9D|9D04BD))` is obtained.



Completing the construction of the FSM for all HEX strings in Ω, the following rule $\Upsilon_{FSM\_TO\_RX_{OPT}}$ is obtained:

```
rule FSM_TO_REGEX_OPT{
    strings:
        $opt={ 42(2A12(04BD9D|9D04BD)|04
           BD(2A129D|9D2A12)|9D(2A1204BD
           |04BD2A12))|04BD(42(2A129D|9
           D2A12)|9D422A12)|9D(42(2
           A1204BD|04BD2A12)|04BD422A12)
           }
    condition:
        $opt }
```

A rule optimized in such a way does not introduce unwanted strings, since $L(e) = \Omega$. However, it shows a not simple rule readability and may require a higher computational effort than the other approaches in cases where t is larger. Finally, the optimization mechanisms introduced are still valid in any case in two binary patterns that share at least a semantic element.

V. Experimental Setup

This section provides the experimental plan, that is, the list of methods and materials used to perform three different analyses: (i) how the entropy and the number of transformations introduced by a metamorphic mutation engine are related; (ii) the detection performances achieved by YARA-rules enhanced using YAMME; (iii) the computational performances required by YAMME rules during a scanning process performed through YARA-scanner.

A. Metamorphic Mutation Engine Entropy Analysis

In this experiment, the entropy generated by a metamorphic mutation engine (MMEEA), that is, Metame [77] used in [57]. It has been selected as an alternative open source tool with respect to Address Space Layout Permutation (ASLP) [78], which was used in [43] to perform a randomization analysis of executable samples. Both tools randomly rewrite the ASM code of an executable sample. This aims at increasing the degree of randomness (or *entropy*) in the transformed low-level sample code. ASLP performs IP and RE, while Metame can additionally execute IR and GCI.

This experiment aims at analyzing the number of mutations introduced by Metame when it is used to transform the aforementioned Wannacry sample, denoted with W_0. In particular, four different tests are performed, which differ for the entropy introduced by the mutation engine when a new malware variant is created. Let z be the malware variant index:

- in MMEEA-1 η variants of W_0 are progressively generated, i.e, $W_{z+1} \leftarrow$ `run Metame on` W_z, `until` $z = \eta - 1$;
- in MMEEA-2 η variants of the same W_0 are generated, i.e., $W_{z+1} \leftarrow$ `run Metame on` W_0, `until` $z = \eta - 1$;
- in MMEEA-3 η variants of the same W_0 are generated, randomly selecting the variant to transform, i.e., $W_{z+1} \leftarrow$ `run Metame on` W_{z_r}, where $z_r = rand(0, z)$ `until` $z = \eta - 1$ and $rand(0, z)$ randomly selects an index within the range $[0, z]$;
- in MMEEA-4 η variants of W_0 are progressively generated as in MMEEA-2, to generate the lowest entropy value as possible, i.e., $W_{z+1} \leftarrow$ `run Metame -f on` W_z, `until` $z = \eta - 1$. Here, the flag -f forces the metamorphic engine to perform all feasible transformations, reducing the metamorphism entropy.

B. YAMME Detection Performance Evaluation

This experiment aims to evaluate metamorphic malware detection performances of rules when these are enhanced using YAMME. Therefore, this section provides the list of materials and methods used for such a purpose.

1) Dataset: In this paper, a dataset including both metamorphic malware and goodware executable samples was used to evaluate the effectiveness of YAMME. Table III reports the dataset sample distribution. In particular:

- Goodware samples have been downloaded from the *Portable Freeware* provider [79];
- Metamorphic malware samples collected in [53] have been considered. These are the same metamorphic malware typologies used in several researches, such as [43], [44], [45], [47], [48], [49], [52], and [53], and are categorized according to the state-of-the-art mutation engines used to generate the specific metamorphic malware family; hence: (i) 700 Metamorphic Worm engine (MWOR) [40] samples divided into 100 for seven different padding ratio values, that is, the ratio of garbage-code inserted to the worm-code; furthermore, several equivalent IRs are applied for each malware variant; (ii) 50 Second Generation virus generator (G2) [29] samples generated by modifying source code according to the aforementioned metamorphic malware obfuscation techniques and by introducing some en(de)cryption routines; (iii) 40 Next Generation Virus Construction Kit (NGVCK) [29] samples generated using obfuscation techniques, such as GCI, IP, and RE (Section III-B.2); (iv) 40 Morphing NGVCK (MetaNG) [57] samples, obtained by morphing the above NGVCK samples using Metame [77].

TABLE III
DATASET SAMPLE DISTRIBUTION

Typology	Number of samples
MWOR	700
G2	50
NGVCK	40
MetaNG	40
Goodware	4463

The number of goodwares was deliberately chosen higher than that of metamorphic malware to increase the chance of matching unwanted strings by $\Upsilon_{LD_{OPT}}$ and $\Upsilon_{LCS_{OPT}}$. Υ_{PERT}, Multi-Step, and $\Upsilon_{FSM\_TO\_RX_{OPT}}$ result in the same metric scores.

2) Metrics: To evaluate YAMME detection performance, some conventional metrics have been used. These are calculated considering malware samples as the positive class, and goodware samples as the negative one. Therefore, correct detections of positive samples are denoted by True Positive (TP). In contrast, a goodware detected as malicious represents a False Positive (FP). Finally, a positive sample undetected by the YARA-rule will be a False Negative (FN). Given such values, the following metrics have been evaluated:

- Precision (PREC), that is, how many detected metamorphic malware actually are identified as done, defined as $PREC = \frac{TP}{TP+FP}$;
- Recall or True Positive Rate (TPR). i.e., how many metamorphic malware have been detected as such with respect to the total number of positive class samples. This metric gives a clear indication of the YARA-rule detection rate, as can be seen in [47] and [53], defined as $TPR = \frac{TP}{TP+FN}$;
- F1-Score, i.e., the harmonic mean between PREC and TPR, hence $F1 - Score = 2 \cdot \frac{PREC \cdot TPR}{PREC+TPR}$;
- False Negative Rate (FNR), that is, how many malicious samples are undetected with respect to the total, therefore $FNR = \frac{FN}{FN+TP} = 1 - TPR$.

The metrics have been evaluated, varying the positive class represented by one of the four metamorphic malware types, respectively, and fixed the negative class given by goodware.

3) Algorithm Used to Generate the Input YARA-Rules for YAMME: AutoYara [13] has been selected since it is intended to generate YARA-rules for files that share some intrinsic characteristics (e.g., same malware family, such as metamorphic malware). This tool aims at finding specific byte-patterns to build specific YARA-byte signatures (note that in some cases the extracted byte-pattern refers to ASCII strings encoded using the HEX notation). In particular, it focuses on byte n-grams, where $n \in \{2^3, 2^4, \ldots, 2^{10}\}$. To achieve such a purpose, AutoYara was trained using the EMBER

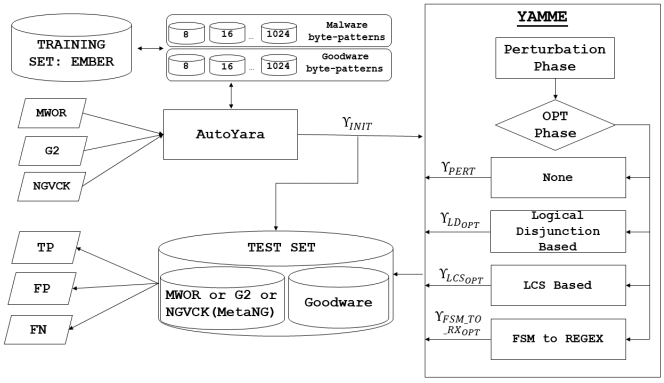


Fig. 2. Detection performance experiment overview.

state-of-the-art dataset [80] consisting of 600000 malware and goodware samples (300000 each). Thus, the algorithm selects an n-gram if it is not very frequent in the overall dataset, since building a specific rule is desired. Therefore, bloom filters are used to store such peculiar n-grams for each different value of n and for each class. Furthermore, patterns having an entropy less than one, as well as patterns having half-bytes equal to 00 or FF, are not part of the bloom filters. At this point, the improved Spectral Coclustering algorithm is used to define the relationships between the byte-patterns (features) selected during the generation process. The employment of such an algorithm is mandatory for two main reasons: (i) the number of biclusters is not known a priori but is determined automatically; (ii) to allow biclusters overlapping. Finally, patterns within the same bicluster are related by the "and" statement, while different biclusters are related by the "or" statement. The byte-patterns are selected according to their capability in intercepting the largest number of input files and to minimize the FP rate (augmenting PREC). Therefore, for a large number of input files, i.e., given as input several metamorphic variants of the same variant, a high detection rate is not guaranteed. Hence, despite being a fast and lightweight tool, post-processing mechanisms that can increase AutoYARA-rules effectiveness are required.

4) Main Differences Between AutoYara and YAMME: AutoYara is a tool used to generate YARA-rules starting from a set of malware samples; whereas, YAMME is a YARA-rules post-processing mechanism, i.e., it can generate a new YARA-rule only starting from another rule. Therefore, in this experiment, YAMME acts as a post-processing tool for AutoYara rules.

Figure 2 provides an overview of the current experiment. To evaluate detection performances achieved by YARA-rules processed using YAMME, first, three different initial YARA-rules (Υ_{INIT}), for each metamorphic malware family (excluding MetaNG) in Table III are generated through AutoYara. These are then processed using the techniques discussed in Section IV. Then, the resulting rules are tested on samples in Table III to compute the aforementioned metrics.

Note that no fine-tuning procedure has been performed on AutoYara. Therefore, as can be seen, in Figure 2, each AutoYara input sample is never-seen-before by the tool and, analogously, YAMME has no possibility of knowing precisely what transformations are actuated by the mutation engines.

TABLE IV
INTERCEPTED MALWARE-EVALUATED YARA-RULE PAIRS

Malware MD5	YARA-rule ID (Source)
84c82835a5d21bbcf75a61706d8ab549	1/2 (AG/OR)
af2379cc4d607a45ac44d62135fb7015	3 (OR)
3c877dfd0d60572be7c939c08c39866d	4 (AG)
e1a9b6f7285a85e682ebcad028472d13	5 (OR)
099ec2767271a59ae4fd2cfa9844c9bf	6 (AG)
f9bae53e77b31841235f698955aece30	7 (OR)
5de75a478ffb3aa01a88f4e539f3edc0	8 (AG)
394298eed78d455416e1e4cf0deb4802	9 (OR)
bd7bad534d1e5a2ad6c11829b96a23e4	10 (AG)

C. YAMME Rules Computational Performance Analysis

In this experiment, the following YAMME rules performances have been analyzed: the average CPU ($\mathcal{CPU}$) and RAM ($\mathcal{R}$) consumption achieved during the scanning process; the Disk Space required ($\mathcal{D}$) to store them; and the scanning process time ($\mathcal{S}$). Each score has been measured with respect to the different YARA-rules produced using different YAMME optimization approaches and using multi-step scanning. Furthermore, it has been considered how the increase in the number of strings contemplated by the YAMME rule ($|\mathcal{G}|$) could impact the overall scanning performances. As a consequence, to make such analysis more evident, the performances achieved by Υ_{INIT} have also been analyzed.

As discussed above, $|\mathcal{G}|$ is affected by the optimization algorithm considered. In particular: $\mathcal{G} = \mathcal{B}$, in case of Υ_{INIT}; $\mathcal{G} = \mathcal{O}_{LD_{OPT}}$, in case of rules optimized using IV-B.1 algorithm, i.e., $\Upsilon_{LD_{OPT}}$; $\mathcal{G} = \mathcal{O}_{LCS_{OPT}}$, in case of rules optimized using IV-B.2 algorithm, i.e., $\Upsilon_{LCS_{OPT}}$; $\mathcal{G} = \Omega$, in the case of rules: (i) no-optimized (Υ_{PERT}); (ii) multi-step scanning; (iii) optimized according to IV-B.3 algorithm, i.e., $\Upsilon_{FSM\_TO\_RX_{OPT}}$.

The analysis refers to recursive scans performed on a batch of 29366 items.

Table IV reports the list of rules (downloaded from the Official Repository (OR) [24] or automatically generated (AG) using AutoYara [13]) processed by YAMME, that intercept the associated malware sample. For visualization purposes, each rule is encoded using a numeric identifier. The average percentages of CPU and RAM usage are recorded using Python `psutil` and `os` libraries. Note that, the scanner used is the original YARA scanner ($\leq$ 3.4); hence, the achieved performances must be referred to such a tool. Therefore, to better analyze the results obtained, the following scanning mechanism must be taken into account. Aho-Corasick is the pattern matching algorithm employed by the YARA-scanner, and additional heuristics are implemented to achieve lower scan times [81]. Furthermore, YARA searches for potential matches using so-called *atoms*. Typically, these are short four-byte character sequences. The Aho-Corasick machine merely uses these substrings. Upon finding a potential match, the output function stores the atoms found in the file or memory in the list of possible matches. After creating this list, an algorithm determines whether the match holds for the entire string. To avoid overloading the checking mechanism with undesired patterns, a unique set of atoms must be chosen

to provide the complete list of matches (without losing any of them). In terms of scanning speed, this decision element is crucial. Therefore, the selection of right atoms is critical to improve scanning computational performance. For example, given `wcrypt(ransom|ware)` string within a YARA-rule. If a match exists for `cryp`, the scanner will check if it was prexifed by an `w` and continues with a `t`. If this is true, it will follow with the regex `(ransom|ware)`. In this way, running a slow regex engine is avoided. Furthermore, the scanning speed is affected by the YARA-rule condition structure, i.e., the lower the number of loop iterations within the condition, the faster the scanner.

Furthermore, the performance impact of increasing in the number of rules has been tested. In particular, all the rules in Table IV are grouped into a single rule object to ensure that each of them is correctly used.

D. Hardware Settings and Implementation

To run the experiments, an Ubuntu 20.04 machine, Intel Xeon(R) E5-1620 v4 CPU @3.50 GHz Octa-Core, 16 GB RAM, is used.

Furthermore, YAMME has been implemented in Python using: *Yaramod* [82] in Steps 4.1 and 4.8; *Capstone* [83] and *Keystone* [84] in Steps 4.2 and 4.7, respectively.

VI. Results and Discussion

A. Metamorphic Mutation Engine Entropy Analysis

Figure 3a shows that by progressively mutating the variants obtained at the previous iteration, $\mathcal{T}$ trend settles within a finite range given by $[\mathcal{T}_{min}, \mathcal{T}_{max}] = [203, 265]$, after an initial spike. The progressive application results in high entropy, as can be seen by the oscillatory $\mathcal{T}$ trend.

Figure 3b shows a $\mathcal{T}$ trend bounded in $[\mathcal{T}_{min}, \mathcal{T}_{max}] = [325, 405]$. Thus, by continuously mutating W_0, there will be a finite number of mutations introduced. The oscillatory $\mathcal{T}$ trend is attributed to high metamorphic entropy.

According to its setup, the experiment MMEEA-3 represents an average case of the two above ones. This can be seen in Figure 3c, where $\mathcal{T}$ trend is limited into the range $[\mathcal{T}_{min}, \mathcal{T}_{max}] = [267, 369]$. In fact, the $[\mathcal{T}_{min}, \mathcal{T}_{max}]$ range is very close to the arithmetic mean of the above intervals.

Finally, in MMEEA-4 (Figure 3d), the loss of metamorphic entropy is confirmed. Brute application of mutation engine results in $\mathcal{T}$ increasing. However, it repeats the same substitutions for each new variant generated, obtaining a steady trend.

The overall analysis shown in Figure 3 indicates that the entropy is a function of $\mathcal{T}$. In particular, the lower the entropy, the higher $\mathcal{T}$. However, high structural entropy is a common feature of metamorphic malware [50], but this leads to a lower number of $\mathcal{T}$. This is advantageous for YAMME, since the shorter the chain of transformations introduced, the greater the chance that this will be part of the transformations covered by the enhanced rule.

B. YAMME Detection Performance Evaluation

The goal of this section is to show how the variants of YARA-rules can intercept the variants of metamorphic malware defined in Table III. For each experiment, the results reported in Tables V-VIII describe the detection performances achieved by the related AutoYARA-rules, when these are (or not in case of Υ_{INIT}) enhanced using YAMME.

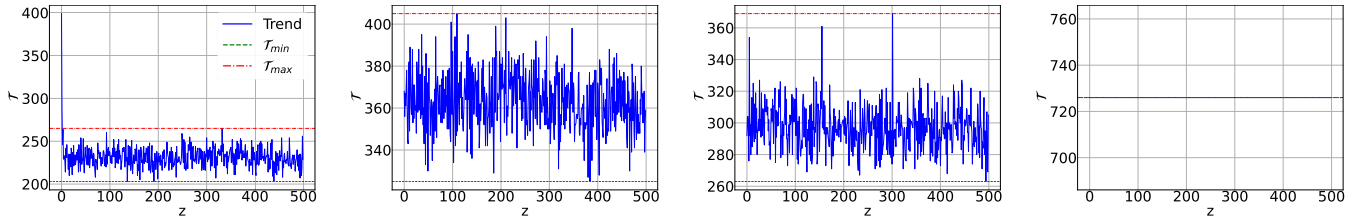
TABLE V
Detection Performances Achieved by YAMME Used to Enhance AutoYARA-Rules on MWOR

Rule	PREC	TPR	F1-Score	FNR
Υ_{INIT}	1.000	0.454	0.624	0.545
$\Upsilon_{LD_{OPT}}$	0.189	1.000	0.319	0.000
$\Upsilon_{LCS_{OPT}}$	0.308	1.000	0.551	0.000
Υ_{PERT} Multi-step $\Upsilon_{FSM\_TO\_RX_{OPT}}$	1.000	1.000	1.000	0.000

TABLE VI
Detection Performances Achieved by YAMME Used to Enhance AutoYARA-Rules on G2

Rule	PREC	TPR	F1-Score	FNR
Υ_{INIT}	1.000	0.600	0.750	0.400
$\Upsilon_{LD_{OPT}}$	0.373	1.000	0.543	0.000
$\Upsilon_{LCS_{OPT}}$	0.625	1.000	0.757	0.000
Υ_{PERT} Multi-step $\Upsilon_{FSM\_TO\_RX_{OPT}}$	1.000	1.000	1.000	0.000

Table V lists the detection performances obtained involving MWOR as malware metamorphic family. The rule generated through AutoYara (Υ_{INIT}) achieves a TPR = 45.4% (FNR = 54.5%). Therefore, given all 700 samples as input, AutoYara generates a rule capable of classifying $\sim$ 318 of them as malicious. However, this rule does not intercept any of the overall set of legitimate samples, as can be seen from the precision score achieved (100%), which positively influences the F1-Score (62.4%). The more evident YAMME contribution is given by the obtained Recall value, which is equal to 100%, making the FNR null. On the other hand, each YAMME rule results in different FPs according to its format: (i) as aforementioned discussed, the approach proposed in Section IV-B.1 generates rules optimized in the sense of unique string ($\Upsilon_{LD_{OPT}}$), however, according to Eq. (4) no-malicious patterns are intercepted, reducing Precision and F1-Score to 18.9% and 31.9%, respectively; (ii) a similar result is achieved by $\Upsilon_{LCS_{OPT}}$; however, this rule results in a lower number of FPs than $\Upsilon_{LD_{OPT}}$ since $|U_{LCS_{OPT}}| \leq |U_{LD_{OPT}}|$, obtaining Precision and F1-Score equal to 30.8% and 55.1%, respectively; (iii) the best detection performances among the compared rule formats have been achieved by Υ_{PERT} ($\Upsilon_{FSM\_TO\_RX_{OPT}}$, Multi-step) since no FPs are introduced preserving AutoYARA-rule precision (100%) and gaining in F1-Scorer (100%) due to a higher Recall than this achieved by Υ_{INIT}. Table VI reports the detection performances obtained by the evaluated YARA-rules on G2 metamorphic malware family. As can be seen, the AutoYARA-rule (Υ_{INIT}) can intercept 30 out of 50 samples using in input to generate the rule itself, resulting in TPR = 60%. Therefore, the remaining samples are incorrectly classified as legitimate, obtaining FNR = 40%. However, in this case, the number of FPs is null (100% precision), and, consequently, an F1-Score = 75%



(a) MMEEA-1 (b) MMEEA-2 (c) MMEEA-3 (d) MMEEA-4

Fig. 3. $\mathcal{T}$ trend per Wannacry variant generated with $\eta = 500$.

TABLE VII
DETECTION PERFORMANCES ACHIEVED BY YAMME USED TO ENHANCE AUTOYARA-RULES ON NGVCK

Rule	PREC	TPR	F1-Score	FNR
Υ_{INIT}	1.000	0.675	0.805	0.325
$\Upsilon_{LD_{OPT}}$	0.018	0.925	0.036	0.075
$\Upsilon_{LCS_{OPT}}$	0.055	0.925	0.109	0.075
Υ_{PERT} Multi-step $\Upsilon_{FSM\_TO\_RX_{OPT}}$	1.000	0.925	0.961	0.075

TABLE VIII
DETECTION PERFORMANCES ACHIEVED BY YAMME USED TO ENHANCE AUTOYARA-RULES ON METANG

Rule	PREC	TPR	F1-Score	FNR
Υ_{INIT}	1.000	0.675	0.805	0.325
$\Upsilon_{LD_{OPT}}$	0.018	0.925	0.036	0.075
$\Upsilon_{LCS_{OPT}}$	0.055	0.925	0.109	0.075
Υ_{PERT} Multi-step $\Upsilon_{FSM\_TO\_RX_{OPT}}$	1.000	0.925	0.961	0.075

is obtained. Conversely, the use of YAMME enables the detection of all metamorphic malware in the analyzed family, i.e., a TPR = 100% (null FNR). However, the trend of FPs obtained as the rule format generated by YAMME changes should be analyzed as follows: (i) according to Eq. (4), the rule $\Upsilon_{LD_{OPT}}$ results in the highest number of FPs among all the compared rules, leading to the lowest precision (37.3%) and F1-Score (54.3%) values; this is due to the undesired strings within $U_{LD_{OPT}}$; (ii) as a consequence of Eq. (5) , the rule $\Upsilon_{LCS_{OPT}}$ intercepts a lower number of goodwares than $\Upsilon_{LD_{OPT}}$ ($|U_{LCS_{OPT}}| \leq |U_{LD_{OPT}}|$); hence, greater precision (62.5%) and F1-Score (75.7%) with respect to the above discussed case are achieved; (iii) the best detection performances for G2 family are achieved by Υ_{PERT} ($\Upsilon_{FSM\_TO\_REGEX}$, or Multi-scan) since no FPs are introduced, therefore, the precision value obtained by Υ_{INIT} is preserved and, due to the improved Recall, an F1-Score = 100% is achieved.

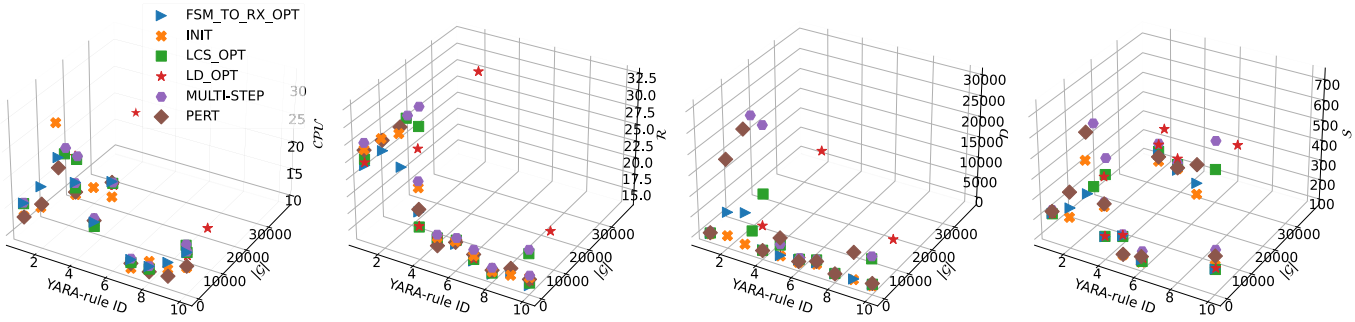
Table VII shows the detection performances achieved by scanning MWOR samples using the relative AutoYARA-rule (Υ_{INIT}) and their variants obtained through YAMME. Υ_{INIT} is able to intercept 27 out of 40 NGVCK samples, achieving a Recall = 67.5% (FNR = 32.5%) and a F1-Score = 80.5%, due to a Precision = 100% (none FPs). In this case, the YAMME-preformed enhancing mechanism on Υ_{INIT} leads to a TPR improvement of 25%; hence, the YAMME rules can intercept 37 of 40 samples tested for NGVCK. However, also in this case the other performances of the YAMME rules are related to the rule format, i.e., they are based on the applied optimization approach: (i) $\Upsilon_{LD_{OPT}}$ results in the highest number of FPs (1972 out of 4463 samples) among all compared rules on NGVCK samples, therefore, in the lowest Precision (1.8%) and F1-Score (3.6%) scores; (ii) $\Upsilon_{LCS_{OPT}}$ improves Precision and F1-Score with respect to those achieved by $\Upsilon_{LCS_{OPT}}$, but even in this case, these values are very low (5.5% and 10.9%, respectively); (iii) Υ_{PERT} ($\Upsilon_{FSM\_TO\_REGEX}$, or Multi-scan) achieves the best performances, since it can preserve the Precision score obtained by Υ_{INIT} (none of goodware samples has been classified as malicious), gaining in F1-Score, thanks to the high Recall obtained.

The results in Table VIII are obtained using the same rules evaluated in Table VII, since NGVCK samples transformed using Metame, i.e. MetaNG, are tested. As can be seen, the detection performance remains unchanged despite the subsequent application of metamorphic transformations on the same samples from which Υ_{INIT} was generated. Then, randomly mutating a sample may be ineffective as in this case, while the mutation of patterns matched by the rule is needed to bypass the rule itself (as done in Figure 1). However, YAMME would make this second strategy ineffective as well, providing a robust defense strategy to contrast these evasion attempts.

YAMME effectiveness should be attributed to the fact that the tested mutation engines, among themselves heterogeneous, use as obfuscation methods those currently contemplated by the proposed mechanism (Section III-B.2). Such a result is enforced by the classical architecture of metamorphic mutation engines [29]. Finally, if Υ_{INIT} has a number of strings such that per single sample there are many matches, then this increases the chances of the success of YAMME.

C. YAMME Rules Computational Performance Analysis

In this section, the results of the experiments defined in Table IV are discussed. In particular, Figure 4 shows the computational performances achieved by the YARA-rules for different rule formats and $|\mathcal{G}|$. Furthermore, for each metric analyzed, the maximum values of $|\mathcal{G}|$ and the involved metric are made explicit to evaluate the impact of the $|\mathcal{G}|$ growth on the scanning overhead. Figure 4a shows the average percentage of CPU consumption achieved during the overall scanning process by YARA-rules, which differ for their structure and $|\mathcal{G}|$. Among the different YARA-rules examined, Υ_{INIT} and Υ_{PERT} formats required less CPU consumption in 8 out of



(a) CPU usage (%) (b) RAM usage (%) (c) Disk Space required (in Bytes) (d) Scan time (in seconds)

Fig. 4. Yara-rules computational performance per different rule format and $|\mathcal{G}|$ achieved for a scanning process involved 29366 items.

TABLE IX

$|\mathcal{G}|$ IMPACT ON $\mathcal{CPU}$ FOR DIFFERENT YARA-RULE FORMAT

YARA-rule format	YARA-rule ID	$\lvert\mathcal{G}\rvert$	$\mathcal{CPU}$ [%]
Υ_{INIT}	2	6	15.7
	3	3	31.5
Υ_{PERT}	3	481	23.5
	6	120	24.0
$\Upsilon_{LD_{OPT}}$	2	37424	18.6
	3	6625	23.2
$\Upsilon_{LCS_{OPT}}$	2	13584	19.0
	3	2756	25.1
Multi-step	2	13917	19.4
	3	3237	25.9
$\Upsilon_{FSM\_TO\_RX_{OPT}}$	3	481	25.3

TABLE X

$|\mathcal{G}|$ IMPACT ON $\mathcal{R}$ FOR DIFFERENT YARA-RULE FORMAT

YARA-rule format	YARA-rule ID	$\lvert\mathcal{G}\rvert$	$\mathcal{R}$ [%]
Υ_{INIT}	2	6	28.5
	3	3	30.3
Υ_{PERT}	3	481	31.1
$\Upsilon_{LD_{OPT}}$	2	37424	27.6
$\Upsilon_{LCS_{OPT}}$	2	13584	26.4
	3	2756	31.6
Multi-step	2	13917	29.2
	3	3237	31.8
$\Upsilon_{FSM\_TO\_RX_{OPT}}$	3	481	25.4
	2	333	27.0

10 cases (4 each). On the other hand, Multi-step scanning achieves more frequently (in 40% of test cases) the highest CPU consumption, resulting in poorer performance. The analysis in Table IX focuses on $|\mathcal{G}|$ impact on $\mathcal{CPU}$ per different rule format. In particular, it shows YARA-rules achieving maximum $|\mathcal{G}|$ and $\mathcal{CPU}$ values, respectively. Ah shown in Table IX, only in one case, that is $\Upsilon_{FSM\_TO\_RX_{OPT}}$, the highest $|\mathcal{G}|$ value corresponds to the highest average percentage of CPU consumption. Furthermore, as shown by Figure 4a the third Υ_{INIT} results in the highest $\mathcal{CPU}$ among all compared approaches. This rule reaches such a value without YAMME employment and for the lowest $|\mathcal{G}|$ value from those listed in Table IX.

Figure 4b illustrates the average percentage of RAM usage with respect to the overall scanning process, using different format of YARA-rules and considering different $|\mathcal{G}|$ values. Fixing the ID of the YARA-rule, in 8 out of 10 cases, $\Upsilon_{FSM\_TO\_RX_{OPT}}$ requires a less average memory consumption. In the remaining cases, the best results are obtained by Υ_{PERT} and $\Upsilon_{LCS_{OPT}}$, respectively. Therefore, increasing $|\mathcal{G}|$ does not affect $\mathcal{R}$ in the case of a single YARA-rule. In contrast, multi-step scanning results in the worst performance, i.e., in the highest memory RAM consumption in all tested cases. This is expected, since a multi-step process needs more memory consumption, as the scan result depends on the occurrence of at least two conditions. Therefore, the first one remains in memory until the second occurs. Table X reports the rules having the maximum $|\mathcal{G}|$ value and achieving the maximum $\mathcal{R}$, per rule format. As shown in Table X, only for Υ_{PERT} and $\Upsilon_{LD_{OPT}}$ the highest $|\mathcal{G}|$ value corresponds to the highest $\mathcal{R}$. In all other cases, $\mathcal{R}$ is not affected by $|\mathcal{G}|$, nether by rule structure, i.e., a rule composed of simple HEX strings or regular expressions. Furthermore, YARA-rule ID 3 results in the maximum $\mathcal{R}$ value in 4 out of 6 different rule formats. However, both the second and ninth YARA-rules have a higher $|\mathcal{G}|$ than the third rule as shown in Figure 4b. Therefore, $|\mathcal{G}|$ is not the unique parameter that affects RAM consumption during the scanning process.

Figure 4c shows the disk space needed (in Byte) to store different formats of YARA-rules, each having a different $|\mathcal{G}|$ value. In this test, $\mathcal{D}$ is, of course, proportional to $|\mathcal{G}|$ if strings different from regular expressions are used. Among the optimization algorithms introduced, the $\Upsilon_{LD_{OPT}}$ rule format is the most advantageous, although it leads to an exponential growth of $|\mathcal{G}|$. On the contrary, Υ_{PERT} and Multi-step scanning result in the highest $\mathcal{D}$. In general, to obtain a less $\mathcal{D}$ consumption, approaches that reduce the number of characters within the generic rule are preferred.

In Figure 4d, the scan time (in seconds) required by the different YARA-rules compared is highlighted. In 40% of the YARA-rules examined, Υ_{INIT} achieve the shortest scan time. Thus, the rule enhancement performed by YAMME leads to better timing performance in 6 out of 10 YARA-rules. In particular, $\Upsilon_{LCS_{OPT}}$ and $\Upsilon_{FSM\_TO\_RX_{OPT}}$ result in the best timing performances for the same number of times. This is an interesting insight, which confirms how the YARA scanner is able to discard every string not representing a potential match, by selecting the appropriate atoms list to preserve scanning speed. As a consequence, $|\mathcal{G}|$ growth and rule format do not

TABLE XI
$|\mathcal{G}|$ IMPACT ON $\mathcal{S}$ FOR DIFFERENT YARA-RULE FORMAT

YARA-rule format	YARA-rule ID	$\lvert\mathcal{G}\rvert$	$\mathcal{S}$ [s]
Υ_{INIT}	2	6	227
	7	6	631
Υ_{PERT}	3	481	661
	9	70	665
$\Upsilon_{LD_{OPT}}$	2	37424	348
	7	86	707
$\Upsilon_{LCS_{OPT}}$	2	13584	290
	7	38	676
Multi-step	2	13917	369
	7	54	712
$\Upsilon_{FSM\_TO\_RX_{OPT}}$	3	481	365
	7	16	664

TABLE XII
MULTIPLE RULES SCANNING COMPUTATIONAL PERFORMANCES

YARA-rule format	$\lvert\mathcal{G}\rvert$	$\mathcal{CPU}$ [%]	$\mathcal{R}$ [%]	$\mathcal{S}$ [s]
Υ_{INIT}	23	22.3	19.3	237
Υ_{PERT}	1063	16.9	21.3	497
$\Upsilon_{LD_{OPT}}$	61434	22.5	19.5	627
$\Upsilon_{LCS_{OPT}}$	24172	22.3	20.7	563
Multi-step	25228	20.8	21.2	653
$\Upsilon_{FSM\_TO\_RX_{OPT}}$	1063	22.8	21.2	588

affect $\mathcal{S}$. To better show this result, Table XI is examined. For each rule format, the pair of rules resulting in maximum $|\mathcal{G}|$ and $\mathcal{S}$ values, respectively, has been selected. Given the data from Table XI, it is observed that in each case the maximum value of $\mathcal{S}$ is not obtained for the maximum value of $|\mathcal{G}|$. Furthermore, very different scan times can be obtained for the same $|\mathcal{G}|$ value. Therefore, the explosion of the number of HEX strings introduced by YAMME does not affect the scanning process in terms of the time required. In this regard, it can be seen that the scan time required by an example of Υ_{INIT} with 6 strings turns out to be almost twice that required by $\Upsilon_{LD_{OPT}}$ with 37424 strings. Moreover, Figure 4d shows that the Multi-step under-performs in 90% of the test cases. This can be attributed to the condition, which must be verified between a couple of YARA-rules.

Finally, Table XII shows the computational performances required when multiple rules are involved during the scanning process. Since the group of rules in Table IV is used, $\mathcal{D}$ is approximately the sum of the values in single-rule experiments. Therefore, this analysis focuses on the remaining metrics dependency by $|\mathcal{G}|$. In particular, it was found that when a scanning process uses multiple rules, there is no performance degradation compared to the case of single rules. In fact, for every test case, there is at least one case where single rules in the same format result in worse performance.

VII. TECHNIQUE FEASIBILITY

Detection of obfuscated malware is a complex problem for static analysis tools such as YARA since these can evade such defense mechanisms by transforming the patterns searched by AV engines. According to [18], the anti-static analysis means used by malware authors can be categorized as follows: (i) information hiding, i.e., a method used to hide ASCII strings, variables, debugging information, import table and other information; (ii) exploitation of the system exception handling performed in Windows-OS-based machine to interfere with the work of AV software and disassembler; (iii) packaging obfuscations that aim at compressing/encrypting an executable sample by increasing its structural entropy; (iv) traditional code obfuscation methods employed by metamorphic mutation engines to generate malware variants. In the fourth case, a malware author can rewrite the ASM code of a malicious sample without altering its malicious scope, i.e., creating a metamorphic variant that bypasses static analysis tools, as shown in Figure 1. Metamorphic malware variants are created using sophisticated and ad-hoc mutation engines developed by malware writers. However, each mutation engine implements common functions according to its architecture [29]: (i) disassembles the source code into an assembly one; (ii) implements code shrinking, i.e., eliminates the effect of GCI at the previous iteration; (iii) swaps instructions using the IP technique; (iv) replaces the instruction with equivalent ones (IR); (v) performs RE and GCI; (vi) implements code transportation and performs inlining and outlining processes; (vii) finally, re-assemble the code into the source-code of the new viral sample.

YAMME can exactly replicate the same actions performed by a metamorphic mutation engine at the YARA-byte-signatures level. Given a YARA-rule, YAMME rewrites its byte-signatures applying the same techniques used to make it unable to detect the generic malware variant. Therefore, given the YARA-rules HEX strings that represent an ASM OPCODE sequence (i.e., a very peculiar malicious action), these are generalized by YAMME computing equivalent OPCODE sequences, that is, the same of those will be obtained applying a malware obfuscation transformation resulting in a known OPCODE sequence mutation. Therefore, the proposed mechanism overcomes these weaknesses exposed by the YARA-rules syntax, which make such rules ineffective in detecting metamorphic malware. However, YARA-rules are widely used by commercial AV and IDPS tools, and very popular open-source engines. Hence, this YARA-rule syntax issue can affect a very extended set of real-world cybersecurity tools.

VIII. CONCLUSION

This paper presented YAMME, a post-processing mechanism usable to strengthen YARA-rules against metamorphic malware capable of evading YARA-byte-signatures by employing general obfuscation techniques. Based on the results obtained, it was found that YAMME is effective in improving YARA-rules detection rate against metamorphic malware. Therefore, YAMME rules can intercept variants of metamorphic malware taken from the real world. Furthermore, the performance of scanning processes using YAMME rules is not affected by the growth in the number of variants covered by the rule, as shown by the computational overhead test results. Possible future works will regard the investigation on additional malware obfuscation methods as well as the usage of Artificial Intelligence-based paradigms, such as, Reinforcement Learning, to expand the perturbation

YAMME capabilities. Furthermore, several studies on additional optimization methods to apply on YAMME-rules will be conducted aimed at further reducing the computational overhead, ensuring detection performances are maintained.

ACKNOWLEDGMENT

Author Contributions: conceptualization: Antonio Coscia and Antonio Maci; data curation and software: Antonio Maci; investigation, methodology and formal analysis: Vincenzo Dentamaro, Stefano Galantucci, and Antonio Maci; writing—original draft preparation and writing—review and editing: Stefano Galantucci and Antonio Maci; supervision and validation: Antonio Coscia and Giuseppe Pirlo; project administration and funding acquisition: Giuseppe Pirlo.

REFERENCES

[1] F. A. Aboaoja, A. Zainal, F. A. Ghaleb, B. A. S. Al-Rimy, T. A. E. Eisa, and A. A. H. Elnour, "Malware detection issues, challenges, and future directions: A survey," *Appl. Sci.*, vol. 12, no. 17, p. 8482, Aug. 2022.

[2] H. Oz, A. Aris, A. Levi, and A. S. Uluagac, "A survey on ransomware: Evolution, taxonomy, and defense solutions," *ACM Comput. Surveys*, vol. 54, no. 11s, pp. 1–37, Jan. 2022.

[3] Z. Huang, Q. Wang, Y. Chen, and X. Jiang, "A survey on machine learning against hardware Trojan attacks: Recent advances and challenges," *IEEE Access*, vol. 8, pp. 10796–10826, 2020.

[4] R. Ball, "Computer viruses, computer worms, and the self-replication of programs," in *Viruses in all Dimensions: How an Information Code Controls Viruses, Software and Microorganisms*. Cham, Switzerland: Springer, 2023, pp. 73–85.

[5] D. Farhat and M. S. Awan, "A brief survey on ransomware with the perspective of internet security threat reports," in *Proc. 9th Int. Symp. Digit. Forensics Secur. (ISDFS)*, Jun. 2021, pp. 1–6.

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

[7] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—A state of the art survey," *ACM Comput. Surveys*, vol. 52, no. 5, pp. 1–48, Sep. 2020.

[8] J. Liang and Y. Kim, "Evolution of firewalls: Toward securer network using next generation firewall," in *Proc. IEEE 12th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jun. 2022, pp. 0752–0759.

[9] Virustotal. (2019). *Yara—The Pattern Matching Swiss Knife For Malware Researchers (and Everyone Else)*. [Online]. Available: https://virustotal.github.io/yara/

[10] M. Botacin et al., "Antiviruses under the microscope: A hands-on perspective," *Comput. Secur.*, vol. 112, Jan. 2022, Art. no. 102500.

[11] A. Zhdanov, "Generation of static YARA-signatures using genetic algorithm," in *Proc. IEEE Eur. Symp. Secur. Privacy Workshops (EuroSPW)*, Jun. 2019, pp. 220–228.

[12] D. P. F. Bilstein, "YARA-signator: Automated generation of code-based YARA rules," *J. Cybercrime Digit. Invest.*, vol. 5, no. 1, pp. 1–13, 2019.

[13] E. Raff et al., "Automatic YARA rule generation using biclustering," in *Proc. 13th ACM Workshop Artif. Intell. Secur.*, Nov. 2020, pp. 71–82.

[14] M. Khalid, M. Ismail, M. Hussain, and M. Hanif Durad, "Automatic YARA rule generation," in *Proc. Int. Conf. Cyber Warfare Secur. (ICCWS)*, Oct. 2020, pp. 1–5.

[15] Q. Si et al., "Malware detection using automated generation of YARA rules on dynamic features," in *Proc. Sci. Cyber Secur., 4th Int. Conf., SciSec*. Matsue, Japan: Springer, Aug. 2022, pp. 315–330.

[16] M. Brengel and C. Rossow, "YARIX: Scalable YARA-based malware intelligence," in *Proc. USENIX Secur. Symp.*, 2021, pp. 3541–3558.

[17] L. E. S. Jaramillo, "Malware detection and mitigation techniques: Lessons learned from Mirai DDOS attack," *J. Inf. Syst. Eng. Manage.*, vol. 3, no. 3, p. 19, Jul. 2018.

[18] Y. Gao, Z. Lu, and Y. Luo, "Survey on malware anti-analysis," in *Proc. 5th Int. Conf. Intell. Control Inf. Process.*, Aug. 2014, pp. 270–275.

[19] M. Medhat, S. Gaber, and N. Abdelbaki, "A new static-based framework for ransomware detection," in *Proc. IEEE 16th Intl Conf Dependable, Autonomic Secure Comput., 16th Intl Conf Pervasive Intell. Comput., 4th Intl Conf Big Data Intell. Comput. Cyber Sci. Technol. Congress(DASC/PiCom/DataCom/CyberSciTech)*, Aug. 2018, pp. 710–715.

[20] M. S. Kumar, J. Ben-Othman, and K. G. Srinivasagan, "An investigation on Wannacry ransomware and its detection," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2018, pp. 1–6.

[21] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 421–430.

[22] M. Ficco, "Malware analysis by combining multiple detectors and observation windows," *IEEE Trans. Comput.*, vol. 71, no. 6, pp. 1276–1290, Jun. 2022.

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

[24] (2020). *Repository of YARA Rules*. Accessed: May 1, 2023. [Online]. Available: https://github.com/Yara-Rules/rules

[25] C Forensics. *Virusshare.com*. Accessed: May 1, 2023. [Online]. Available: https://virusshare.com/

[26] B. Bashari Rad and M. Masrom, "Metamorphic virus detection in portable executables using opcodes statistical feature," 2011, *arXiv:1104.3229*.

[27] N. Naik, P. Jenkins, R. Cooke, J. Gillett, and Y. Jin, "Evaluating automatically generated YARA rules and enhancing their effectiveness," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Dec. 2020, pp. 1146–1153.

[28] M. Hayes, A. Walenstein, and A. Lakhotia, "Evaluation of malware phylogeny modelling systems using automated variant generation," *J. Comput. Virology*, vol. 5, no. 4, pp. 335–343, Nov. 2009.

[29] K. Brezinski and K. Ferens, "Metamorphic malware and obfuscation—A survey of techniques, variants and generation kits," *Secur. Commun. Netw.*, Oct. 2021, doi: 10.13140/RG.2.2.19702.52802.

[30] 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.

[31] 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.

[32] R. Murali, P. Thangavel, and C. S. Velayutham, "Evolving malware variants as antigens for antivirus systems," *Exp. Syst. Appl.*, vol. 226, Sep. 2023, Art. no. 120092.

[33] S. Noreen, S. Murtaza, M. Z. Shafiq, and M. Farooq, "Evolvable malware," in *Proc. 11th Annu. Conf. Genetic Evol. Comput.*, Jul. 2009, pp. 1569–1576.

[34] J. Choi, D. Shin, H. Kim, J. Seotis, and J. B. Hong, "AMVG: Adaptive malware variant generation framework using machine learning," in *Proc. IEEE 24th Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Dec. 2019, pp. 246–24609.

[35] M. Sewak, S. K. Sahay, and H. Rathore, "X-swarm: Adversarial DRL for metamorphic malware swarm generation," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops Affiliated Events (PerCom Workshops)*, Mar. 2022, pp. 169–174.

[36] M. Sewak, S. K. Sahay, and H. Rathore, "ADVERSARIALuscator: An adversarial-DRL based obfuscator and metamorphic malware swarm generator," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2021, pp. 1–9.

[37] M. Sewak, S. K. Sahay, and H. Rathore, "DOOM: A novel adversarial-DRL-based op-code level metamorphic malware obfuscator for the enhancement of IDS," in *Proc. Adjunct ACM Int. Joint Conf. Pervasive Ubiquitous Comput. ACM Int. Symp. Wearable Comput.*, Sep. 2020, pp. 131–134.

[38] M. Sewak, S. K. Sahay, and H. Rathore, "DRLDO: A novel DRL based de-ObfuscationSystem for defense against metamorphic malware," 2021, *arXiv:2102.00898*.

[39] T. Quertier, B. Marais, S. Morucci, and B. Fournel, "MERLIN–malware evasion with reinforcement learning," 2022, *arXiv:2203.12980*.

[40] S. Madenur Sridhara and M. Stamp, "Metamorphic worm that carries its own morphing engine," *J. Comput. Virol. Hacking Techn.*, vol. 9, no. 2, pp. 49–58, May 2013.

[41] B. Jin, J. Choi, J. B. Hong, and H. Kim, "On the effectiveness of perturbations in generating evasive malware variants," *IEEE Access*, vol. 11, pp. 31062–31074, 2023.

[42] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *J. Comput. Virology*, vol. 1, nos. 1–2, pp. 13–23, Nov. 2005.

[43] Q. Zhang and D. S. Reeves, "MetaAware: Identifying metamorphic malware," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 411–420.

[44] V. P. Nair, H. Jain, Y. K. Golecha, M. S. Gaur, and V. Laxmi, "MEDUSA: Metamorphic malware dynamic analysis using signature from API," in *Proc. 3rd Int. Conf. Secur. Inf. Netw.*, Sep. 2010, pp. 263–269.

[45] D. Baysa, R. M. Low, and M. Stamp, "Structural entropy and metamorphic malware," *J. Comput. Virol. Hacking Techn.*, vol. 9, no. 4, pp. 179–192, Nov. 2013.

[46] I. Sorokin, "Comparing files using structural entropy," *J. Comput. Virology*, vol. 7, no. 4, pp. 259–265, Nov. 2011.

[47] A. A. E. Elhadi, M. A. Maarof, B. I. A. Barry, and H. Hamza, "Enhancing the detection of metamorphic malware using call graphs," *Comput. Secur.*, vol. 46, pp. 62–78, Oct. 2014.

[48] P. Vinod, V. Laxmi, M. S. Gaur, and G. Chauhan, "MOMENTUM: MetamOrphic malware exploration techniques using MSA signatures," in *Proc. Int. Conf. Innov. Inf. Technol. (IIT)*, Mar. 2012, pp. 232–237.

[49] G. Canfora, A. N. Iannaccone, and C. A. Visaggio, "Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics," *J. Comput. Virol. Hacking Techn.*, vol. 10, no. 1, pp. 11–27, Feb. 2014.

[50] E. Radkani, S. Hashemi, A. Keshavarz-Haddad, and M. A. Haeri, "An entropy-based distance measure for analyzing and detecting metamorphic malware," *Int. J. Speech Technol.*, vol. 48, no. 6, pp. 1536–1546, Jun. 2018.

[51] S. K. Sasidharan and C. Thomas, "A survey on metamorphic malware detection based on hidden Markov model," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2018, pp. 357–362.

[52] R. Mirzazadeh, M. H. Moattar, and M. V. Jahan, "Metamorphic malware detection using linear discriminant analysis and graph similarity," in *Proc. 5th Int. Conf. Comput. Knowl. Eng. (ICCKE)*, Oct. 2015, pp. 61–66.

[53] S. Alam, R. N. Horspool, and I. Traore, "MARD: A framework for metamorphic malware analysis and real-time detection," in *Proc. IEEE 28th Int. Conf. Adv. Inf. Netw. Appl.*, May 2014, pp. 480–489.

[54] S. Alam, R. N. Horspool, I. Traore, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," *Comput. Secur.*, vol. 48, pp. 212–233, Feb. 2015.

[55] S. Alam, R. N. Horspool, and I. Traore, "MAIL: Malware analysis intermediate language: A step towards automating and optimizing malware detection," in *Proc. 6th Int. Conf. Secur. Inf. Netw.*, Nov. 2013, pp. 233–240.

[56] Y. T. Ling, N. F. M. Sani, M. T. Abdullah, and N. A. W. A. Hamid, "Nonnegative matrix factorization and metamorphic malware detection," *J. Comput. Virol. Hacking Techn.*, vol. 15, no. 3, pp. 195–208, Sep. 2019.

[57] Y. T. Ling et al., "Metamorphic malware detection using structural features and nonnegative matrix factorization with hidden Markov model," *J. Comput. Virol. Hacking Techn.*, vol. 18, pp. 183–203, Oct. 2021.

[58] L. Wang, D. Xu, J. Ming, Y. Fu, and D. Wu, "MetaHunt: Towards taming malware mutation via studying the evolution of metamorphic virus," in *Proc. 3rd ACM Workshop Softw. Protection*, 2019, pp. 15–26.

[59] F. Biondi, F. Dechelle, and A. Legay, "MASSE: Modular automated syntactic signature extraction," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2017, pp. 96–97.

[60] P. Black, I. Gondal, A. Bagirov, and M. Moniruzzaman, "Malware variant identification using incremental clustering," *Electronics*, vol. 10, no. 14, p. 1628, Jul. 2021.

[61] M. Belaoued, A. Boukellal, M. A. Koalal, A. Derhab, S. Mazouzi, and F. A. Khan, "Combined dynamic multi-feature and rule-based behavior for accurate malware detection," *Int. J. Distrib. Sensor Netw.*, vol. 15, no. 11, 2019, Art. no. 1550147719889907.

[62] N. Naik, P. Jenkins, N. Savage, L. Yang, K. Naik, and J. Song, "Augmented YARA rules fused with fuzzy hashing in ransomware triaging," in *Proc. IEEE Symp. Series Comput. Intell. (SSCI)*, Dec. 2019, pp. 625–632.

[63] N. Naik, P. Jenkins, N. Savage, L. Yang, K. Naik, and J. Song, "Embedding fuzzy rules with YARA rules for performance optimisation of malware analysis," in *Proc. IEEE Int. Conf. Fuzzy Syst. (FUZZ-IEEE)*, Jul. 2020, pp. 1–7.

[64] N. Naik et al., "Fuzzy hashing aided enhanced YARA rules for malware triaging," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Dec. 2020, pp. 1138–1145.

[65] N. Naik et al., "Embedded YARA rules: Strengthening YARA rules utilising fuzzy hashing and fuzzy rules for malware analysis," *Complex Intell. Syst.*, vol. 7, no. 2, pp. 687–702, Apr. 2021.

[66] L. Xu and M. Qiao, "YARA rule enhancement using Bert-based strings language model," in *Proc. 5th Int. Conf. Adv. Electron. Mater., Comput. Softw. Eng. (AEMCSE)*, Apr. 2022, pp. 221–224.

[67] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proc. IEEE Symp. Secur. Privacy (SP)*, Jun. 2006, p. 15.

[68] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting signature learning by training maliciously," in *Recent Advances in Intrusion Detection*. Hamburg, Germany: Springer, Sep. 2006, pp. 81–105.

[69] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Functionality-preserving black-box optimization of adversarial windows malware," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 3469–3478, 2021.

[70] A. Mohanta and A. Saldanha, *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*. Cham, Switzerland: Springer, 2020.

[71] K. Kaushal, P. Swadas, and N. Prajapati, "Metamorphic malware detection using statistical analysis," *Int. J. Soft Comput. Eng.*, vol. 2, no. 3, pp. 49–53, 2012.

[72] G. Jäger and J. Rogers, "Formal language theory: Refining the Chomsky hierarchy," *Phil. Trans. Roy. Soc. B, Biol. Sci.*, vol. 367, no. 1598, pp. 1956–1970, Jul. 2012.

[73] D. E. Knuth and J. L. Szwarcfiter, "A structured program to generate all topological sorting arrangements," *Inf. Process. Lett.*, vol. 2, no. 6, pp. 153–157, Apr. 1974.

[74] Cisco-Talos. (2003). *Clamav*. [Online]. Available: https://github.com/Cisco-Talos/clamav

[75] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proc. 7th Int. Symp. String Process. Inf. Retrieval.*, 2000, pp. 39–48.

[76] J. Wang and Y. Dong, "Measurement of text similarity: A survey," *Information*, vol. 11, no. 9, p. 421, Aug. 2020.

[77] A. Ortega. (2019). *Metame—A Metamorphic Code Engine for Arbitrary Executables*. [Online]. Available: https://github.com/a0rtega/metame

[78] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Secur. Appl. Conf. (ACSAC)*. IEEE, 2006, pp. 339–348.

[79] *The Portable Freeware Collection*. Accessed: May 1, 2023. [Online]. Available: https://www.portablefreeware.com/

[80] H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," 2018, *arXiv:1804.04637*.

[81] D. Regéciová, D. Kolár, and M. Milkovic, "Pattern matching in YARA: Improved Aho-Corasick algorithm," *IEEE Access*, vol. 9, pp. 62857–62866, 2021.

[82] Avast-Software. (2020). *Yaramod—Parsing of YARA Rules Into AST and Building New Rulesets*. [Online]. Available: https://yaramod.readthedocs.io/en/latest/parsing_rulesets.html

[83] W. C. Nguyen and A. Quynh. (2019). *Capstone—The Ultimate Disassembler*. [Online]. Available: https://www.capstone-engine.org/lang_python.html

[84] W. C. Nguyen, A. Quynh, and S. Schirra. (2020). *Keystone—The Ultimate Assembler*. [Online]. Available: https://www.keystone-engine.org/docs/tutorial.html

Open Access funding provided by 'Università degli Studi di Bari "Aldo Moro"' within the CRUI CARE Agreement