Journal of Computer Virology and Hacking Techniques (2021) 17:167–183
https://doi.org/10.1007/s11416-021-00377-z

INVITED PAPER

Learning metamorphic malware signatures from samples

Marco Campion[1] · **Mila Dalla Preda**[1] · **Roberto Giacobazzi**[1]

Received: 1 September 2020 / Accepted: 6 January 2021 / Published online: 19 February 2021

Abstract
Metamorphic malware are self-modifying programs which apply semantic preserving transformations to their own code in order to foil detection systems based on signature matching. Metamorphism impacts both software security and code protection technologies: it is used by malware writers to evade detection systems based on pattern matching and by software developers for preventing malicious host attacks through software diversification. In this paper, we consider the problem of automatically extracting metamorphic signatures from the analysis of metamorphic malware variants. We define a metamorphic signature as an abstract program representation that ideally captures all the possible code variants that might be generated during the execution of a metamorphic program. For this purpose, we developed *MetaSign*: a tool that takes as input a collection of metamorphic code variants and produces, as output, a set of transformation rules that could have been used to generate the considered metamorphic variants. *MetaSign* starts from a control flow graph representation of the input variants and agglomerates them into an automaton which approximates the considered code variants. The upper approximation process is based on the concept of widening automata, while the semantic preserving transformation rules, used by the metamorphic program, can be viewed as rewriting rules and modeled as grammar productions. In this setting, the grammar recognizes the language of code variants, while the production rules model the metamorphic transformations. In particular, we formalize the language of code variants in terms of pure context-free grammars, which are similar to context-free grammars with no terminal symbols. After the widening process, we create a positive set of samples from which we extract the productions of the grammar by applying a learning grammar technique. This allows us to learn the transformation rules used by the metamorphic engine to generate the considered code variants. We validate the results of *MetaSign* on some case studies.

Keywords Metamorphic malware · Malware signature · Widening automata · Pure context-free grammars · Learning grammars

1 Introduction

Detecting and neutralizing computer malware, such as worms, viruses, trojans and spyware, is a major challenge in modern computer security, involving both sophisticated intrusion detection strategies and advanced code manipulation tools and methods. Despite the relentless effort of researches in developing sophisticated malware detection tools, the number of malware is growing exponentially together with their complexity [1]. According to *Symantec's 2019 Internet Threat Report*, the number of new malware samples had an increase of 22.9% in 2017, and the number of destructive malware had an increase of 25% in 2018 [1]. Interestingly, the 63% of malware comes from already existing or known attacks, as reported in *Ponemon Institute's 2018 State of Endpoint Security Risk* [2]. The *Ponemon Institute's* reports that 76% of organizations are using signatures-based antimalware systems to protect themselves. These antimalware tools analyze known malware samples in order to extract their signatures and collect them in a database. Pattern matching is then used to compare the byte sequence comprising the body of the malware against the signature

The research has been partially supported by the Grant PRIN2017 (code: 201784YSZ5) and the project "Dipartimenti di Eccellenza 2018-2022" funded by the Italian Ministry of Education, Universities and Research (MIUR).

✉ Marco Campion
marco.campion@univr.it

Mila Dalla Preda
mila.dallapreda@univr.it

Roberto Giacobazzi
roberto.giacobazzi@univr.it

[1] Università degli Studi di Verona, Verona, Italy

database [3]. This process takes a long time and the malware samples may leave undetected in this period. Moreover, the efficiency of signature-based detection systems heavily depends on the completeness of the database of malware signatures used (which may be different among different anti-malware tools).

Malware writers have responded by using a variety of camouflage techniques in order to avoid signature-based detection. *Encryption* [4] is one of the simplest methods employed by malware writers to avoid detection. It is based on two main sections: the main body, also known as the payload, and a decryption loop which is responsible of the encryption and decryption of the payload. *Oligomorphism* [5] is an advanced form of encryption: it contains a collection of different decryption routines that are randomly chosen for every new infection. This ensures that the decryption code varies among the different malware instances. *Polymorphism* [6] is actually the most complicated type of oligomorphism and encryption. The difference stems in the unlimited number of encryption methods that allow us to generate an endless sequence of decryption patterns.

Metamorphism [4] emerged in the last decade as an effective alternative strategy to foil signature-based malware detectors. Unlike the previous camouflage techniques, metamorphic malware have no encrypted code (no need of decryption), but like polymorphic malware they employ a mutation engine that, instead of modifying the decryption routine, mutates the code of the whole malware. Thus, *Metamorphic malware* [4] are self-modifying programs which iteratively apply code transformation rules that preserve the semantics of programs. These code transformations change the syntax of code in order to foil detection systems based on signature matching. These programs are equipped with a metamorphic engine that usually represents the 90% of the whole program code [4]. This engine takes as input the malware own code and it produces, at run time, a syntactically different but semantically equivalent program. One of the first known metamorphic virus produced for DOS was *ACG*, in 1998 [3], and the first effort on 32-bits metamorphic virus targeting the Portable Executable files was *W32.Appartition* that spread in 2000 [3]. The anatomy of a metamorphic malware includes:

1. Disassembler;
2. Code transformer;
3. Assembler.

When the malware finds the location of its own code, it uses an internal disassembler to convert the code into assembly instructions. The heart of the mutation engine is a code transformer, also called obfuscator, that is responsible of changing the binary sequence of malware code by applying semantic preserving rewriting rules. The last module, the Assembler, converts the mutated assembly code produced by the mutation engine, into machine binary code.

We call *metamorphic variants* the program variants generated by the mutation engine. At the assembly level, these semantic preserving transformations used by the metamorphic engine include: semantic-nop/junk insertion, code permutation, register swap and substitution of equivalent sequences of instructions [7] (see Fig. 1 for an example).

The large amount of possible metamorphic code variants makes it impractical to maintain a signature set that is large enough to cover most or all of these variants, thus making standard signature-based detection ineffective. Conversely, heuristic techniques may be prone to false positives or false negatives. The key to identify these type of malicious programs consists in considering semantic program features and not purely syntactic program features, thus capturing code mutations while preserving the semantic intent [9]. For this reason, we would like to capture those semantic aspects that allow us to detect all the possible variants that

Original code

```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through dead-code insertion

```
call 0h
pop ebx
lea ecx, [ebx+42h]
nop                 (*)
nop                 (*)
push ecx
push eax
inc eax             (**)
push eax
dec [esp - 0h]      (**)
dec eax             (**)
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through code transposition

```
        call 0h
        pop ebx
        jmp S2
S3:     push eax
        push eax
        sidt [esp - 02h]
        jmp S4
        add ebx, 1Ch
        jmp S6
S2:     lea ecx, [ebx+42h ]
        push ecx
        jmp S3
S4:     pop ebx
        cli
        jmp S5
S5:     mov ebp, [ebx]
```

Code obfuscated through instruction substitution

```
call 0h
pop ebx
lea ecx, [ebx+42h]
sub esp, 03h
sidt [esp - 02h]
add [esp], 1Ch
mov ebx, [esp]
inc esp
cli
mov ebp, [ebx]
```

Fig. 1 Examples of semantic preserving transformations applied to a fragment of *Chernobyl/CHI* malware, taken from [8]. Newly added instructions are highlighted

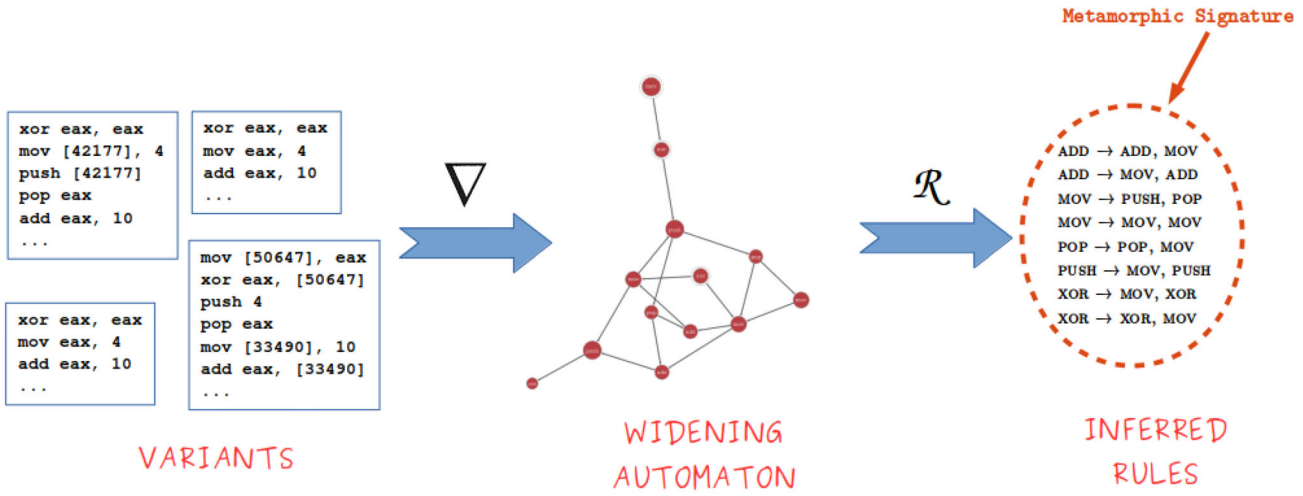

Fig. 2 Capturing the metamorphic signature

can be generated by the metamorphic engine. We use the term *metamorphic signature* to refer to an abstract program representation that ideally captures all the possible code variants that might be generated during the execution of a metamorphic program. A metamorphic signature is therefore any (possibly decidable) approximation of the properties of code evolution.

The goal of this work is to statically extract an approximated version of the so called metamorphic signature, i.e., a signature of the metamorphic engine itself. In this setting, a metamorphic signature consists of a set of rewriting rules that the malware can use to change its code. These rules can be represented as a pure context-free grammar [10], having instructions as terminal symbols, and can be transformed into equivalent instructions following the grammar productions. For this purpose, we have built a tool, called *MetaSign*, that takes as input simplified/abstract versions of the metamorphic code variants, and it represents them as Control Flow Graphs (CFG). The resulting representation is isomorphic to Finite State Automata (FSA) over an alphabet of instructions [11]. For this reason, from now on, we use the terms CFG and automaton interchangeably. Starting from CFGs representation of the variants, *MetaSign* embeds them in an over-approximating CFG through a specific *widening* operator applied to automata [11]. The key idea is to build a unique CFG for all the code variants that over-approximates the union of their recognized languages, i.e., it models the metamorphic behavior as a regular language of abstract instructions. This process is called *regular metamorphism* [12]. Finally, from the resulting widening CFG, *MetaSign* tries to *learn* the rewriting rules used to generate each variant through a simple learning grammar technique [13,14] and it produces, as output, the inferred rules. The general structure of the tool is represented in Fig. 2.

In order to validate the quality of the tool output, a metamorphic engine is implemented in *MetaSign*, allowing us to generate a collection of metamorphic code variants from a given program. Our metamorphic engine takes as input a program written in an intermediate language similar to the `x86` assembly [15] and it randomly chooses the rewriting rules to apply in order to generate the metamorphic variants. The metamorphic rules implemented are a subset of those used by the metamorphic malware *MetaPHOR* [4]. This metamorphic engine allows us to quickly generate numerous test sets, feed (part of) them as inputs to *MetaSign* and check the quality of the results by comparing the rules inferred by *MetaSign* with those actually applied by the metamorphic engine.

The rest of this paper is organized as follows: in Sect. 2 we discuss some related works, Sect. 3 presents background concepts used in the rest of the paper, Sect. 4 explains our approach towards capturing metamorphic signatures, Sect. 5 describes the implementation details of *MetaSign* , in Sect. 6 we present some results and consideration applied to three case studies, in Sect. 7 we discuss benefit, drawbacks and future directions of our approach.

2 Related work

Behavioural malware detectors are based on an abstract behavioural model of malware whose design is driven by the a priori knowledge of the obfuscating transformations typically used by malware writers to evade detection. So, for example, variable renaming is typically handled by using symbolic names for variables. Formal methods have been extensively used for specifying the malicious behaviour of malware while abstracting from implementation details usually changed by obfuscation. The following papers are some related works based on formal methods that have inspired

our work on metamorphic signatures extraction. In [16] the authors propose a malware detection scheme based on the identification of suspicious sequences of system calls. In particular, they consider a subgraph of the program CFG, which contains only the nodes that represent certain system calls and, finally, they check if this subgraph contains some known malicious system call sequences. In [17] the authors describe a malware detection system based on language containment and unification. The malicious code and the possible infected program are modeled as automata with unresolved symbols and placeholders for registers dealing with certain types of obfuscation. In this configuration, a program exhibits malicious behaviour if the intersection between the malware's automaton language and the program's automaton language is not empty. In [18] the authors specify malicious behaviour through a Linear Temporal Logic (LTL) formula and then use the SPIN model checker to check if this property is satisfied by the CFG of a suspicious program. In [19] the authors introduce a new Computation Tree Predicate Logic (CTPL) temporal logic, which is an extension of the logic CTL, which takes into account the quantification of the registers, allowing a natural presentation of malicious patterns. In [20] the authors describe the malicious behaviour through a template that is a generalization of the malicious code that aims at expressing the malicious intent while excluding the details of the implementation. The idea is that the template should not distinguish between irrelevant variants of the same malware obtained through obfuscation processes. For example, the proposed template uses symbolic variables/constants to handle the renaming of variables and registers, and considers the malware CFG in order to handle code reordering. Finally, they provide an algorithm to verify if a program presents the behaviour expressed by the template. This algorithm is based on the unification between the program variables/constants and the symbolic variables/constants of the malware. All the approaches presented so far use formal methods to build a generalized semantic signature that ideally matches all the malware variants. The generalization of the signature is guided by the knowledge of the transformations typically used by malware to avoid detection. Whereas our approach aims at extracting the generalized signature from the analysis of a specific metamorphic malware. Indeed, the goal of our detection strategy is to learn the code transformation rules applied by the metamorphic engine embedded in the malware under analysis, and then to use these rules to recognize any possible malware variant.

Other approaches are based on the specification of the metamorphic engine obtained by manually analyzing the metamorphic code. For example, in [21], Walenstein et al. propose to model the metamorphic engine as a term-rewriting systems. The authors then reduce the problem of recognizing metamorphic variants to the normalizer construction problem (NCP), namely to the problem of ensuring a normal form to a term-rewriting system. Indeed, program normalization (when possible) provides an efficient way to remove the variety introduced by metamorphism. The authors assume that the metamorphic engine to analyze is already known and it is represented by a set of rewriting rules. This knowledge is typically the result of a time and cost consuming tracking analysis, based on emulation and heuristics, which requires intensive human interaction in order to achieve an abstract specification of code features that are common to the malware variants obtained through various obfuscations and mutations. Conversely, we propose a tool that is able to automatically infer the set of rewriting rules that model the metamorphic engine. Indeed, as future work, it would be interesting to apply the NPC approach to the rewriting rules extracted with our methodology.

Structural similarity-based approaches, such as structural entropy and compression-based techniques, have been applied by Baysa et al. in [22] and Lee et al. in [23] to metamorphic malware detection. While structural entropy involves the examination of the raw bytes of the mutated file, compression-based detection involves the use of compression ratios of the mutated file in a bid to create sequences that represent the file. Rather than considering the raw bytes structure of metamorphic malware variants, we specify code mutations in terms of grammar rules extracted from the analysis of some metamorphic malware samples.

Wong and Stamp [24] proposed a statistical-based technique to detect metamorphic malware based on hidden Markov Model (HMM) and provided a benchmark used in other studies as Canforaet al. [25], Lin and Stamp [26], Musale et al. [27], Shanmugam et al. [28] on metamorphic malware. They analyzed the similarity degree of metamorphism produced by different malware generators, such as G2, MPCGEN, NGVCK and VCL32, by training HMM on the opcode sequences of the metamorphic malware samples. Instead, we base our metamorphic model on the formal language of pure context-free grammars and we analyze the similarity degree on rewriting rules, i.e., pure context-free grammars productions, captured by the learner.

In [12] Dalla Preda et al. follow the idea of extracting the specification of the metamorphic engine and of the possible code variants, from the metamorphic code analysis. They introduce a semantics for self-modifying code, called phase semantics, and prove its correctness by proving that it is an abstract interpretation of standard trace semantics. Phase semantics precisely models the metamorphic behaviour of the code, providing a set of program traces which correspond to the possible evolution of the metamorphic code during execution. Therefore, they demonstrate that metamorphic signatures can be automatically extracted by abstract interpretation of phase semantics. In particular, they introduce the notion of regular metamorphism, in which the invariants of phase semantics can be modeled as a FSA representing

the code structure of all possible metamorphic changes of a metamorphic code. As a matter of fact, the work in [12] can be considered as a general formal framework for modeling malware metamorphism, while our work provides a practical application of the formal framework that allows us to automatically extract the metamorphic signatures as a rewriting rules system.

3 Preliminaries

Mathematical notation Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \setminus T$ the set-difference between S and T, with $S \subset T$ strict inclusion and with $S \subseteq T$ inclusion. Let A^* be the set of finite sequences, also called strings, of elements of A. We denote with ϵ the empty string, with $|w|$ the length of the string $w \in A^*$, such that $|\epsilon| = 0$, and the concatenation of $w, v \in A^*$ as wv.

Finite State Automata (FSA) An *FSA* M is a tuple (Q, δ, S, F, A) where:

- Q is the set of states;
- $\delta : Q \times A \to \wp(Q)$ is the transition relation;
- $S \subseteq Q$ is the set of initial states;
- $F \subseteq Q$ is the set of final states;
- A is the finite alphabet of symbols.

A *transition* is a tuple (q, s, q), where $q, q' \in Q$ and $s \in A$, such that $q' \in \delta(q, s)$. Let $w \in A^*$ be a string of alphabet symbols, we denote with $\delta^* : Q \times A^* \to \wp(Q)$ the extension of δ to strings is defined as follow:

- $\delta^*(q, \epsilon) = q$;
- $\delta^*(q, ws) = \cup_{q' \in \delta^*(q,w)} \delta(q', s)$.

A string $w \in A^*$ is accepted by the automata M if there exists an initial state $q_0 \in S$ such that $\delta^*(q_0, w) \cap F \neq \emptyset$. The language $\mathcal{L}(M)$ accepted by an FSA M is the set of all strings accepted by M, that is:

$$\mathcal{L}(M) = \{w \in A^* \mid \exists q_0 \in S.\ \delta^*(q_0, w) \cap F \neq \emptyset\}$$

Given an FSA M and a partition π over its states, the *quotient* automaton over π, denoted M/π, is the tuple $M/\pi = (Q', \delta', S', F', A)$ induced by M, where:

- $Q' = \{[q]_\pi \mid q \in Q\}$ is the new set of states based on the partition π;
- $\delta' : Q' \times A \to \wp(Q')$ is the transition relation;
- $S' = \{[q]_\pi \mid q \in S\}$ the initial states;
- $F' = \{[q]_\pi \mid q \in F\}$ the final states.

The transition relation δ' is defined as:

$$\delta'([q]_\pi, s) = \cup_{p \in [q]_\pi} \left\{[q']_\pi \mid q' \in \delta(p, s)\right\}$$

An FSA $M = (Q, \delta, S, F, A)$ can be equivalently specified as a *graph* $M = (Q, E, S, F)$ with a node $q \in Q$ for each automaton state and a labeled edge $(q, s, q') \in E$ if and only if $q' \in \delta(q, s)$. We define the *language* of length k of a node of an automaton as the set of all the strings of length less or equal than k that are reachable from the considered node [11].

Widening automata We refer to the widening operation over FSA described by D'Silva [11]. We consider an increasing sequence $M_0 M_1 \ldots M_k$ of FSA ordered with respect to language inclusion, that is:

$$\mathcal{L}(M_0) \subseteq \mathcal{L}(M_1) \subseteq \cdots \subseteq \mathcal{L}(M_k)$$

where $\mathcal{L}(M_i)$, with $i \in [0, k]$, is the language recognized by M_i. Moreover, D'Silva in his thesis considers a fix-point computation of a function H on automata where

$$\mathcal{L}(M_{i+1}) = \mathcal{L}(M_i) \cup \mathcal{L}(H(M_i))$$

Take two FSA over a finite alphabet A in the considered sequence $M_i = (Q_i, E_i, S_i)$ and $M_j = (Q_j, E_j, S_j)$ with $i < j$, where Q is the set of states, $E \subseteq Q \times A \times Q$ is the set of triples representing edges between states, and $S \subseteq Q$ is the set of initial states. The *widening* between M_i and M_j is formalized in terms of an equivalence relation $R \subseteq Q_i \times Q_j$ between the set of states of the two automata. The equivalence relation R, also called *widening seed*, is used to define another equivalence relation $\equiv_R \subseteq Q_j \times Q_j$ over the states of M_j, such that:

$$\equiv_R = R \circ R^{-1}$$

The widening between M_i and M_j, denoted $M_i \triangledown M_j$, is given by the quotient of M_j with respect to the partition induced by $\equiv_R$:

$$M_i \triangledown M_j = M_j / \equiv_R$$

By changing the widening seed, i.e., the equivalence relation R, we obtain different widening operators. It has been proved in [11] that convergence is guaranteed when the widening seed is the relation $R_n \subseteq Q_i \times Q_j$ such that $(q_i, q_j) \in R_n$ if q_i and q_j recognize the same language of strings of length at most n. When considering the widening seed R_n, we have that two states q and q' of M_j are in equivalence relation $\equiv_{R_n}$ if they recognize the same language of strings of length at most n that is recognized by a state r of M_i, i.e., if there exists $r \in Q_i : (r, q) \in R_n$ and $(r, q') \in R_n$. Thus, the parameter n tunes the length of the strings that we consider

for establishing the equivalence of states and, therefore, for merging them in the widening, namely the more abstract will be the result of the widening. Observe that, the smaller is n the more information will be lost by the widening. We denote with $\triangledown_n$ the widening operator that uses R_n as widening seed.

Pure grammars A *pure grammar* [10] is a triple $G = (\Sigma, P, S)$ where:

- Σ is a finite alphabet;
- $S \subseteq \Sigma^*$ is a finite set of words called axioms;
- P is a finite set of ordered couples (x, y) of words in the alphabet Σ.

Elements of P are called productions and denoted by $x \to y$. Given a pure grammar G, a word $w \in \Sigma^*$ directly derives a word $w' \in \Sigma^*$, written $w \underset{G}{\Rightarrow} w'$, briefly $w \Rightarrow w'$ if G is understood, if there exist two words $w_1, w_2 \in \Sigma^*$ and a production $x \to y \in P$ such that:

$$w = w_1 x w_2 \ \wedge \ w' = w_1 y w_2$$

The reflexive transitive closure of the relation $\Rightarrow$ is written $\overset{*}{\Rightarrow}$. The language generated by a pure grammar G is called *pure language* [10] and it is defined by:

$$\mathcal{L}(G) \ = \ \{w \in \Sigma^* \mid s \overset{*}{\Rightarrow} w \text{ with } s \in S\}$$

If the left part of each production $x \to y$ of G is composed of one letter, namely $|x| = 1$, then G is called *pure context-free* (PCF) grammar. The language generated by a PCF grammar is called *pure context-free language* [10]. A PCF grammar can be viewed as a context-free grammar with no non-terminal symbols.

Example 1 Consider the language $L_1 = \{a^n c b^n \mid n \geq 1\}$. L_1 is generated by a PCF grammar with axiom acb and production $c \to acb$. This means that L_1 is a PCF language. Consider now the language $L_2 = \{a^n b^n \mid n \geq 1\}$. L_2 is generated by a pure grammar with axiom ab and production $ab \to a^2 b^2$. By analyzing the possible productions for a and b, it is easy to conclude that L_2 is a pure language and not PCF. Note that, despite L_2 is non-PCF, it is a regular language.

To conclude, we recall some important results with regard to pure languages and the other classes of languages in the Chomsky hierarchy. We denote with *PURE* the class of pure languages, *PCF* the class of PCF languages, *CF* the class of context-free languages and *REG* the class of regular languages. The following statements are proved in [10,29]:

- $PCF \subset PURE$;
- $REG \subset PURE$;

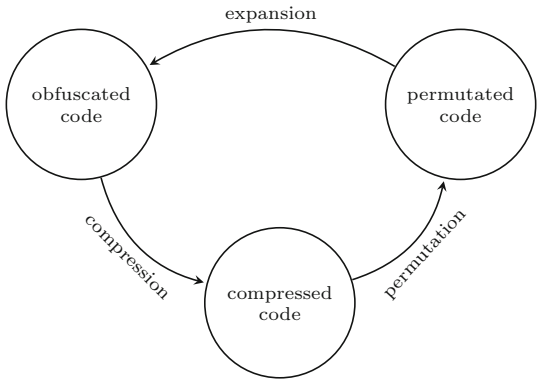

Fig. 3 *MetaPHOR* metamorphic engine behaviour

- $REG \neq PCF$ (see Example 1);
- $CF \neq PURE$;
- $PCF \subset CF$.

4 The approach

As argued in the previous sections, in this work we aim at defining an automatic technique for the extraction of a metamorphic signature that does not need any a priori knowledge of the code transformation rules used by the metamorphic engine, where this engine could be implemented either internally in the malware code or through an external program, like in Android malware [30]. Our work is based on the analysis of the metamorphic process of the metamorphic malware *MetaPHOR*. *MetaPHOR* [4], also known as *Win32/Smile* or *ETAP*, was written by the malware writer Mental Driller. It was released in the most recent version in March 2002 and it is a cross-platform infector capable of infecting both Windows 32-bit files and Linux ELF files. The Mental Driller named it *MetaPHOR* from the words *"Metamorphic Permutating High-Obfuscating Reassembler"*, which accurately describes this malware. An innovative feature of the *MetaPHOR* metamorphic engine is the use of an intermediate representation which allows to abstract away from the complexity of the underlying processor's instruction-set and to simplify the metamorphic transformations.

Besides the other malware phases, the most interesting one for our purposes is the metamorphic phase, which follows the general structure presented in Sect. 1 (see Fig.3): disassembly, compression, permutation, expansion, reassembly. The disassembly phase translates binary code in the intermediate representation mentioned before, while the reassemply phase converts it back to binary code. The malware writer points out that these two phases are the most difficult and delicate since they represent the 80% of the malware code and if an error occurs during one of these phases, the entire malware execution crashes (with possible error windows) [4]. The

basic idea of the compression process is to compress in one instruction what the expansion process codes in many. This is achieved by randomly applying transformations that are the inverse of the ones used by the expansion process. Examples of semantic-preserving transformations implemented in *MetaPHOR* are [4]:

1. `Instr` → `Instr` rules:
   - `xor Reg ; -1` → `not Reg`
   - `sub Reg ; Imm` → `add Reg,Imm`
2. `Instr ; Instr` → `Instr` rules:
   - `push Imm ; pop Reg` → `mov Reg,Imm`
   - `mov Mem,Imm ; push Mem` → `push Imm`
3. `Instr; Instr ; Instr` → `Instr` rules:
   - `mov Mem,Reg; op Mem,Reg2; mov Reg, Mem` → `op Reg,Reg2`
4. `Instr ; Instr ; Instr` → `Instr ; Instr` rules:
   - `mov Mem,Reg; test Mem,Reg2; jcc @xxx` → `test Reg,Reg2 ; jcc @xxx`

The permutation phase simply splits the code into blocks of random size and, once they have been computed and shuffled in memory, they are linked by direct jump-instructions and a jump at the first code block is inserted at the very beginning of the code. For a detailed description of the *MetaPHOR* implementation, the reader can refer to [4].

Our idea consists in assembling an approximated representation of the considered metamorphic variants and, afterwards, using a *learning grammar* technique in order to extract the rewriting rules that could form a metamorphic signature (Fig. 2). The approximating process is called *regular metamorphism* and it consists in a single automaton, i.e., a regular language, created by the *widening* operation on the CFG representations of the malware variants. The language recognized by this automaton over approximates the possible execution paths of the considered metamorphic variants. Hence, the two core parts of our approach are: the *regular metamorphism* and the application of a *learning algorithm* capable of inferring the possible transformation rules used by the metamorphic malware.

Regular metamorphism Rather than consider the standard CFG model with nodes given by program's instructions, we consider an abstract CFG where the operands of each instruction are removed through an abstraction function:

$$\alpha : Instr \rightarrow Instr$$

For example, the instruction `mov eax, 0` is abstracted by α in `mov`, i.e., α(`mov eax, 0`) = `mov`. We have chosen this abstraction in order to be independent from the particular locations used in the expansion/compression process (location renaming can be handled by using symbolic names). After generating the language of each node of the CFG of each metamorphic variant, we apply the widening operation, choosing as order the number of their instructions . If $V_0 V_1 \ldots V_k$ are an ordered sequence of metamorphic variants, then we denote with $\mathcal{G}(V_i)$ the CFG representation of program V_i. Hence, depending on the widening seed R_n, we compute the following widening sequence:

$$W_0 = \alpha(\mathcal{G}(V_0)) \quad W_{i+1} = W_i \bigtriangledown_n (W_i \cup \alpha(\mathcal{G}(V_{i+1}))) \qquad (1)$$

The automaton W_k is built as the limit of the above widening sequence and it recognizes all the metamorphic variants $V_0 V_1 \ldots V_k$.

Example 2 Figure 4 shows the widening of two CFGs with the language length of each node set to 2. Note that each node represents the instruction executed when the control flow reaches that node. This means that, the language of length 2 of a node is given by the instruction in it and all the subsequent instructions directly connected to it. For example, the initial node with a `mov` instruction recognizes the language

```
((mov),(mov,cmp))
```

while the `jge` node recognizes

```
((jge),(jge,sub),(jge,add))
```

If we increase the widening language length to 3, we obtain the CFG in Fig. 5. Note that the graph of both results recognizes the same language, but the CFG in Fig. 5 presents more nodes. This is due to the higher precision in merging nodes, i.e., less nodes present the same language.

Learning rewriting rules The general problem of inferring a grammar starting from an input set of strings of that language (the positive set), is an interdisciplinary field studied for decades and, particularly, in the last years [13]. This problem can be transformed into the general problem of inferring a grammar starting from a set of strings belonging to a language. In particular, we try to infer a grammar that is able to generate, at least, all the strings given as input to the algorithm and belonging to the language to be studied. The goal is to learn a set of rewriting rules that can generate all the possible metamorphic variants which also can be generated by the unknown metamorphic engine we are trying to defeat. Pure grammars have been chosen as a formal representation for the rewriting rules, because they do not present terminal symbols but all the symbols are considered as non-terminals.

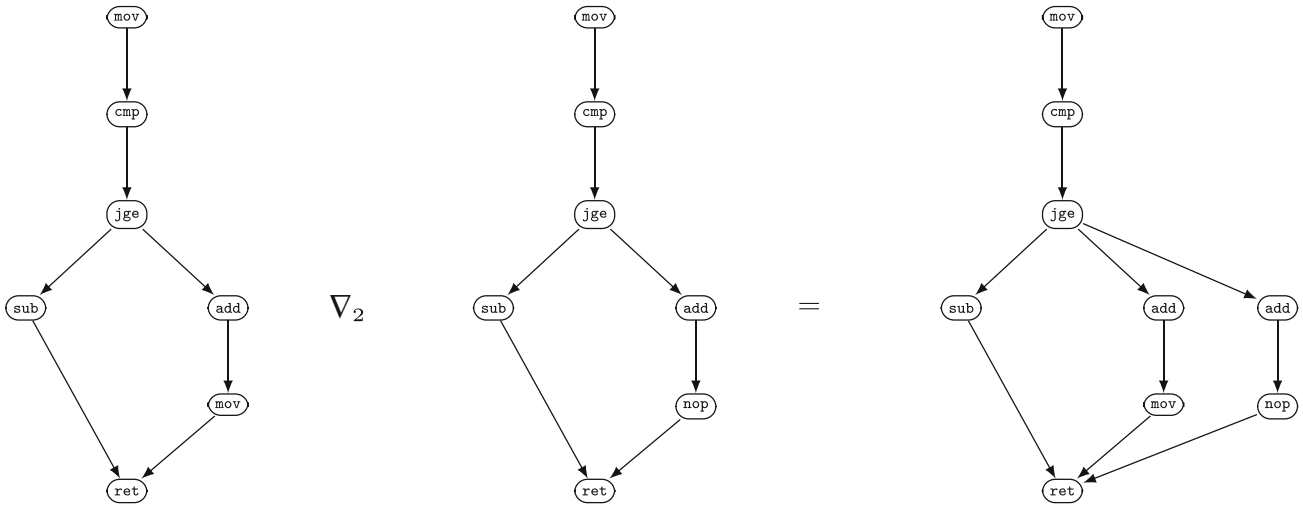

Fig. 4 Widening between two CFGs with widening seed set to 2

In fact, the metamorphic transformation rules are instructions of the same type, that is, they can be transformed into other instructions by applying the correct production. Since the general problem of learning pure grammars from a positive set is undecidable [14], we move to the formalism of pure context-free grammars where each production has the left part size fixed to one symbol. Obviously, this restriction will lead to a loss of precision in the rules inferred by the tool as it will only be possible to infer productions of the form $\{x \to y \mid |x| = 1\}$.

We can infer an approximated set of rewriting rules from W_k by analyzing the "positive set" containing all minimum paths, i.e., strings of words that will cover all the edges connecting the initial nodes and the final nodes. In our case, the language to learn consists of sequences of instructions forming the metamorphic variants given as input (assuming that they have been generated by a metamorphic engine). Starting from the positive set obtained from the widening CFG W_k, we apply a simple PCF grammars learner algorithm in order to capture a (sub)set of transformation rules generating the variants given as input. The idea that stems from the learner algorithm [13], consists in constructing a transformation rule r such that, starting from two variants V_i, V_j where $V_i < V_j$ in terms of lines of code, we can obtain V_j from V_i by applying the rewriting rule r, written $V_i \xrightarrow{r} V_j$. The rule r is obtained from the simplification between the two variant strings without considering the `ret` instruction.

Example 3 Suppose that, from the positive set, we captured the following two paths (we omit the final `ret`):

```
xor,mov,push
xor,push,pop,push
```

By transforming the two variants as a rewriting rule and after the simplification process, we obtain:

$$\sout{\texttt{xor}},\texttt{mov},\sout{\texttt{push}} \to \sout{\texttt{xor}},\texttt{push},\texttt{pop},\sout{\texttt{push}}$$

Considering the left part has length 1, then the rule

$$\texttt{mov} \to \texttt{push},\texttt{pop}$$

is added to the set of inferred rules.

Finally, the algorithm proceeds by eliminating redundant rules. An inferred rewriting rule is called redundant, if it can be generated by other rules in the set of inferred rewriting rules.

Example 4 Let us suppose that the learning algorithm has already inferred the following rewriting rules:

1) $\texttt{mov} \to \texttt{mov},\texttt{mov}$
2) $\texttt{mov} \to \texttt{push},\texttt{pop}$

Suppose that the learner infers the rule:

$$\texttt{mov} \to \texttt{push},\texttt{pop},\texttt{mov},\texttt{mov}$$

This rule is superfluous because:

$$\texttt{push},\texttt{pop},\underline{\texttt{mov}},\underline{\texttt{mov}} \overset{1)}{\Rightarrow} \underline{\texttt{push}},\underline{\texttt{pop}},\texttt{mov} \overset{2)}{\Rightarrow} \texttt{mov},\texttt{mov}$$

That is, starting from the right part of the new inferred rule, we obtained the right part of rule 1) and both rules have the same left instruction `mov`. Therefore, we can conclude that the new rewriting rule is redundant.

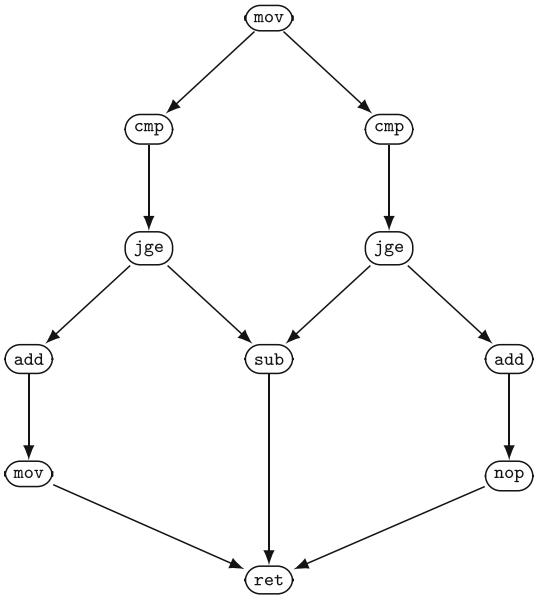

Fig. 5 Widening seed set to 3 (∇_3)

5 From theory to practice: *MetaSign* tool

In this section, we present *MetaSign*[1] an engine tool written in *Python 3*. This tool is able to automatically extract an approximation of the metamorphic malware signature from a set of metamorphic variants written in an intermediate `x86`-like language. In particular, a metamorphic signature consists of a set of rewriting rules that might be used by the metamorphic engine to generate the whole set of variants given as input. We say "might" because the real transformation rules could differ from the ones generated by the tool, due to several reasons: the approximation process of the widening function, an input set that is not large enough, the high dissimilarity in the code of different variants, and so on. Besides the learning rewriting rules function, the tool provides the possibility of randomly generating variants starting from an input program and a fixed set of rewriting rules.

Depending on the execution parameters, the tool can be executed in one of the following ways (see Fig. 6):

- execution of the metamorphic engine to generate a desired number of variants starting from a set of instructions (which will be the starting program) in a `x86`-like language written on an input text file (①);
- extraction of the widening CFG from a set of variants given as input in order to build an unique abstract representation of the considered metamorphic variants (②);
- inferring the rewriting rules from the variants approximation obtained through the widening process applied to a set of variants, e.g., a subset of the first phase (② → ③);
- finally, you can run all the operations above (① → ② → ③).

Following the idea of *MetaPHOR*, our tool generates and reads variants written in an intermediate language, both with the aim of simplifying and abstracting the `x86` assembly language. The intermediate language consists of the classic `x86` instructions having *Intel* syntax for:

- data manipulation: `mov, push, pop, lea`;
- mathematical operations: `add, sub, and, xor, or`;
- comparison: `cmp, test`;
- conditional jumps: `je, jne, jl, jle, jg, jge`;
- unconditional jumps: `jmp, call`;
- `ret, nop`.

There are three kinds of operands: registers (`eax, ebx, ecx, edx, esp, ebp, esi, edi`), integer values and memory values. Memory values are represented by square brackets between integer vales or registers, e.g., `[77382]` and `[eax]`, meaning that the real value is pointed by the address memory `77382` or the address memory inside `eax`. New memory values, added by transformation rules, are generated randomly. For jump instructions, the memory value to which the instruction can jump corresponds to the line number where the target instruction is located (the first line starts from zero). Analogously, for function calls, the operand of instruction `call` corresponds to the line number of the first statement of the function. Each function (including the main function) must end with the instruction `ret`.

Example 5 Consider the following program P, where the numbers on the left correspond to line numbers:

```
0: mov eax, 1      4: jmp 1
1: cmp eax, 1000   5: ret
2: jge 5           6: add eax, 1
3: call 6          7: ret
```

The CFG of the above program is depicted in Fig. 7.

In case of jump or call instructions, the control flow follows the line number indicated in the instruction (the first line of all programs starts with 0). Note that, call instructions start new functions with the first instruction pointed by the line number.

In the next three sections, we present thoroughly the three core parts of *MetaSign*, shown in Fig. 6: the metamorphic engine (Sect. 5.1), the widening on CFGs (Sect. 5.2) and the rewriting rules learner (Sect. 5.3).

[1] The latest version of the tool is freely available at https://github.com/LabSPY-univr/MetaSign.

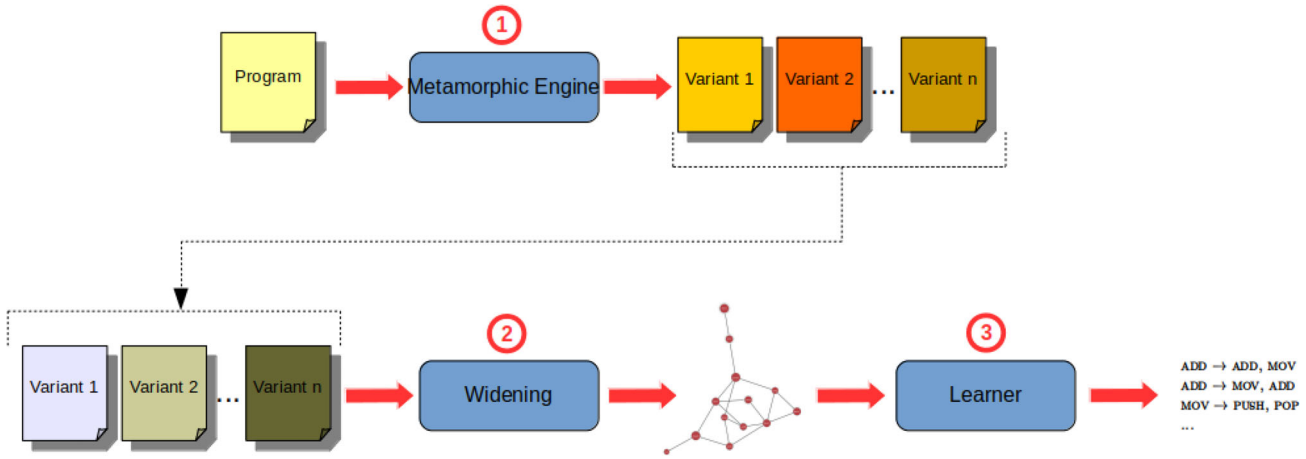

Fig. 6 *MetaSign* execution phases

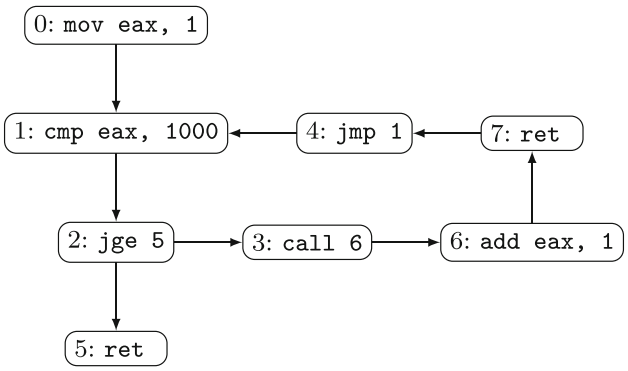

Fig. 7 CFG of Example 5

5.1 The metamorphic engine

MetaSign can be executed as a metamorphic engine: it takes as input a text file containing a sequence of instructions following the syntax of the intermediate language defined before, and the number of variants to be generated. This functionality will help us in estimating the output quality of the tool, simulating a set of real metamorphic variants generated by the same metamorphic engine. The implemented rewriting rules are a subset of the rules used by the *MetaPHOR* metamorphic engine. They consist in instruction transformations that preserve the semantics and they could be applied either to expand (following the rule from right to left) or to reduce (left to right) the selected instruction. On the time of writing this article, the following transformation rules are implemented in our tool:

- `push Imm ; pop Reg` ↔ `mov Reg,Imm`
- `push Reg ; pop Reg2` ↔ `mov Reg2,Reg`
- `mov Mem,Imm ; push Mem` ↔ `push Imm`
- `mov Mem,Reg ; push Mem` ↔ `push Reg`
- `pop Mem2 ; mov Mem,Mem2` ↔ `pop Mem`
- `pop Mem ; mov Reg,Mem` ↔ `pop Reg`
- `mov Mem,Imm ; OP Reg,Mem` ↔ `OP Reg,Imm`
- `mov Mem2,Mem ; OP Reg,Mem2` ↔ `OP Reg,Mem`
- `pop Mem ; push Mem` ↔ `nop`

where `Imm` denotes a numeric value, `Reg` a register, `Mem` a memory value (a number between square brackets), and `OP` is any operator accepting two operands. The left and right operands of the same rewriting rule must be the same. For example, in order to apply the compression rule number three, the two `Mem` values on the left must be equal, while, for expansion, the `Imm` will be equal to the `Imm` value on the two instructions generated. If we consider the α abstraction on the instructions presented in the previous section, and that `OP` could be either `mov, add, sub, and, or, xor, lea, cmp` or `test`, than we have *13* possible rewriting rules.

After reading the first input variant, the metamorphic engine randomly selects: the rewriting rule to apply, the line of the program where to apply the rule, and whether to apply the rule as expansion or reduction. If it is not possible to apply the rewriting rule to the selected instruction, the following instruction will be considered. If the rewriting rule does not fit any instructions, then another rewriting rule will be picked up randomly (see Fig. 8 for an example).

5.2 Widening control flow graphs

Each metamorphic variant is represented as a CFG. Each node of the CFG contains one instruction, e.g., `mov eax, 4` , while the edges represent possible control flow of the program execution. Afterwards, *MetaSign* abstracts instructions by eliminating the operands (the α abstraction defined in Sect. 4). In order to agglomerate all the CFGs of each metamorphic variant into an unique approximating CFG, we use a widening operator. This allows us to obtain a unique

```
jmp 4
mov eax,4
push eax
push 10
pop eax
```

Rule 1 (←)
mov → push,pop

```
jmp 5
push 4
pop eax
push eax
push 10
pop eax
```

Rule 4 (←)
push → mov,push

```
jmp 6
push 4
pop eax
mov [74538],eax
push [74538]
push 10
pop eax
```

Rule 1 (→)
push,pop → mov

```
jmp 5
push 4
pop eax
mov [74538],eax
push [74538]
mov eax,10
```

Fig. 8 Example of code transformations

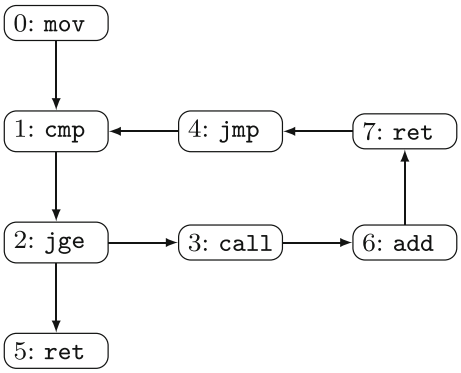

Fig. 9 Abstract CFG of Example 6

CFG that contains all the input metamorphic variants that generalizes the considered mutations. To this end, we have to compute the language of each node of the CFG.

Example 6 Consider again the program P in Example 5. The CFG obtained after the application of the abstraction α, is represented in Fig. 9.

The alphabet of the CFG of P is {mov, cmp, jge, call, jmp, add, ret}, and the language of length 2 recognized by each node is:

```
lang(0) = {(mov),(mov,cmp)}
lang(1) = {(cmp),(cmp,jge)}
lang(2) = {(jge),(jge,call),(jge,ret)}
lang(3) = {(call),(call,add)}
lang(4) = {(add),(add,ret)}
lang(5) = {(ret),(ret,jmp)}
lang(6) = {(jmp),(jmp,cmp)}
lang(7) = {(ret)}
```

Starting from an increasing sequence of variants, in terms of number of instructions, $V_0 V_1 V_2 \ldots V_n$, the widening operator ∇_k is defined in Equation (1). At the end, the widening operator merges all the nodes with the same language of length k. The pseudocode in Algorithm 1 represents the implementation of the widening process.

The algorithm takes as inputs the CFG of the current widening W_i augmented, for each node, with its language of length k, and outputs the widening CFG $W_{i+1} = W_i \nabla_k (W_i \cup \alpha(V_{i+1}))$. The function markNodes, marks all the nodes to be merged in one node if they present the same language and it saves them in the list to_merge, while function genLang generates the new language of length k of each node in the CFG W_{i+1}.

Algorithm 1 Widening algorithm

```
1: procedure WIDENING(W_i, V_{i+1}, k)
2:     W_{i+1} ← graphUnion(W_i, V_{i+1})
3:     to_merge ← markNodes(W_{i+1})
4:     while to_merge ≠ ∅ do
5:         mergeNodes(to_merge, W_{i+1})
6:         to_merge ← markNodes(W_{i+1})
7:     end while
8:     genLang(W_{i+1}, k)
9:     return W_{i+1}
10: end procedure
```

5.3 Rewriting rules learner

The learning algorithm implemented in *MetaSign* is a simplified version of the algorithm proposed in [13] for learning pure grammars from a set of words. It takes as input a CFG and it follows three phases:

1. creating the positive set;
2. learning rewriting rules;
3. eliminating spurious inferred rules.

The positive set consists of a set of code variants where all the instructions are abstracted. This set will be the input from which the rewriting rules are inferred. The length min of the smallest variant is calculated, i.e., the variant with the fewest instructions. Then, all the paths of length min of the graph that go from a root node (the first instruction) to the final node (the ret instruction) are visited. For each path found, the set of instructions related to the visited nodes are inserted in the positive set. During this process, each visited edge will be marked. When the path of length min has been found, if all the edges are marked then the search is interrupted without visiting other paths. Otherwise the variable min is incremented.

Let (V_i, V_j) be a pair of variants, with $|V_i| < |V_j|$. The idea of the learning algorithm is to add a production rule r of the form $V_i \xrightarrow{r} V_j$. The rewriting rule r is inferred through simplification rules between the two variants (V_i, V_j). The pseudocode of the learning algorithm is presented in Algorithm 2.

There are three kinds of simplification rules:

Algorithm 2 Learner algorithm

```
1: procedure INFER(positive_set)
2:     for (V_i, V_j) ∈ positive_set, V_i < V_j do
3:         rules = ∅
4:         while |V_i| ≠ 1 do
5:             left_simplify(V_i, V_j)
6:             if no change then
7:                 break
8:             end if
9:         end while
10:        rules = rules ∪ {V_i → V_j}
11:        while |V_i| ≠ 1 do      ▷ V_i, V_j restored to the previous value
12:            right_simplify(V_i, V_j)
13:            if no change then
14:                break
15:            end if
16:        end while
17:        rules = rules ∪ {V_i → V_j}
18:        while |V_i| ≠ 1 do      ▷ V_i, V_j restored to the previous value
19:            leftright_simplify(V_i, V_j)
20:            if no change then
21:                break
22:            end if
23:        end while
24:        rules = rules ∪ {V_i → V_j} ▷ V_i, V_j restored to the previous
    value
25:    end for
26:    removes_spurious(rules)
27:    return rules
28: end procedure
```

– left_simplify: compare the first instruction of V_i and V_j and delete them if they are the same. This process continues until two different instructions are encountered: in this case, if $|V_i| > 1$ then the comparison restarts from the last instruction of V_i and V_j, otherwise ($|V_i| = 1$) the rule is added to the set of inferred rules;
– right_simplify: it is similar to the previous one, but starts from the last instruction;
– leftright_simplify: compare the first instruction of V_i and V_j and, if they are equal, it deletes them and starts again but from the bottom instruction of V_i and V_j.

The algorithm applies the left_simplify repeatedly until a rule is added to the set of inferred rules and then it starts back with right_simplify and, finally, with function leftright_simplify.

After the simplification phase, the algorithm produces a set of rewriting rules of the form $\{x \to y \mid |x| = 1\}$. However, most of these rules are superfluous since they can be generated by other inferred rules. The elimination algorithm (removes_spurious) tries to reduce the right part of each rewriting rule by applying all rewriting rules inferred in the reduction form (from right to left). If at the end of this procedure, the rule is reduced to another rule already in the inferred set, then that rule can be eliminated.

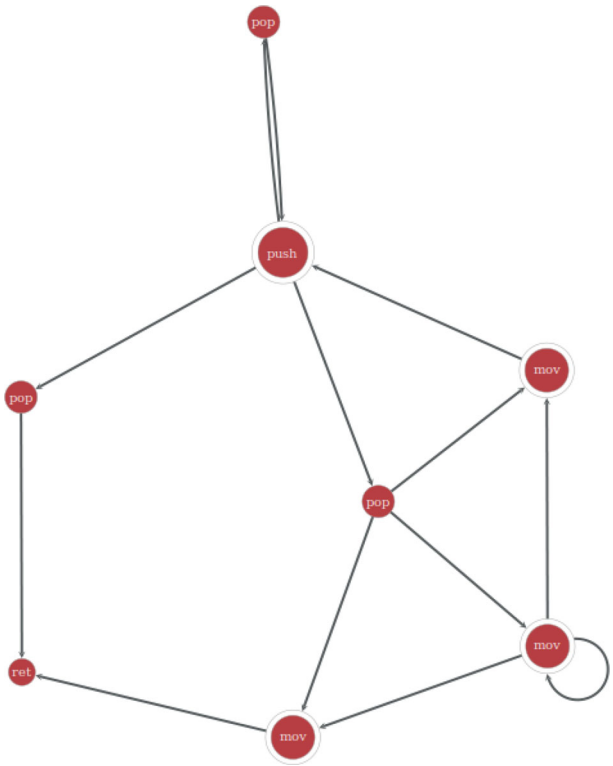

Fig. 10 Widening graph recognizing metamorphic variants of P (the language length is set to 2)

6 Case studies

In this section, we present some results and considerations applied to three case studies.

We start with a simple case study taken from [12] where the authors manually built the graph accepting all the possible metamorphic variants generated from the following program: [P : mov e, 10] The hidden metamorphic engine applies the following transformation rules:

R1 `push` e_2 ; `pop` e_1 ↔ `mov` e_1,e_2
R2 `mov` e_2, e_1 ; `push` e_2 ↔ `push` e_1
R3 `pop` e_2 ; `mov` e_1,e_2 ↔ `pop` e_1

Note that, by substituting e with either an immediate value `Imm`, a register or a memory value, in the above three rules we obtain the first six rules presented in Sect. 5.1 which are implemented in *MetaSign*. Moreover, both the example in [12] and *MetaSign* abstract instructions by removing the operands. *MetaSign* allows us to automatically generate numerous metamorphic variants of program P and to automatically derive the widening graph reported in Fig. 10.

Starting from this graph the learner of *MetaSign* automatically derives the following set of rewriting rules:

```
mov -> ['push', 'pop']
```

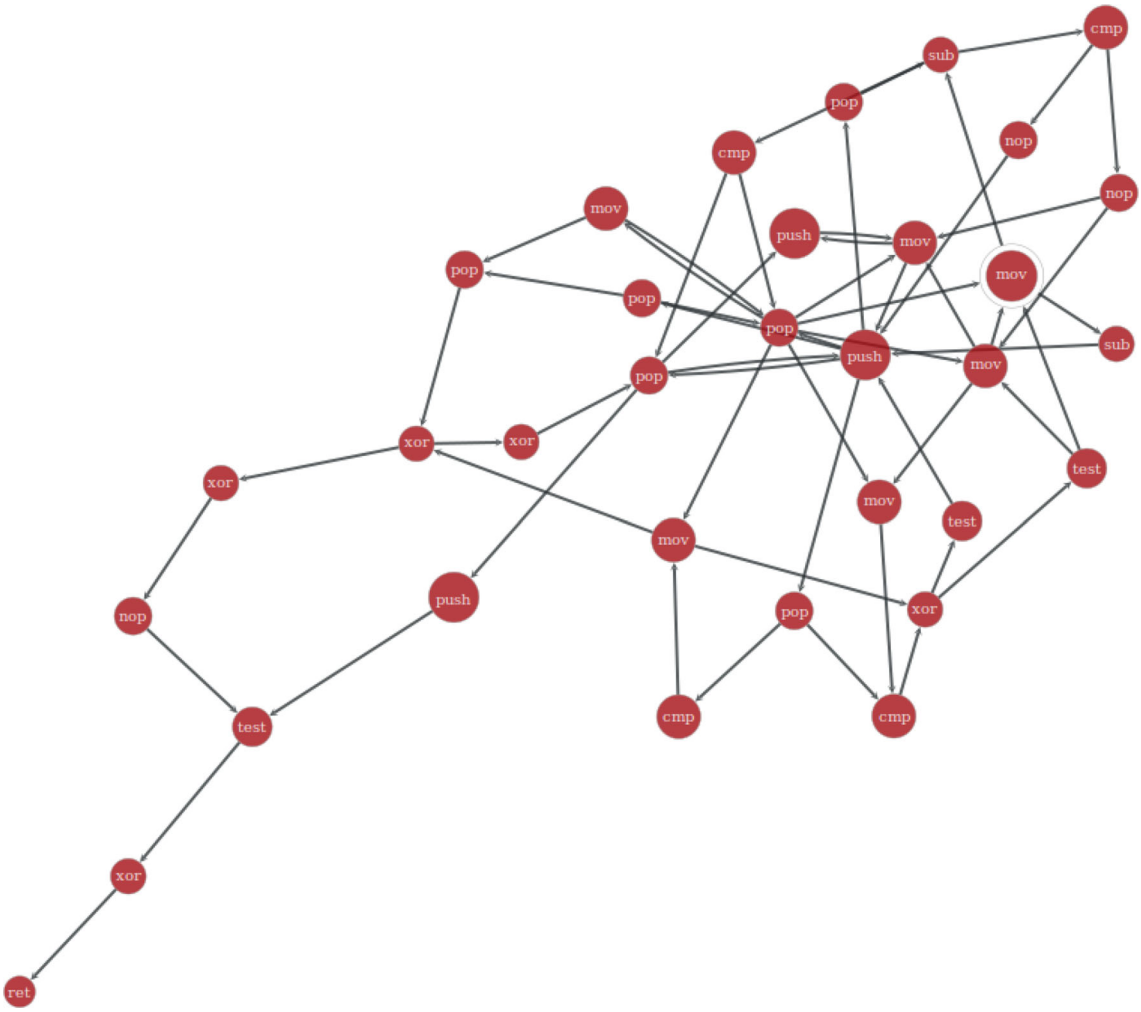

Fig. 11 The graph obtained by the widening operator with language length set to 2

```
mov -> ['mov', 'mov']
pop -> ['pop', 'mov']
push -> ['mov', 'push']
```

Our learner successfully captures the three rules used by the unknown metamorphic malware and also a spurious rule `mov` → `mov, mov` induced by the widening approximation.

In the second case study, we compare results obtained with different widening seeds. To this end, we set an initial starting program *Test* (Fig. 12) which has no malicious behaviour: it is just a set of random instructions. Then, we use the *MetaSign* metamorphic engine to generate from *Test* 50 semantically equivalent variants, by randomly applying rules presented in Sect. 5.1. After that, we randomly select a subset of 25 code variants. This subset is provided as input to the widening process (with language length set to 2) and, finally, to the learning algorithm. The resulting widening graph, produced as output by the tool, is shown in Fig. 11. Unfortunately, the learner could not infer any rewriting rules. At first glance, this result may appear as an error in the implementation, however, by looking more carefully at the possible paths of the graph, we observe that all paths from any root node to the `ret` node, starting from the minimum length (the smallest variant in terms of instructions), will mark all the edges as visited. For this reason, the set of positive examples contains all code variants of the same length and therefore it is not possible to infer any rewriting rules. This result is caused by the numerous spurious variants injected by the widening process that agglomerates the nodes with the same language of length 2. In fact, due to the numerous cycles, i.e., regularities inserted by the widening, a path from the root to the end node of length less than any actual real variant, is "increased" until reaching the minimum length (in this example equal to 20), thus creating a spurious variant.

If we increase the level of precision of the widening by setting the parameter of the language length to 3, we obtain the graph in Fig. 13 and the following rewriting rules inferred by the learner:

```
0: mov [ebp], [esp]   11: mov eax, ebx
1: sub ebp, 4         12: push eax
2: push 100           13: pop [440303]
3: pop ecx            14: pop [443905]
4: cmp eax, exc       15: xor eax, 0
5: xor eax, 0         16: xor eax, eax
6: test eax, eax      17: nop
7: mov eax, 4         18: test eax, 0
8: sub eax, 1         19: xor eax, 0
9: cmp eax, ebx       20: ret
10: nop
```

Fig. 12 *Test* program

```
cmp -> ['cmp', 'mov']     mov -> ['push', 'mov']
mov -> ['push', 'pop']    mov -> ['mov', 'push']
mov -> ['pop', 'mov']     nop -> ['pop', 'push']
nop -> ['pop', 'mov']     nop -> ['nop', 'mov']
pop -> ['pop', 'push']    pop -> ['pop', 'mov']
pop -> ['mov', 'pop']     pop -> ['nop', 'mov']
push -> ['mov', 'push']   sub -> ['mov', 'sub']
test -> ['test', 'mov']   xor -> ['mov', 'xor']
```

Clearly, by increasing the length of the widening language seed, we obtain a graph with more nodes, hence, more precise. In fact, it is possible to infer the rewriting rules even if there are numerous spurious rules still due to the presence of spurious paths induced by the widening.

If we increase by another unit the level of precision of the widening, setting the length of the language to 4 (Fig. 14) we obtain the following rewriting rules inferred:

```
cmp -> ['cmp', 'mov']     cmp -> ['mov', 'cmp']
mov -> ['push', 'pop']    mov -> ['mov', 'mov']
nop -> ['nop', 'mov']     nop -> ['pop', 'push']
pop -> ['pop', 'mov']     pop -> ['mov', 'pop']
push -> ['mov', 'push']   sub -> ['mov', 'sub']
test -> ['test', 'mov']   xor -> ['mov', 'xor']
```

Thanks to the greater precision of the widening, this time the inferred rules are more precise and, despite the presence of rules that are not used by the metamorphic engine, they represent an acceptable result. Moreover, setting language length to 5, we obtain the same result as before. Observe that the precision of the widening graph improves by increasing the length of the language used as widening seed, and this implies a downgrade in the performances of the algorithm in terms of space and time consumption.

As third case study, we evaluate the learner outputs on three samples using the *precision* and *recall* metrics. The precision metric shows the percentage of how many correct rewriting rules *MetaSign* has caught with respect to all the inferred rules, while the recall metric is the fraction of the total amount of correct rewriting rules that were retrieved by the learner, with respect to the (unknown) rules actually used by the metamorphic engine. Formally:

$$\text{Precision} = \frac{tP}{tP + fP} \qquad \text{Recall} = \frac{tP}{tP + fN}$$

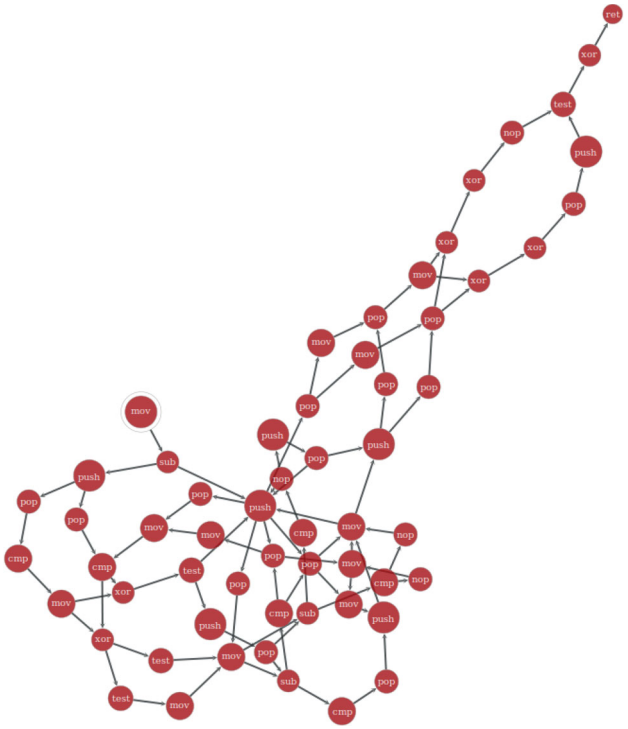

Fig. 13 The graph obtained by the widening operator with language length set to 3

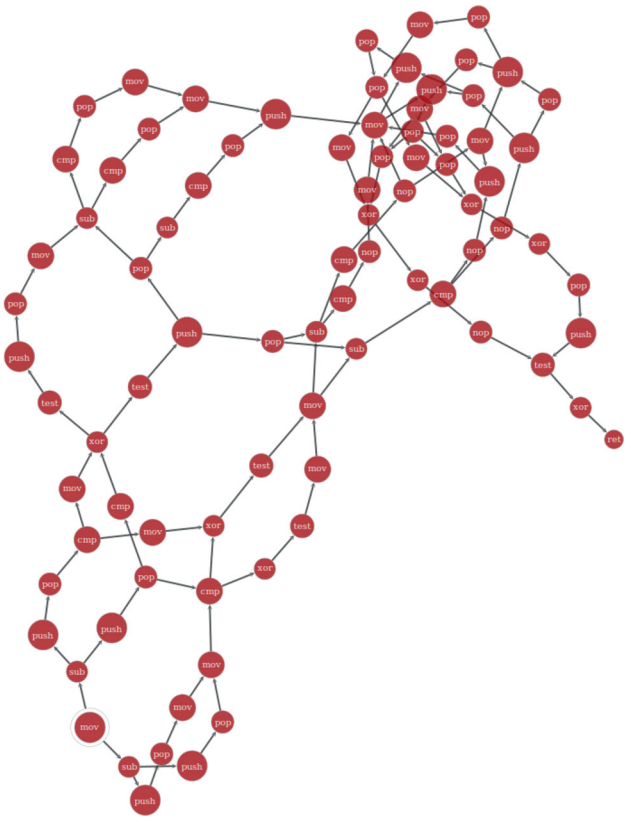

Fig. 14 The graph obtained by the widening operator with language length set to 4

```
0: push ebp          10: sub ecx, 1
1: mov ebp, esp      11: cmp ecx, 0
2: mov eax, [77800]  12: jne 8
3: mov ecx, [77900]  13: xor edx, 0
4: lea edx, [77950]  14: and edx, 0
5: mov ebx, eax      15: mov esp, ebp
6: add ebx, ecx      16: pop ebp
7: sub ebx, 1        17: nop
8: xor [ebx], 1      18: ret
9: sub ebx, 1
```

Fig. 15 *XOR-Encryption* routine

Table 1 Results for *Test* program

Widening	Precision (%)	Recall (%)
∇_3	33,56	100
∇_4	50,13	100
∇_5	57,34	100
∇_6	64,10	100

Table 2 Results for *Chernobyl/CHI* code

Widening	Precision (%)	Recall (%)
∇_3	40,05	100
∇_4	57,42	100
∇_5	72,72	100
∇_6	72,72	100

where tP (true positive) is the number of correct rewriting rules inferred, fP (false positive) the number of wrong rewriting rules inferred, while fN (false negative) is the number of missed correct rewriting rules. We considered three programs for evaluating the results: the *Test* program shown in Fig. 12, a fragment of the obfuscated *Chernobyl/CHI* malware code shown in Fig. 1 (the second on the left), and a simple *XOR-Encryption* routine, reported in Fig. 15, where we inserted some dead code in order to trigger more rewriting rules. For each sample, we generated 200 variants through our *MetaSign* metamorphic engine. On *Test* and *Chernobyl/CHI* codes, *MetaSign* was able to apply 8 of the 13 rewriting rules implemented, while 10 on the *XOR-Encryption* routine. Clearly, this result depends on the program instructions. For example, *MetaSign* can not apply neither the rewriting rule `cmp` $\rightarrow$ `mov`, `cmp` nor `and` $\rightarrow$ `mov`, `and` on the *Chernobyl/CHI* code, because the instructions `cmp` and `and` are not used. Finally, for each program, we randomly selected three (possibly non-disjoint) subsets of 50 variants and we provide them as inputs to the learner algorithm. Tables 1, 2 and 3 report the arithmetic mean of precision and recall of the results of the learning on the three subsets for the different widening seed.

It is worth to note that, this extended set of applications supports the observation made in the previous case study: by increasing the widening seed, we get higher precision.

Table 3 Results for *XOR-Encryption* routine

Widening	Precision (%)	Recall (%)
∇_3	34,89	90
∇_4	44,47	90
∇_5	58,52	90
∇_6	58,52	90

However, when running the experiments with the widening ∇_6 on *Chernobyl/CHI* and *XOR-Encryption*, we obtain the same results for precision and recall as with widening ∇_5. This happens because we have reached a sort of fixpoint on the precision that we can obtain with the considered learning algorithm. Better results can be achieved by exploring other algorithms for learning grammars from positive examples. Nonetheless, on the *Test* and *Chernobyl/CHI* we caught all the rewriting rules used by the metamorphic engine, while on *XOR-Encryption*, on average, we missed one rule on the three subsets.

7 Discussion and future directions

The results of the previous section are promising for three main reasons. Firstly, we developed a tool capable of automatically inferring rewriting rules starting from a widening approximation, i.e., without the need of any manual analysis. We can tune the widening precision through the language length parameter in order to get better results for the inferred rules by the learner. Obviously, an important role is also played by the quality of the learner, which can be a limit when a more precise results is needed. Second, the implementation is parametric on both the rewriting rules implemented and the abstraction function on instructions. This means that, it can be easily editable in order to test new transformation rules with different abstractions. Finally, the third reason consists in the way we can populate a set of input test: the metamorphic engine implemented in *MetaSign* allows us to easily create numerous variants according the engine. Therefore, the quality of the learner can be quickly tested on the generated metamorphic variants.

On the other hand, we are aware that numerous future works need to be accomplish in order to complete the process of our new methodology in analyzing metamorphic behaviors. As a priority of future work, we will try to apply this tool to a set of real malware variants. Currently, our proposed tool reads program code written in our intermediate `x86`-like language. The conversion process should be automated by an ad-hoc disassembler, similar to the one implemented in *MetaPHOR*. In this way, an automated metamorphic signature extraction program can be implemented:

given a set of disassembled payload of metamorphic variants, or parts of them, generated by the same metamorphic engine, we can feed them to *MetaSign* in order to extract a possible metamorphic signature, i.e., a set of rewriting rules used by the unknown metamorphic engine. In this work, we considered one level of abstraction on the instructions, that is, we discard the operands of each assembly instruction. Clearly, this strong abstraction influences the accuracy results since *MetaSign* is more likely to capture a rule by visiting a widening CFG where all nodes, i.e., instructions, have no arguments. It would be interesting to consider different abstractions assigning to the operands, for example, symbolic values such as those of [31]. We would like to point out that *MetaSign* is implemented from scratch, i.e., with no code reuse from other implemented software. Space and time complexity of the whole program need to be optimized as this task was not a priority for our purposes. The current implementation can be extended with new rewriting rules and a new learner algorithm. Currently, only rules having form $\{x \to y, y \to x \mid |x| = 1 \wedge |y| = 2\}$ are implemented. The learning algorithm is a critical core part, as the third case study highlighted: an optimal learner algorithm should exploit the tuning of the widening language length parameter. Moreover, the new learner needs to be able to learn, in an approximate way, more complex rewriting rules in order to catch more sophisticated metamorphic engines and it should be able to generate a sound metamorphic signature, i.e., a set of rules that generate all the real metamorphic variants, namely, all the possible variants that the unknown metamorphic engine can create, admitting some false positives (spurious variants). Finally, new more expressive formal languages should be considered modeling code mutations, such as, e.g., indexed grammars [32,33] or context-sensitive grammars, and their respective learning algorithms.

8 Conclusion

Metamorphic attacks require new semantic models and analysis for coping with unknown obfuscation strategies, generated by an unknown metamorphic engine. In order to model the self-modifying nature of a metamorphic malware, it is necessary to provide a model of program behavior that allows the program to change during execution. In this work, we tried to capture the behavior of the metamorphic engine itself, namely we tried to find a set of rules that allow us to predict possible mutations of code variants starting from a set of examples. To this end, we presented *MetaSign*, an automated metamorphic signature extraction that has three main functions: metamorphic engine, widening of code variants and learning of rewriting rules. Thanks to the metamorphic engine, it is possible to quickly generate numerous variants using an intermediate `x86`-like language. These variants are created by randomly applying rewriting rules implemented in *MetaSign*. Starting from a set of metamorphic code variants, the goal is to capture the unknown rewriting rules used by the metamorphic engine to generate them. *MetaSign* uses a widening operator to generate a CFG that approximates all the variants of the input set. Rewriting rules are then represented as productions of a pure context-free grammar. From the learning algorithm and the elimination of superfluous rewriting rules algorithm, it is possible to obtain a set of rules that describes, in an approximate way, the possible evolution of code variants. The experimental results show two strictly correlated key points. The first point consists in the choice of the language length parameter of the widening operator which affects the precision of the learned rules. The lower the value is, the more the nodes are merged together because they are more likely to present the same language. In this case, the presence of spurious paths will be definitely higher, and, therefore, there will be less precision in the results inferred by the learner. On the contrary, the higher the length of the language is, the greater is the precision of the widening CFG, namely, fewer spurious paths. Obviously, the increase in precision comes at a cost in terms of time execution and memory consumption. The second key point involve the learner implementation which has to exploit a more detailed widening CFG in terms of instructions.

Funding Open Access funding provided by Università degli Studi di Verona.

References

1. Schwab, W., and Poujol, M.: "The state of industrial cybersecurity 2018," *Trend Study Kaspersky Reports*, p. 33 (2018)
2. Institute's, P.: Ponemon institute's 2018 state of endpoint security risk (2018)
3. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional, New York (2005)
4. Beaucamps, P.: Advanced metamorphic techniques in computer viruses (2007)
5. Aycock, J.: *Computer viruses and malware*, vol. 22. Springer Science & Business Media, New York (2006)
6. OKane, P., Sezer, S., McLaughlin, K.: Obfuscation: the hidden malware. IEEE Security Privacy **9**(5), 41–47 (2011)

7. Bruschi, D., Martignoni, L., Monga, M.: Code normalization for self-mutating malware. IEEE Secur. Priv. **5**(2), 46–54 (2007)
8. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. Tech. rep., Wisconsin Univ-Madison Dept of Computer Sciences (2006)
9. Dalla Preda, M.: The grand challenge in metamorphic analysis. In: International Conference on Information Systems, Technology and Management, pp. 439–444, Springer, Berlin (2012)
10. Maurer, H.A., Salomaa, A., Wood, D.: Pure grammars. Inf. Control **44**(1), 47–72 (1980)
11. D'Silva, V.: Widening for automata. Diploma Thesis, Institut Fur Informatick, Universitat Zurich (2006)
12. Dalla Preda, M., Giacobazzi, R., Debray, S.K.: Unveiling metamorphism by abstract interpretation of code properties. Theor. Comput. Sci. **577**, 74–97 (2015)
13. De la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, Cambridge (2010)
14. Koshiba, T., Mäkinen, E., Takada, Y.: Inferring pure context-free languages from positive data. Acta Cybernetica **14**(3), 469–477 (2000)
15. Degenbaev, U.: Formal specification of the x86 instruction set architecture (2012)
16. Bergeron,J., Debbabi, M., Desharnais, J., Erhioui, M.M., Lavoie, Y., Tawbi, N.: Static detection of malicious code in executable programs. In: Symposium on Requirements Engineering for Information Security (2001)
17. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, pp. 169–186 (2003)
18. Singh, P., Lakhotia, A.: Static verification of worm and virus behaviour in binary executables using model checking. In: Proceedings of the 4th IEEE Information Assurance Workshop (2003)
19. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05), vol. 3548 of LNCS, pp. 174–187 (2005)
20. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P'05), pp. 32–46 (2005)
21. Walenstein, A., Mathur, R., Chouchane, M.R., Lakhotia, A.: Constructing malware normalizers using term rewriting. J. Comput. Virol. **4**(4), 307–322 (2008)
22. Baysa, D., Low, R.M., Stamp, M.: Structural entropy and metamorphic malware. J. Comput. Virol. Hacking Tech. **9**(4), 179–192 (2013)
23. Lee, J., Austin, T.H., Stamp, M.: Compression-based analysis of metamorphic malware. Int. J. Secure. Network. **10**(2), 124–136 (2015)
24. Wong, W., Stamp, M.: Hunting for metamorphic engines. J. Comput. Virol. **2**(3), 211–229 (2006)
25. Canfora, G., Iannaccone, A.N., Visaggio, C.A.: Static analysis for the detection of metamorphic computer viruses using repeated-instructions counting heuristics. J. Comput. Virol. Hacking Tech. **10**(1), 11–27 (2014)
26. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. J. Comput. Virol. **7**(3), 201–214 (2011)
27. Musale, M., Austin, T.H., Stamp, M.: Hunting for metamorphic javascript malware. J. Comput. Virol. Hacking Tech. **11**(2), 89–102 (2015)
28. Shanmugam, G., Low, R.M., Stamp, M.: Simple substitution distance and metamorphic detection. J. Comput. Virol. Hacking Tech. **9**(3), 159–170 (2013)
29. Bucher, W., Hagauer, J.: It is decidable whether a regular language is pure context-free. Theoret. Comput. Sci. **26**(1–2), 233–241 (1983)
30. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109, IEEE (2012)
31. Lakhotia, V., Dalla Preda, M., Giacobazzi, R.: Fast location of similar code fragments using semantic 'juice'. In: 2nd Workshop on Program Protection and Reverse Engineering PPREW 2013, ACM (2013)
32. Aho, A.V.: Indexed grammars - an extension of context-free grammars. J. ACM **15**(4), 647–671 (1968)
33. Campion, M., Dalla Preda, M., and Giacobazzi, R.: Abstract interpretation of indexed grammars. In: International Static Analysis Symposium, pp. 121–139, Springer, Berlin (2019)