

Unveiling Metamorphism

by Abstract Interpretation of Code Properties

Mila Dalla Preda^a, Roberto Giacobazzi^a, Saumya Debray^b

^aUniversity of Verona, Italy, {mila.dallapreda,roberto.giacobazzi}@univr.it

^bUniversity of Arizona, USA, debray@cs.arizona.edu

Abstract

Metamorphic code includes self-modifying semantics-preserving transformations to exploit code diversification. The impact of metamorphism is growing in security and code protection technologies, both for preventing malicious host attacks, e.g., in software diversification for IP and integrity protection, and in malicious software attacks, e.g., in metamorphic malware self-modifying their own code in order to foil detection systems based on signature matching. In this paper we consider the problem of automatically extracting metamorphic signatures from metamorphic code. We introduce a semantics for self-modifying code, later called *phase semantics*, and prove its correctness by showing that it is an abstract interpretation of the standard trace semantics. Phase semantics precisely models the metamorphic code behavior by providing a set of traces of programs which correspond to the possible evolutions of the metamorphic code during execution. We show that metamorphic signatures can be automatically extracted by abstract interpretation of the phase semantics. In particular, we introduce the notion of regular metamorphism, where the invariants of the phase semantics can be modeled as finite state automata representing the code structure of all possible metamorphic change of a metamorphic code, and we provide a static signature extraction algorithm for metamorphic code where metamorphic signatures are approximated in regular metamorphism.

Keywords: Abstract interpretation, program semantics, metamorphic malware detection, self-modifying programs.

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. Traditional misuse malware detectors (also known as *signature-based detectors*) are typically syntactic in nature: they use pattern matching to compare the byte sequence comprising the body of the malware against a *signature database* [34]. Malware writers have responded by using a variety of techniques in order to avoid detection: Encryption, oligomorphism with mutational decryption patterns, and polymorphism with different encryption methods for generating an endless sequence of decryption

patterns are typical strategies for achieving malware diversification. Metamorphism emerged in the last decade as an effective alternative strategy to foil misuse malware detectors. Metamorphic malware apply semantics-preserving transformations to modify its own code so that one instance of the malware bears very little resemblance to another instance, in a kind of *body-polymorphism* [33], even though semantically their functionality is the same. Thus, a metamorphic malware is a malware equipped with a *metamorphic engine* that takes the malware, or parts of it, as input and morphs it at runtime to a syntactically different but semantically equivalent variant, in order to avoid detection. Some of the basic metamorphic transformations commonly used by malware are semantic-nop/junk insertion, code permutation, register swap and substitution of equivalent sequences of instructions [7, 33]. It is worth noting that most of these transformations can be seen as special cases of code substitution [23]. The quantity of metamorphic variants possible for a particular piece of malware makes it impractical to maintain a signature set that is large enough to cover most or all of these variants, making standard signature-based detection ineffective [10]. Existing malware detectors therefore fall back on a variety of heuristic techniques, but these may be prone to false positives (where innocuous files are mistakenly identified as malware) or false negatives (where malware escape detection) at worst. The reason for this vulnerability to metamorphism lies upon the purely syntactic nature of most existing and commercial detectors. The key for identifying metamorphic malware lies, instead, in a deeper understanding of their semantics. Preliminary works in this direction by Dalla Preda et al. [16], Christodorescu et al. [11], and Kinder et al. [27] confirm the potential benefits of a semantics-based approach to malware detection. Still a major drawback of existing semantics-based methods relies upon their need of an *a priori* knowledge of the obfuscations used to implement the metamorphic engine. Because of this, it is always possible for any expert malware writer to develop alternative metamorphic strategies, even by simple modification of existing ones, able to foil any given detection scheme. Indeed, handling metamorphism represents one of the main challenges in modern malware analysis and detection [18].

Contributions. 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. We propose a different approach to metamorphic malware detection based on the idea that *extracting metamorphic signatures is approximating malware semantics*. Program semantics concerns here the way code changes, i.e., the effect of instructions that modify other instructions. We face the problem of determining how code mutates, by catching properties of this mutation, without any a priori knowledge about the implementation of the metamorphic transformations. We use a formal semantics to model the execution behavior of self-modifying code commonly encountered in malware. Using this as the basis, we propose a theoretical model for statically deriving, by abstract interpretation, an abstract specification of all possible code variants that can be generated during the execution of a metamorphic malware. Traditional static analysis techniques are not adequate for this purpose, as they typically assume that programs do not change during execution. We therefore define a more general semantics-based

behavioral model, called *phase semantics*, that can cope with changes to the program code at run time. The idea is to partition each possible execution trace of a metamorphic program into *phases*, each collecting the computations performed by a particular code variant. The sequence of phases (once disassembled) represents the sequence of possible code mutations, while the sequence of states within a given phase represents the behavior of a particular code variant.

Phase semantics precisely expresses all the possible phases, namely code variants, that can be generated during the execution of a metamorphic code. Phase semantics can then be used as a metamorphic signature for checking whether a program is a metamorphic variant of another one. Indeed, thanks to the precision of phase semantics, we have that a program Q is a metamorphic variant of a program P if and only if Q appears in the phase semantics of P . Unfortunately, due to the possible infinite sequences of code variants that can be present in the phase semantics of P , the above test for metamorphism is undecidable in general. Thus, in order to gain decidability, we need to loose precision and do so by using the well established theory of abstract interpretation [12, 13]. Indeed, abstract interpretation is used here to extract the invariant properties of phases, which are properties of the generated program variants. Abstract domains represent here properties of the code shape in phases. We use the domain of finite state automata (FSA) for approximating phases and provide a static semantics of traces of FSA as an abstraction of the phase semantics. We introduce the notion of *regular metamorphism* as a further approximation obtained by abstracting sequences of FSA into a single FSA. This abstraction provides an upper regular language-based approximation of *any* metamorphic behavior of a program and it leads to a decidable test for metamorphism. This is particularly suitable to extract metamorphic signatures for engines implemented themselves as FSA of basic code transformations, which correspond to the way most classical metamorphic generators are implemented [23, 31, 35]. Our approach is general and language independent, providing an adequate theoretical foundation for the systematic design of algorithms and methods devoted to the extraction of approximate metamorphic signatures from any metamorphic code P . The main advantage of the phase semantics here is in modeling code mutations without isolating the metamorphic engine from the rest of the viral code. The approximation of the phase semantics by abstract interpretation can make decidable whether a given binary matches a metamorphic signature, without knowing any features of the metamorphic engine itself.

Structure of the paper. In Section 3 we describe the behavior of a metamorphic program as a graph, later called *program evolution graph*, where each vertex is a standard static representation of programs (e.g., a control flow graph) and whose edges represent possible run-time changes to the code. We then define the phase semantics of a program as the set of all possible paths in the program evolution graph and we prove its correctness by showing that it is a sound abstract interpretation of standard trace semantics. Thus, phase semantics provides a precise description of the history of run-time code modifications, namely the sequences of “code snapshots” that can be generated during execution. Then, in Section 4, we introduce a general method for extracting metamorphic signatures as abstract interpretation of phase semantics. The result of the analysis is a correct approximate specification of the malware evolution, namely an

approximated representation of all its possible metamorphic variants. This method is instantiated in Section 5 by abstracting programs with FSA over the alphabet of abstract binary instructions and phase semantics by sequences of FSA. Here, the abstract phase semantics is given by a set of sequences of FSA and it provides a behavioral model for the evolution of any metamorphic program. Next, in Section 6, we introduce regular metamorphism modeling metamorphic engines as FSA of basic code transformations. This is achieved by approximating the phase semantics of program P as a unique FSA $\mathbf{W}[[P]]$, whose language contains the strings of instructions that corresponds to the runs of all the metamorphic variants of P . $\mathbf{W}[[P]]$ provides an approximated metamorphic signature of P which can be used to verify whether a program is a metamorphic variant of P by language inclusion which is decidable for regular languages. In Section 7 we discuss how the presented approach could be applied to the analysis of the metamorphic virus MetaPHOR. The paper ends with a discussion and related works.

The results presented in this work are an extended and reviewed version of [17].

2. Background

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 S_\perp be set S augmented with the *undefined value* \perp , i.e., $S_\perp = S \cup \{\perp\}$. $\langle P, \leq \rangle$ denotes a poset P with ordering relation \leq , while $\langle P, \leq, \vee, \wedge, \top, \perp \rangle$ denotes a complete lattice P , with ordering \leq , least upper bound (lub) \vee , greatest lower bound (glb) \wedge , greatest element (top) \top , and least element (bottom) \perp . Often, \leq_P will be used to denote the underlying ordering of a poset P , and \vee_P, \wedge_P, \top_P and \perp_P denote the basic operations and elements of a complete lattice P . We use the symbol \sqsubseteq to denote pointwise ordering between functions: If X is any set, $\langle P, \leq \rangle$ is a poset and $f, g : X \rightarrow P$ then $f \sqsubseteq g$ if for all $x \in X$, $f(x) \leq g(x)$. If $f : S \rightarrow T$ and $g : T \rightarrow Q$ then $g \circ f : S \rightarrow Q$ denotes the composition of f and g , i.e., $g \circ f = \lambda x. g(f(x))$. A function $f : P \rightarrow Q$ on posets is (Scott)-continuous when it preserves the lub of countable chains in P . A function $f : C \rightarrow D$ on complete lattices is additive when for any $Y \subseteq C$, $f(\vee_C Y) = \vee_D f(Y)$, and it is co-additive when $f(\wedge_C Y) = \wedge_D f(Y)$. Given a function $f : S \rightarrow T$, its point-wise extension from $\wp(S)$ to $\wp(T)$ is $\lambda X. \{ f(x) \mid x \in X \}$.

Let A^* be the set of finite sequences, also called strings, of elements of A with ϵ the empty string, and with $|\omega|$ the length of string $\omega \in A^*$. We denote the concatenation of $\omega, \nu \in A^*$ as $\omega :: \nu$. We say that a string $s_0 \dots s_h$ is a subsequence of a string $t_0 \dots t_n$, denoted $s_0 \dots s_h \preceq t_0 t_1 \dots t_n$, if there exists $l \in [1, n] : \forall i \in [0, h] : s_i = t_{l+i}$.

Finite State Automata (FSA). An FSA M is a tuple (Q, δ, S, F, A) , where Q is the set of states, $\delta : Q \times A \rightarrow \wp(Q)$ is the transition relation, $S \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states and A is the finite alphabet of symbols. A transition is a tuple (q, s, q') such that $q' \in \delta(q, s)$. Let $\omega \in A^*$, we denote with $\delta^* : Q \times A^* \rightarrow \wp(Q)$ the extension of δ to strings: $\delta^*(q, \epsilon) = \{q\}$ and $\delta^*(q, \omega s) = \bigcup_{q' \in \delta^*(q, \omega)} \delta(q', s)$. A string $\omega \in A^*$ is accepted by M if there exists $q_0 \in S : \delta^*(q_0, \omega) \cap F \neq \emptyset$. The language $\mathcal{L}(M)$ accepted by an FSA M is the set of all strings accepted by M . Given an FSA M and a partition π over its states, the *quotient automaton* $M/\pi =$

(Q', δ', S', F', A) is defined as follows: $Q' = \{[q]_\pi \mid q \in Q\}$, $\delta' : Q' \times A \rightarrow \wp(Q')$ is the function $\delta'([q]_\pi, s) = \bigcup_{p \in [q]_\pi} \{[q']_\pi \mid q' \in \delta(p, s)\}$, $S' = \{[q]_\pi \mid q \in S\}$, and $F' = \{[q]_\pi \mid q \in F\}$. 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 automata state and a labeled edge $(q, s, q') \in E$ if and only if $q' \in \delta(q, s)$.

Abstract Interpretation. Abstract interpretation is based on the idea that the behavior of a program at different levels of abstraction is an approximation of its (concrete) semantics [12, 13]. The concrete program semantics is computed on the so-called concrete domain, i.e., the poset of mathematical objects on which the program runs, here denoted by $\langle C, \leq_C \rangle$, where the ordering relation encodes relative precision: $c_1 \leq_C c_2$ means that c_1 is a more precise (concrete) description than c_2 . Approximation is encoded by an abstract domain $\langle A, \leq_A \rangle$, which is a poset of abstract values that represent some approximated properties of concrete objects and whose partial order models relative precision. In abstract interpretation abstraction is formally specified as a Galois connection (GC) (C, α, γ, A) , i.e., an adjunction [12, 13], namely as a concrete domain C and an abstract domain A related through an abstraction map $\alpha : C \rightarrow A$ and a concretization map $\gamma : A \rightarrow C$ such that: $\forall a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Thus, $\alpha(c) \leq_A a$ or equivalently $c \leq_C \gamma(a)$ means that a is a sound approximation in A of c . GCs ensure that $\alpha(c)$ actually provides the best possible approximation in the abstract domain A of the concrete value $c \in C$. Recall that a tuple (C, α, γ, A) is a GC iff α is additive iff γ is co-additive. This means that given an additive (resp. co-additive) function f between two domains we can always build a GC by considering its right (resp. left) adjoint map. In particular, we have that every abstraction map α induces a concretization map γ and vice versa, formally: $\gamma(y) = \bigvee \{x \mid \alpha(x) \leq y\}$ and $\alpha(x) = \bigwedge \{y \mid x \leq \gamma(y)\}$. As usual, we denote a GC as a tuple (C, α, γ, A) . Given two GCs $(C, \alpha_1, \gamma_1, A_1)$ and $(A_1, \alpha_2, \gamma_2, A_2)$, their composition $(C, \alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2, A_2)$ is still a GC. Abstract domains can be compared with respect to their relative degree of precision: If A_1 and A_2 are both abstract domains of a common concrete domain C , we say that A_1 is more precise than A_2 when for any $a_2 \in A_2$ there exists $a_1 \in A_1$ such that $\gamma_1(a_1) = \gamma_2(a_2)$, i.e., when $\gamma_2(A_2) \subseteq \gamma_1(A_1)$. Given a GC (C, α, γ, A) and a concrete predicate transformer (semantics) $F : C \rightarrow C$, we say that $F^\sharp : A \rightarrow A$ is a *sound*, i.e., correct, approximation of F in A if for any $c \in C$ and $a \in A$, if a approximates c then $F^\sharp(a)$ must approximate $F(c)$, i.e., $\forall c \in C, \alpha(F(c)) \leq_A F^\sharp(\alpha(c))$. When $\alpha \circ F = F^\sharp \circ \alpha$, the abstract function F^\sharp is a *complete* abstraction of F in A . While any abstract domain induces the canonical *best correct approximation* $\alpha \circ F \circ \gamma$ of $F : C \rightarrow C$ in A , not all abstract domains induce a complete abstraction [24]. The least fix-point of an operator F on a poset $\langle P, \leq \rangle$, when it exists, is denoted by $\text{lfp}^{\leq} F$, or by $\text{lfp} F$ when the partial order is clear from the context. Any continuous operator $F : C \rightarrow C$ on a complete lattice $C = \langle C, \leq_C, \bigvee_C, \bigwedge_C, \top_C, \perp_C \rangle$ admits a least fix-point: $\text{lfp}^{\leq_C} F = \bigvee_{n \in \mathbb{N}} F^n(\perp_C)$, where for any $i \in \mathbb{N}$ and $x \in C$: $F^0(x) = x$; $F^{i+1}(x) = F(F^i(x))$. If $F^\sharp : A \rightarrow A$ is a correct approximation of $F : C \rightarrow C$ on $\langle A, \leq_A \rangle$, i.e., if $\alpha \circ F \sqsubseteq F^\sharp \circ \alpha$, then $\alpha(\text{lfp}^{\leq_C} F) \leq_A \text{lfp}^{\leq_A} F^\sharp$. Convergence can be ensured through *widening* iterations along increasing chains [12]. A widening operator $\nabla : P \times P \rightarrow P$ approximates the lub, i.e., $\forall X, Y \in P : X \leq_P (X \nabla Y)$ and $Y \leq_P (X \nabla Y)$, and it is such that the increasing chain W^i , where $W^0 = \perp$ and

Syntactic categories:

$n \in Val$	(values)
$a \in Loc$	(memory locations)
$e \in \mathbb{E}$	(expressions)
$I \in \mathbb{I}$	(instructions)
$m \in \mathcal{M} : Loc \rightarrow Val_{\perp}$	(memory map)
$P \in Loc \times \mathcal{M} \times Val^* = \mathbb{P}$	(programs)

Expressions:

$e ::= n \mid MEM[e] \mid MEM[e_1] \text{ op } MEM[e_2] \mid MEM[e_1] \text{ op } n$

Instructions:

$I ::= \text{call } e \mid \text{ret} \mid \text{pop } e \mid \text{push } e \mid \text{nop} \mid \text{mov } e_1, e_2 \mid \text{input} \Rightarrow e \mid$
 $\text{if } e_1 \text{ goto } e_2 \mid \text{goto } e \mid \text{halt}$

Figure 1: Syntax of an abstract assembly language

$W^{i+1} = W^i \nabla F(W^i)$ is not strictly increasing for \leq_P . The limit of the sequence W^i provides an upper-approximation of the lfp of F on P , i.e., $\text{lfp}^{\leq_P} F \leq_P \lim_{i \rightarrow \infty} W^i$.

3. Modeling Metamorphism*3.1. Abstract Assembly Language*

Executable programs make no fundamental distinction between code and data: Memory locations contain binary values that can be interpreted either as representing data or as encoding instructions. This makes it possible to modify a program by operating on a memory location as though it contains data, e.g., by adding or subtracting some value from it, and then interpret the result as code and execute it. To model this, we define a program to be a tuple $P = (a, m, \theta)$, where a denotes the *entry point* of P , namely the address of the first instruction of P , m specifies the contents of memory (both code and data), and θ represents the program stack. Without loss of generality we consider a finite set $Loc \subseteq \mathbb{N}$ of possible memory locations and a finite set Val of possible values that can be stored either in the memory or in the stack. Since a memory location contains a natural number that can be interpreted either as data or as instruction¹ we use an injective function $\text{encode} : \mathbb{I} \rightarrow Val$ that given an instruction $I \in \mathbb{I}$ returns its binary representation $\text{encode}(I) \in Val$, and a function $\text{decode} : Val \rightarrow \mathbb{I}_{\perp}$ such that:

$$\text{decode}(n) = \begin{cases} I & \text{if } I \in \mathbb{I} \text{ and } \text{encode}(I) = n \\ \perp & \text{otherwise} \end{cases}$$

The syntax of the abstract assembly language, whose structure is inspired from IA-32 assembly language, that we are considering is shown in Figure 1. Instruction

¹To simplify the discussion, we assume that each instruction occupies a single location in memory. While this does not hold true of variable-instruction-length architectures such as the Intel IA-32, the issues raised by variable-length instructions are orthogonal to the topic of this paper, and do not affect any of our results.

$\text{input} \Rightarrow e$ represents the assignment of the value in input to the memory location identified by expression e ; instruction $\text{mov } e_1, e_2$ denotes the mov -instruction where e_1 is the destination and e_2 the source, while the other instructions have the usual meaning. Observe that the set of instructions identified by the syntax of Figure 1 is unbounded since it allows an unlimited number of expressions. However, the set Val of possible values that can be stored in memory is finite and this implies that also the set of instructions that can be encoded by Val is finite. In the rest of the paper we use \mathbb{I} to denote the finite set of instructions that can be encoded by Val . A program state is a tuple $\langle a, m, \theta, \mathcal{J} \rangle$ where m is the memory map, a is the address of the next instruction to be executed, $\theta \in Val^*$ is the content of the stack and $\mathcal{J} \in Val^*$ is the input string. In particular, we use notation $n :: \theta$ to refer to a stack that has value n on the top, and notation $n :: \mathcal{J}$ to refer to an input stream whose next input value is n . Let $\Sigma = Loc_{\perp} \times \mathcal{M} \times Val^* \times Val^*$ be the set of all possible program states. We denote with σ_i the i -th element of a sequence of states $\sigma \in \Sigma^*$. The semantics of expressions is specified by function $\mathcal{E} : \mathbb{E} \times \mathcal{M} \rightarrow Val$:

$$\begin{aligned} \mathcal{E}[[n]]_m &= n \\ \mathcal{E}[[MEM[e]]]_m &= m(\mathcal{E}[[e]]_m) \\ \mathcal{E}[[MEM[e_1] \text{ op } MEM[e_2]]]_m &= \mathcal{E}[[MEM[e_1]]]_m \text{ op } \mathcal{E}[[MEM[e_2]]]_m \\ \mathcal{E}[[MEM[e_1] \text{ op } n]]_m &= \mathcal{E}[[MEM[e_1]]]_m \text{ op } n \end{aligned}$$

and the semantics of instructions through function $\mathcal{I} : \mathbb{I} \times \Sigma \rightarrow \Sigma$:

$$\begin{aligned} \mathcal{I}[[\text{call } e]]\langle a, m, \theta, \mathcal{J} \rangle &= \langle \mathcal{E}[[e]]_m, m, (a+1) :: \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{ret}]]\langle a, m, n :: \theta, \mathcal{J} \rangle &= \langle n, m, \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{mov } e_1, e_2]]\langle a, m, \theta, \mathcal{J} \rangle &= \langle a+1, m[\mathcal{E}[[e_1]]_m \leftarrow \mathcal{E}[[e_2]]_m], \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{input} \Rightarrow e]]\langle a, m, \theta, n :: \mathcal{J} \rangle &= \langle a+1, m[\mathcal{E}[[e]]_m \leftarrow n], \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{if } e_1 \text{ goto } e_2]]\langle a, m, \theta, \mathcal{J} \rangle &= \begin{cases} \langle \mathcal{E}[[e_2]]_m, m, \theta, \mathcal{J} \rangle & \text{if } \mathcal{E}[[e_1]]_m \neq 0 \\ \langle a+1, m, \theta, \mathcal{J} \rangle & \text{otherwise} \end{cases} \\ \mathcal{I}[[\text{pop } e]]\langle a, m, n :: \theta, \mathcal{J} \rangle &= \langle a+1, m[\mathcal{E}[[e]]_m \leftarrow n], \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{goto } e]]\langle a, m, \theta, \mathcal{J} \rangle &= \langle \mathcal{E}[[e]]_m, m, \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{push } e]]\langle a, m, \theta, \mathcal{J} \rangle &= \langle a+1, m, \mathcal{E}[[e]]_m :: \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{halt}]]\langle a, m, \theta, \mathcal{J} \rangle &= \langle \perp, m, \theta, \mathcal{J} \rangle \\ \mathcal{I}[[\text{nop}]]\langle a, m, \theta, \mathcal{J} \rangle &= \langle a+1, m, \theta, \mathcal{J} \rangle \end{aligned}$$

Given the instruction semantics \mathcal{I} described above we can define the *transition relation* between states $\mathcal{T} : \Sigma \rightarrow \Sigma$ as:

$$\mathcal{T}(\langle a, m, \theta, \mathcal{J} \rangle) = \mathcal{I}[[\text{decode}(m(a))]]\langle a, m, \theta, \mathcal{J} \rangle$$

The *maximal finite trace semantics* of program $P = (a, m, \theta)$, denoted as $\mathbf{S}[[P]]$, is given by the least fix-point of the function $\mathcal{F}_{\mathcal{T}}[[P]] : \wp(\Sigma^*) \rightarrow \wp(\Sigma^*)$:

$$\mathcal{F}_{\mathcal{T}}[[P]](X) = \text{Init}[[P]] \cup \{ \sigma \sigma_i \sigma_{i+1} \mid \sigma_{i+1} = \mathcal{T}(\sigma_i), \sigma \sigma_i \in X \}$$

where $\text{Init}[[P]] = \{ \langle a, m, \theta, \mathcal{J} \rangle \mid P = (a, m, \theta), \mathcal{J} \in Val^* \text{ is an input stream} \}$ denotes the set of initial states of P .

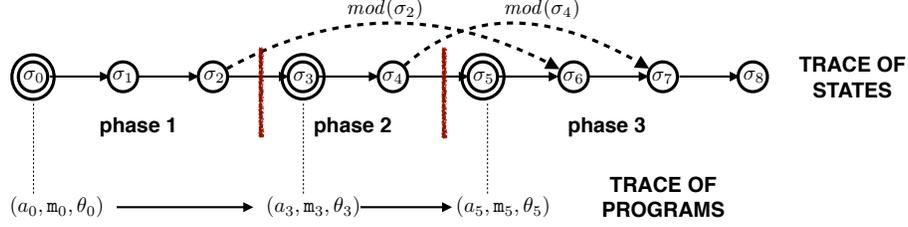


Figure 2: Phases of a trace

3.2. Phase Semantics

While the trace semantics of a program gives a faithful account of its low-level behavior, it is at far too fine granularity for our purposes. Since we are interested primarily in comparing the metamorphic variants of a malware, it is convenient to group together sequences of states into *phases* that correspond to the computations of a particular code variant. Thus, a phase is a maximal sequence of states in an execution trace that does not overwrite any memory location storing an instruction that is going to be executed later in the same trace. Given an execution trace $\sigma = \sigma_0 \dots \sigma_n$, we can identify *phase boundaries* by considering the sets of memory locations modified by each state $\sigma_i = \langle a_i, m_i, \theta_i, \mathcal{I}_i \rangle$ with $i \in [0, n]$: every time that a location a_j , with $i < j \leq n$, of a future instruction is modified by the execution of state σ_i , then the successive state σ_{i+1} is a phase boundary, since it stores a modified version of the code. We can express this more formally by denoting with $mod(\sigma_i) \subseteq Loc$ the set of memory locations that are modified by the instruction executed in state σ_i :

$$mod(\sigma_i) = \begin{cases} \{\mathcal{E}\llbracket e \rrbracket m\} & \text{if } decode(m_i(a_i)) \in \{move, e', input \Rightarrow e, pope\} \\ \emptyset & \text{otherwise} \end{cases}$$

This allows us to formally define the phase boundaries and the phases of a trace.

Definition 3.1. The set of phase boundaries of a trace $\sigma = \sigma_0 \dots \sigma_n \in \Sigma^*$, where $\forall i \in [0, n] : \sigma_i = \langle a_i, m_i, \theta_i, \mathcal{I}_i \rangle$, is:

$$bound(\sigma) = \{\sigma_0\} \cup \{\sigma_i \mid mod(\sigma_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}$$

The set of phases of a trace $\sigma \in \Sigma^*$ is:

$$phases(\sigma) = \left\{ \sigma_i \dots \sigma_j \mid \begin{array}{l} \sigma = \sigma_0 \dots \sigma_i \dots \sigma_j \sigma_{j+1} \dots \sigma_n, \\ \sigma_i, \sigma_{j+1} \in bound(\sigma), \forall l \in [i+1, j] : \sigma_l \notin bound(\sigma) \end{array} \right\}$$

Observe that, since a phase does not execute instructions that are modified within the phase, the memory map of the first state of a phase always specifies the code snapshot that is executed along the phase. Hence, the sequence of the initial states of the phases of an execution trace highlights the different code snapshots encountered during that execution. Figure 2 provides a graphical representation of the notions of phase

boundaries and phases inside a trace. Indeed, Figure 2 represents an execution trace $\sigma_0\sigma_1 \dots \sigma_8$ where $\sigma_i = \langle a_i, m_i, \theta_i, \mathfrak{J}_i \rangle$. Let us assume that the execution of state σ_2 overwrites the instruction that will be executed by state σ_6 , namely $a_6 \in \text{mod}(\sigma_2)$, and we graphically represent this with the dashed arrow from σ_2 to σ_6 labeled with $\text{mod}(\sigma_2)$. Analogously, the dashed arrow from σ_4 to σ_7 denotes the fact that the execution of state σ_4 overwrites the instruction executed by state σ_7 . In this situation, the phase boundaries of the trace are the initial state σ_0 and the states σ_3 and σ_5 that are immediately after the execution of a state that modifies an instruction that will be later executed. The phase boundaries of the considered trace are represented as states with a double circle. The phase boundaries are then used to identify the successive phases in the trace. As argued before the code executed by each phase is specified by the first state of the phase. Thus, the execution trace $\sigma_0\sigma_1 \dots \sigma_8$ corresponds to the the program evolution $(a_0, m_0, \theta_0)(a_3, m_3, \theta_3)(a_5, m_5, \theta_5)$ depicted at the bottom of Figure 2. In general, different executions of a program give rise to different sequences of code snapshots. A complete characterization of all code snapshots that can be obtained over any possible execution of a self-modifying program is given by the *program evolution graph*. Here, each vertex is a code snapshot P_i corresponding to a phase of a program execution trace, and an edge $P_i \rightarrow P_j$ indicates that in some execution trace of the program, a phase with code snapshot P_i can be followed by a phase with code snapshot P_j .

Definition 3.2. *The program evolution graph of a program P_0 is $\mathbf{G}[[P_0]] = (V, E)$:*

$$V = \{P_i = (a_i, m_i, \theta_i) \mid \sigma = \sigma_0.. \sigma_i.. \sigma_n \in \mathbf{S}[[P_0]] : \sigma_i = \langle a_i, m_i, \theta_i, \mathfrak{J}_i \rangle \in \text{bound}(\sigma)\}$$

$$E = \left\{ (P_i, P_j) \left| \begin{array}{l} P_i = (a_i, m_i, \theta_i), P_j = (a_j, m_j, \theta_j), \\ \sigma = \sigma_0.. \sigma_i.. \sigma_{j-1} \sigma_j.. \sigma_n \in \mathbf{S}[[P_0]] : \\ \sigma_i = \langle a_i, m_i, \theta_i, \mathfrak{J}_i \rangle, \sigma_j = \langle a_j, m_j, \theta_j, \mathfrak{J}_j \rangle, \\ \sigma_i \dots \sigma_{j-1} \in \text{phases}(\sigma) \end{array} \right. \right\}$$

If we consider the example in Figure 2 we have that the program evolution graph that we can build from the single trace of states $\sigma_0\sigma_1 \dots \sigma_8$ is the trace of programs depicted at the bottom of the figure, where the edges of the program evolution graph connect successive code variants. A path in $\mathbf{G}[[P_0]]$ is a sequence of programs $P_0 \dots P_n$ such that for every $i \in [0, n[$ we have that $(P_i, P_{i+1}) \in E$.

Definition 3.3. *Given a program P_0 , the set of all possible (finite) paths of the program evolution graph $\mathbf{G}[[P_0]]$ is the phase semantics of P_0 , denoted $\mathbf{S}^{Ph}[[P_0]]$:*

$$\mathbf{S}^{Ph}[[P_0]] = \{P_0 \dots P_n \mid P_0 \dots P_n \text{ is a path in } \mathbf{G}[[P_0]]\}$$

Example. Consider for instance the metamorphic program P_0 of Figure 3, where the numbers on the left denote memory locations. x and y are memory locations that will store the input or the result of manipulation of the input data, while l is a memory location that contains the location where P_0 will write instructions that will then be executed. During its execution program P_0 loops between the modification of its own code guided by the metamorphic engine (stored at memory locations from 8 to 14) and

P_0 :	
1: <code>mov l, 100</code>	8: <code>mov MEM[l], MEM[4]</code>
2: <code>input \Rightarrow x</code>	9: <code>mov MEM[l] + 1, MEM[5]</code>
3: <code>if (MEM[x] mod 2) goto 7</code>	10: <code>mov MEM[l] + 2, encode(goto 6)</code>
4: <code>mov y, MEM[x]</code>	11: <code>mov 4, encode(nop)</code>
5: <code>mov x, MEM[x]/2</code>	12: <code>mov 5, encode(goto MEM[l])</code>
6: <code>goto 8</code>	13: <code>mov l, MEM[l] + 3</code>
7: <code>mov x, (MEM[x] + 1)/2</code>	14: <code>goto 2</code>

Figure 3: A metamorphic program P_0

the execution of its intended functionality (stored at memory locations from 2 to 7). Observe that the execution of instruction `mov MEM[l], MEM[4]` at location 8 has the effect of writing at the location indicated by the value stored in l the value stored at location 4 (that in this case is an instruction). Moreover, observe that instruction `mov l, MEM[l] + 3` at location 13 has the effect of increasing by 3 the address contained in location l . Observe that the first time that the metamorphic engine is executed it writes a `nop` at memory location 4 and copies the original content of this location to the free location identified by $MEM[l]$; then it adds some `goto` instructions to preserve the original semantics. The second time it writes a `nop` at memory location 4 and copies the original content of this location, that now is a `nop` to the free location identified by $MEM[l] + 3$, and then adds some `goto` instructions to preserve the original semantics. Thus, the effect of this metamorphic engine is to keep on adding a `nop` before the execution of the instruction originally stored at location 4. To better understand this simple example in Figure 4 we report a particular execution trace of P_0 corresponding to the input stream $\mathcal{J} = 7 :: 6$. The states marked with a black bullet identify the phase boundary of the considered execution and are used to define the phases of the considered execution trace. The example highlights the fact that phase semantics provides a very low level description of code evolution and it identifies a new phase every time that an instruction that will be later executed is modified in memory. This means that when considering a metamorphic engine that operates several instruction modifications for generating the new variant we will end-up by observing a phase for each modification. In this cases it would be nice and useful to further abstract the model of phase semantics in order to observe only the “final” code variant obtained through this series of intermediate modifications. The formal definition of this abstraction will be based on some knowledge of the metamorphic engine that we are considering. In Section 7 we will provide an example of this when considering the metamorphic malware MetaPHOR.

3.3. Fix-point Phase Semantics

It is possible to define a function on $\langle \wp(\mathbb{P}^*), \subseteq \rangle$ that iteratively computes the phase semantics of a given program. In order to do this we introduce the notion of *mutating transition*: A transition between two states that leads to a state which is a phase boundary.

phase-1	•	$\sigma_1 = \langle 1, m_1, \theta, 7 :: 6 \rangle$ $\sigma_2 = \langle 2, m_2 = m_1[l \leftarrow 100], \theta, 7 :: 6 \rangle$ $\sigma_3 = \langle 3, m_3 = m_2[x \leftarrow 7], \theta, 6 \rangle$ $\sigma_4 = \langle 7, m_4 = m_3, \theta, 6 \rangle$ $\sigma_5 = \langle 8, m_5 = m_4[x \leftarrow 4], \theta, 6 \rangle$
phase-2	•	$\sigma_6 = \langle 9, m_6 = m_2[100 \leftarrow \text{encode}(\text{mov } y, \text{MEM}[x])], \theta, 6 \rangle$
phase-3	•	$\sigma_7 = \langle 10, m_7 = m_6[101 \leftarrow \text{encode}(\text{mov } x, \text{MEM}[x]/2)], \theta, 6 \rangle$
phase-4	•	$\sigma_8 = \langle 11, m_8 = m_7[102 \leftarrow \text{encode}(\text{goto } 6)], \theta, 6 \rangle$
phase-5	•	$\sigma_9 = \langle 12, m_9 = m_8[4 \leftarrow \text{encode}(\text{nop})], \theta, 6 \rangle$
phase-6	•	$\sigma_{10} = \langle 13, m_{10} = m_9[5 \leftarrow \text{encode}(\text{goto } 100)], \theta, 6 \rangle$ $\sigma_{11} = \langle 14, m_{11} = m_{10}[l \leftarrow 103], \theta, 6 \rangle$ $\sigma_{12} = \langle 2, m_{12} = m_{11}, \theta, 6 \rangle$ $\sigma_{13} = \langle 3, m_{13} = m_{12}[x \leftarrow 6], \theta, \varepsilon \rangle$ $\sigma_{14} = \langle 4, m_{14} = m_{13}, \theta, \varepsilon \rangle$ $\sigma_{15} = \langle 5, m_{15} = m_{14}, \theta, \varepsilon \rangle$ $\sigma_{16} = \langle 100, m_{16} = m_{14}, \theta, \varepsilon \rangle$ $\sigma_{17} = \langle 101, m_{17} = m_{16}[y \leftarrow 6], \theta, \varepsilon \rangle$ $\sigma_{18} = \langle 102, m_{18} = m_{17}[x \leftarrow 3], \theta, \varepsilon \rangle$ $\sigma_{19} = \langle 6, m_{19} = m_{18}, \theta, \varepsilon \rangle$ $\sigma_{20} = \langle 8, m_{20} = m_{19}, \theta, \varepsilon \rangle$
phase-7	•	$\sigma_{21} = \langle 9, m_{21} = m_{20}[103 \leftarrow \text{encode}(\text{nop})], \theta, \varepsilon \rangle$
phase-8	•	$\sigma_{22} = \langle 10, m_{22} = m_{21}[104 \leftarrow \text{encode}(\text{goto } 100)], \theta, \varepsilon \rangle$
phase-9	•	$\sigma_{23} = \langle 11, m_{23} = m_{22}[105 \leftarrow \text{encode}(\text{goto } 6)], \theta, \varepsilon \rangle$
phase-10	•	$\sigma_{24} = \langle 12, m_{24} = m_{23}[4 \leftarrow \text{encode}(\text{nop})], \theta, \varepsilon \rangle$
phase-11	•	$\sigma_{25} = \langle 13, m_{25} = m_{24}[5 \leftarrow \text{encode}(\text{goto } 103)], \theta, \varepsilon \rangle$ $\sigma_{26} = \langle 14, m_{26} = m_{25}[l \leftarrow 106], \theta, \varepsilon \rangle$ $\sigma_{27} = \langle 2, m_{27} = m_{26}, \theta, \varepsilon \rangle$...

Figure 4: Execution trace of P_0 with input stream $\mathfrak{J} = 7 :: 6$

Definition 3.4. We say that a pair of states (σ_i, σ_j) is a mutating transition of P_0 , denoted $(\sigma_i, \sigma_j) \in MT(P_0)$, if there exists a trace $\sigma = \sigma_0 \dots \sigma_i \sigma_j \dots \sigma_n \in \mathbf{S}[[P_0]]$ such that $\sigma_j \in \text{bound}(\sigma)$.

From the above notion of mutating transition we can derive the program transformer $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \rightarrow \wp(\mathbb{P})$ that associates with each set of programs the set of their possible metamorphic variants, i.e., $P_j \in \mathcal{T}^{Ph}(P_i)$ means that, during execution, program P_i can be transformed into program P_j .

Definition 3.5. The transition relation between programs $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \rightarrow \wp(\mathbb{P})$ is given by the point-wise extension of $\mathcal{T}^{Ph} : \mathbb{P} \rightarrow \wp(\mathbb{P})$ where $\forall P_0 \in \mathbb{P}$:

$$\mathcal{T}^{Ph}(P_0) = \left\{ P_l \mid \begin{array}{l} P_l = (a_l, m_l, \theta_l), \sigma = \sigma_0 \dots \sigma_{l-1} \sigma_l \in \mathbf{S}[[P_0]], \sigma_l = \langle a_l, m_l, \theta_l, \mathfrak{J}_l \rangle, \\ (\sigma_{l-1}, \sigma_l) \in MT(P_0), \forall i \in [0, l-1]: (\sigma_i, \sigma_{i+1}) \notin MT(P_0) \end{array} \right\}$$

and $\mathcal{T}^{Ph}(P_0) = \emptyset$ if P_0 is not self-modifying.

The transition relation \mathcal{T}^{Ph} can be extended to traces of programs by defining function $\mathcal{F}_{\mathcal{T}^{Ph}}[[P_0]] : \wp(\mathbb{P}^*) \rightarrow \wp(\mathbb{P}^*)$ as:

$$\mathcal{F}_{\mathcal{T}^{Ph}}[[P_0]](Z) = P_0 \cup \{z P_i P_{i+1} \mid P_{i+1} \in \mathcal{T}^{Ph}(P_i), z P_i \in Z\}$$

The following lemma shows that $\mathcal{F}_{\mathcal{T}}[[P_0]]$ and $\mathcal{F}_{\mathcal{T}^{Ph}}[[P_0]]$ are extensive functions when considering prefix ordering on strings.

Lemma 3.6. *The following implications hold:*

- (1) *If $P_0 \dots P_l P_{l+1} \in \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket (Z)$ then $P_0 \dots P_l \in Z$ and $P_{l+1} \in \mathcal{T}^{Ph}(P_l)$.*
- (2) *If $\sigma_0 \dots \sigma_n \sigma_{n+1} \in \mathcal{F}_{\mathcal{T}} \llbracket P \rrbracket (X)$ then $\sigma_0 \dots \sigma_n \in X$ and $\sigma_{n+1} \in \mathcal{T}(\sigma_n)$.*
- (3) *If $\sigma_0 \dots \sigma_l \dots \sigma_n \in \mathbf{S} \llbracket P_0 \rrbracket$, where $\sigma_0 = \langle a_0, m_0, \theta_0, \mathcal{J}_0 \rangle$, $\sigma_l = \langle a_l, m_l, \theta_l, \mathcal{J}_l \rangle$, $P_0 = (a_0, m_0, \theta_0)$ and $P_l = (a_l, m_l, \theta_l)$, then $\sigma_l \dots \sigma_n \in \mathbf{S} \llbracket P_l \rrbracket$.*

PROOF: Immediate by definition of $\mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$, $\mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket$ and $\mathbf{S} \llbracket P \rrbracket$.

The following result shows how function $\mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$ iteratively computes the phase semantics of P_0 .

Theorem 3.7. $lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket = \mathbf{S}^{Ph} \llbracket P_0 \rrbracket$.

PROOF: In the rest of the proof we consider that $\forall i \in \mathbb{N}$: $\sigma_i = \langle a_i, m_i, \theta_i, \mathcal{J}_i \rangle$ and $P_i = (a_i, m_i, \theta_i)$. Let us show that $\mathbf{S}^{Ph} \llbracket P_0 \rrbracket \subseteq lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$. We prove this by showing by induction on the length of the string that $\forall z \in \mathbb{P}^* : z \in \mathbf{S}^{Ph} \llbracket P_0 \rrbracket \Rightarrow z \in lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$. The only path of length one in $\mathbf{G} \llbracket P_0 \rrbracket$ is P_0 , and the only string of length one in $lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$ is P_0 . Assume that the above implication holds for every string of length n and let us show that it holds for $z = P_0 \dots P_l P_k$ of length $n + 1$. If $P_0 \dots P_l P_k$ is a path of $\mathbf{G} \llbracket P_0 \rrbracket = (V, E)$ it means that $P_0 \dots P_l$ is a path of $\mathbf{G} \llbracket P_0 \rrbracket = (V, E)$ and that $(P_l, P_k) \in E$. By Definition 3.2 of program evolution graph, this means that $P_0 \dots P_l$ is a path of $\mathbf{G} \llbracket P_0 \rrbracket$ and that there exists $\sigma = \sigma_0 \dots \sigma_{l-1} \sigma_l \dots \sigma_{k-1} \sigma_k \dots \sigma_n \in \mathbf{S} \llbracket P_0 \rrbracket$ such that $\sigma_l \dots \sigma_{k-1} \in \text{phases}(\sigma)$. By Definition 3.1 of phases this means that $\{\sigma_l, \sigma_k\} \subseteq \text{bound}(\sigma)$ and $\forall i \in]l, k - 1[: \sigma_i \notin \text{bound}(\sigma)$. By induction hypothesis and by point (3) of Lemma 3.6 we have that $P_0 \dots P_l \in lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$ and that $\exists \sigma' = \sigma_l \dots \sigma_{k-1} \sigma_k \dots \sigma_n \in \mathbf{S} \llbracket P_l \rrbracket$ such that $\sigma_k \in \text{bound}(\sigma')$ and $\forall i \in]l, k - 1[: \sigma_i \notin \text{bound}(\sigma')$. By Definition 3.4 of mutating transition this implies that $(\sigma_{k-1}, \sigma_k) \in \text{MT}(P_l)$ and $\forall i \in [l, k - 1[: (\sigma_i, \sigma_{i+1}) \notin \text{MT}(P_l)$. By Definition 3.5 of \mathcal{T}^{Ph} this means that $P_k \in \mathcal{T}^{Ph}(P_l)$. From which we can conclude that, $P_0 \dots P_l P_k \in lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$.

Let us now show that $lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket \subseteq \mathbf{S}^{Ph} \llbracket P_0 \rrbracket$. We prove this by showing that $\forall z \in \mathbb{P}^* : z \in lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket \Rightarrow z \in \mathbf{S}^{Ph} \llbracket P_0 \rrbracket$. Consider $P_0 \dots P_l P_{l+1} \in lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$. This means that $\forall i \in [0, l[: P_{i+1} \in \mathcal{T}^{Ph}(P_i)$. By Definition 3.5 of \mathcal{T}^{Ph} , we have that $\forall i \in [0, l[$, there exists $n_i \in \mathbb{N}$ such that $\sigma^i = \sigma_i \sigma_{i_1} \dots \sigma_{i_{n_i}} \sigma_{i+1} \in \mathbf{S} \llbracket P_i \rrbracket$ such that $(\sigma_{i_{n_i}}, \sigma_{i+1}) \in \text{MT}(P_i)$, $\forall k \in [1, n_i[: (\sigma_{i_k}, \sigma_{i_{k+1}}) \notin \text{MT}(P_i)$, and $(\sigma_i, \sigma_{i_1}) \notin \text{MT}(P_i)$. By concatenating all the traces connecting pairs of programs in transition relation we obtain a trace $\sigma = \sigma^0 \sigma^1 \dots \sigma^l = \sigma_0 \sigma_{0_1} \dots \sigma_{0_{n_0}} \sigma_1 \dots \sigma_{l_{n_l}} \sigma_{l+1} \in \mathbf{S} \llbracket P_0 \rrbracket$ such that $\forall i \in [0, l + 1[: \sigma_i \in \text{bound}(\sigma) \wedge \forall k \in [1, n_i[: \sigma_{i_k} \notin \text{bound}(\sigma)$. By Definition 3.1 this means that $\forall i \in [0, l[: \sigma_i \sigma_{i_1} \dots \sigma_{i_{n_i}} \in \text{phases}(\sigma)$. By Definition 3.2 of program evolution graph, this means that $\forall i \in [0, l[: (P_i, P_{i+1}) \in E$, which implies that $P_0 \dots P_l P_{l+1} \in \mathbf{S}^{Ph} \llbracket P_0 \rrbracket$ and concludes the proof.

3.4. Correctness and Completeness of Phase Semantics

We prove the correctness of phase semantics by showing that it is a sound approximation of maximal finite trace semantics, namely by providing a pair of adjoint maps $\alpha_{Ph} : \wp(\Sigma^*) \rightarrow \wp(\mathbb{P}^*)$ and $\gamma_{Ph} : \wp(\mathbb{P}^*) \rightarrow \wp(\Sigma^*)$, for which the fix-point computation of $\mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$ approximates the fix-point computation of $\mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket$.

Given a trace $\sigma = \langle a_0, m_0, \theta_0, \mathcal{J}_0 \rangle \dots \sigma_{i-1} \sigma_i \dots \sigma_n$ we define the abstraction function α_{Ph} as $\alpha_{Ph}(\sigma) = \langle a_0, m_0, \theta_0 \rangle \alpha_{Ph}(\sigma_i \dots \sigma_n)$ such that $\sigma_i \in \text{bound}(\sigma)$ and $\forall l \in [0, i-1] : \sigma_l \notin \text{bound}(\sigma)$, while $\alpha(\epsilon) = \epsilon$. The idea is that abstraction α_{Ph} observes only the states of a trace that are phase boundaries and it can be lifted pointwise to $\wp(\Sigma^*)$ giving rise to the Galois connection $(\wp(\Sigma^*), \alpha_{Ph}, \gamma_{Ph}, \wp(\mathbb{P}^*))$. The following result shows that the fix-point computation of phase semantics approximates the fix-point computation of trace semantics, thus proving that phase semantics is a sound abstraction of trace semantics on the abstract domain $\wp(\mathbb{P}^*)$.

Theorem 3.8. $\forall X \in \wp(\Sigma^*) \alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X)) \subseteq \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket (\alpha_{Ph}(X))$.

PROOF: Let us consider $P_0 \dots P_l P_k \in \alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X))$. Function α_{Ph} is additive, i.e., $\alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X)) = \alpha_{Ph}(X) \cup \alpha_{Ph}(\mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X))$, and we can therefore distinguish the two following cases:

(1): $P_0 \dots P_l P_k \in \alpha_{Ph}(X)$ which immediately implies that the program sequence $P_0 \dots P_l P_k \in \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket (\alpha_{Ph}(X))$.

(2): $P_0 \dots P_l P_k \in \alpha_{Ph}(\mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X))$. In this case we have that there exists $\sigma \sigma_i \sigma_j \in \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X)$ such that $\alpha_{Ph}(\sigma \sigma_i \sigma_j) = P_0 \dots P_l P_k$. By point (2) of Lemma 3.6, this means that $\exists \sigma \sigma_i \in X$ such that $\sigma_j = \mathcal{T}(\sigma_i)$, and $\alpha_{Ph}(\sigma \sigma_i \sigma_j) = P_0 \dots P_l P_k$. We have two possible cases:

(A) $\alpha_{Ph}(\sigma \sigma_i) = P_0 \dots P_l P_k$, which means that $P_0 \dots P_l P_k \in \alpha_{Ph}(X)$ and therefore we are back to case (1).

(B) $\alpha_{Ph}(\sigma \sigma_i) = P_0 \dots P_l$ and $\alpha_{Ph}(\sigma \sigma_i \sigma_j) = P_0 \dots P_l P_k$. This means that $P_0 \dots P_l \in \alpha_{Ph}(X)$ and, by following the definition of α_{Ph} and point (3) of Lemma 3.6, that $\sigma \sigma_i \sigma_j = \sigma_0 \dots \sigma_l \dots \sigma_i \sigma_j \in \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X)$ is such that $\sigma' = \sigma_l \dots \sigma_i \sigma_j \in \mathbf{S} \llbracket P_l \rrbracket$ and $\sigma_j \in \text{bound}(\sigma')$ and $\forall f \in [l+1, i] : \sigma_f \notin \text{bound}(\sigma')$. This means that $P_0 \dots P_l \in \alpha_{Ph}(X)$ and $\sigma' = \sigma_l \dots \sigma_i \sigma_j \in \mathbf{S} \llbracket P_l \rrbracket$ is such that $(\sigma_i, \sigma_j) \in \text{MT}(P_l)$ and $\sigma_j = \langle a_k, m_k, \theta_k, \mathcal{J}_k \rangle$ and $\forall f \in [l, i] : (\sigma_f, \sigma_{f+1}) \notin \text{MT}(P_l)$. By Definition 3.5 of \mathcal{T}^{Ph} , this means that $P_0 \dots P_l \in \alpha_{Ph}(X)$ and $P_k \in \mathcal{T}^{Ph}(P_l)$, where $P_k = \langle a_k, m_k, \theta_k \rangle$. From this we can conclude that $P_0 \dots P_l P_k \in \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket (\alpha_{Ph}(X))$.

Observe that, in general, the converse of the theorem above may not hold, namely we may have that $\alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket (X)) \subset \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket (\alpha_{Ph}(X))$. In fact, given $X \in \wp(\Sigma^*)$, the concrete function $\mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket$ makes only one transition in \mathcal{T} and this may not be a mutating transition, while the abstract function $\mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$ ‘‘jumps’’ directly to the next mutating transition. Even if the fix-point computation of $\mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$ is not step-wise complete [24], it is complete at the fix-point, as shown by the following theorem, proving completeness of the phase semantics.

Theorem 3.9. $\alpha_{Ph}(\text{lfp}^{\subseteq} \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket) = \text{lfp}^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$.

PROOF: For readability in the proof we omit the apex denoting the partial order on which the lfp is computed. In the rest of the proof we consider that for every $i \in \mathbb{N}$: $\sigma_i = \langle a_i, m_i, \theta_i, \mathcal{J}_i \rangle$ and $P_i = \langle a_i, m_i, \theta_i \rangle$.

Let us show that $\alpha_{Ph}(\text{lfp} \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket) \subseteq \text{lfp} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$. We prove this by showing by induction on the length of the string that $\forall z \in \mathbb{P}^* : z \in \alpha_{Ph}(\text{lfp} \mathcal{F}_{\mathcal{T}} \llbracket P_0 \rrbracket) \Rightarrow z \in \text{lfp} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket$. By definition we have that the only string of length 1 that belongs to

$\alpha_{Ph}(\text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket)$ is P_0 , and the only string of length 1 that belongs to $\text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket$ is P_0 . Assume that the above implication holds for any string of length n and consider the string $P_0 \dots P_l P_k$ of length $n + 1$. If $P_0 \dots P_l P_k \in \alpha_{Ph}(\text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket)$ it means that $\exists \sigma = \sigma_0 \dots \sigma_l \dots \sigma_{k-1} \sigma_k \in \text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket$ such that $\alpha_{Ph}(\sigma) = P_0 \dots P_l P_k$, namely that $\exists \sigma = \sigma_0 \dots \sigma_l \dots \sigma_{k-1} \sigma_k \in \mathbf{S}\llbracket P_0 \rrbracket$ such that $\alpha_{Ph}(\sigma) = P_0 \dots P_l P_k$. By induction and by following the definition of α_{Ph} and point (3) of Lemma 3.6, we have that $P_0 \dots P_l \in \text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket$ and $\exists \sigma' = \sigma_l \dots \sigma_{k-1} \sigma_k \in \mathbf{S}\llbracket P_l \rrbracket$ such that $\sigma_k \in \text{bound}(\sigma')$ and $\forall i \in [l + 1, k - 1] : \sigma_i \notin \text{bound}(\sigma')$. This means that $(\sigma_{k-1}, \sigma_k) \in \text{MT}(P_l)$ and $\forall i \in [l, k - 1[: (\sigma_i, \sigma_{i+1}) \notin \text{MT}(P_l)$. By Definition 3.5 of \mathcal{T}^{Ph} we have that $P_k \in \mathcal{T}^{Ph}(P_l)$, from which we can conclude that $P_0 \dots P_l P_k \in \text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket$.

Let us show that $\text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket \subseteq \alpha_{Ph}(\text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket)$. We prove this by showing by induction on the length of the strings that $\forall z \in \mathbb{P}^* : z \in \text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket \Rightarrow z \in \alpha_{Ph}(\text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket)$. For the strings of length 1 we have the same argument as before. Assume that the above implication holds for every string of length n and consider $P_0 \dots P_l P_k$ of length $n + 1$. Assume that $P_0 \dots P_l P_{l+1} \in \text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket$. From point (1) of Lemma 3.6 this means that $P_0 \dots P_l \in \text{lf}p\mathcal{F}_{T^{Ph}}\llbracket P_0 \rrbracket$ and $P_{l+1} \in \mathcal{T}^{Ph}(P_l)$. By induction and from Definition 3.5 of \mathcal{T}^{Ph} we have that $P_0 \dots P_l \in \alpha_{Ph}(\text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket)$ and $\exists \sigma^l = \sigma_l \sigma_{l+1} \dots \sigma_{l_{nl}} \sigma_{l+1} \in \mathbf{S}\llbracket P_l \rrbracket$ such that $(\sigma_{l_{nl}}, \sigma_{l+1}) \in \text{MT}(P_l)$ and $\forall i \in [l, l_{nl}[: (\sigma_{l_i}, \sigma_{l_{i+1}}) \notin \text{MT}(P_l)$ and $(\sigma_l, \sigma_{l+1}) \notin \text{MT}(P_l)$. This means that $\exists \sigma_0 \dots \sigma_l \in \text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket$ such that $\alpha_{Ph}(\sigma_0 \dots \sigma_l) = P_0 \dots P_l$. This implies that there exists a trace $\sigma_0 \dots \sigma_l \dots \sigma_{l+1} \in \text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket$ such that $\alpha_{Ph}(\sigma_0 \dots \sigma_l \dots \sigma_{l+1}) = P_0 \dots P_l P_{l+1}$ and therefore that $P_0 \dots P_l P_{l+1} \in \alpha_{Ph}(\text{lf}p\mathcal{F}_T\llbracket P_0 \rrbracket)$.

4. Abstracting metamorphism

Consider a sequence $P_0 P_1 P_2 \dots \in \mathbf{S}^{Ph}\llbracket P_0 \rrbracket$ in the phase semantics of a metamorphic program P_0 . By definition this means that during execution the program P_0 may evolve into the syntactically different but semantically equivalent program P_1 , then into program P_2 and so on. Thus, a program Q is a metamorphic variant of a program P_0 , denoted $P_0 \rightsquigarrow_{Ph} Q$, if Q is an element of at least one trace in $\mathbf{S}^{Ph}\llbracket P_0 \rrbracket$. This leads to the following *concrete test for metamorphism*:

$$P_0 \rightsquigarrow_{Ph} Q \Leftrightarrow \exists P_0 P_1 \dots P_n \in \mathbf{S}^{Ph}\llbracket P_0 \rrbracket, \exists i \in [0, n] : P_i = Q \quad (1)$$

Thanks to the completeness of phase semantics, the above concrete test for metamorphic variant avoids both false positives and false negatives. Unfortunately, the phase semantics of a metamorphic program may present infinite traces, modeling infinite code evolutions, and this means that it may not be possible to decide whether a program appears (or not) in the phase semantics of a metamorphic program. So phase semantics provides a very precise model of code evolution that leads to a test for metamorphism that is undecidable in general. In order to gain decidability we have to loose precision. In particular, our idea is to abstract the phase semantics in order to obtain an approximated model of code evolution that leads to a decidable abstract test for metamorphism. Indeed, the considered model of metamorphic code behavior is based on a very low-level representation of programs as memory maps that simply give the contents of memory locations together with the address of the instruction to be executed

next. While such a representation is necessary to precisely capture the effects of code self-modification, it is not a convenient representation if we want to statically recognize the different code snapshots encountered during a program's execution. Instead, we would like to consider a suitable abstraction of the domain of programs where it is possible to compute an approximation of the metamorphic behavior of a self-modifying program that can be used to decide whether a program is a metamorphic variant of another one. The idea is to determine an abstract interpretation of phase semantics, namely to approximate the computation of phase semantics on an abstract domain that captures properties of the evolution of the code, rather than of the evolution of program states, as usual in abstract interpretation. Following the standard abstract interpretation approach to static program analysis [12], we have to:

- Define an abstract domain $\langle A, \sqsubseteq_A \rangle$ of code properties that gives rise to a Galois connection $(\wp(\mathbb{P}^*), \alpha_A, \gamma_A, A)$;
- Define the abstract transition relation $\mathcal{T}^A : \wp(A) \rightarrow \wp(A)$ and the abstract function $\mathcal{F}_{\mathcal{T}^A} \llbracket P_0 \rrbracket : A \rightarrow A$ such that $\text{lfp}^{\sqsubseteq_A} \mathcal{F}_{\mathcal{T}^A} \llbracket P_0 \rrbracket = \mathbf{S}^A \llbracket P_0 \rrbracket$ provides an abstract specification of the metamorphic behavior on A ;
- Prove that $\mathbf{S}^A \llbracket P_0 \rrbracket$ is a correct approximation of phase semantics $\mathbf{S}^{Ph} \llbracket P_0 \rrbracket$, namely that $\alpha_A(\text{lfp}^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}} \llbracket P_0 \rrbracket) \sqsubseteq_A \text{lfp}^{\sqsubseteq_A} \mathcal{F}_{\mathcal{T}^A} \llbracket P_0 \rrbracket$.

The abstract specification $\mathbf{S}^A \llbracket P_0 \rrbracket$, obtained as abstract interpretation of the phase semantics, induces an abstract notion of metamorphic variant with respect to the abstract domain A : *A program Q is a metamorphic variant of program P_0 with respect to the abstract domain A , denoted $P_0 \rightsquigarrow_A Q$, if $\mathbf{S}^A \llbracket P_0 \rrbracket$ approximates Q in the abstract domain A .* This leads to the following *abstract test for metamorphism wrt A* :

$$P_0 \rightsquigarrow_A Q \Leftrightarrow \alpha_A(Q) \sqsubseteq_A \mathbf{S}^A \llbracket P_0 \rrbracket$$

In this sense, $\mathbf{S}^A \llbracket P_0 \rrbracket$ is an *abstract metamorphic signature* for P_0 . Whenever $\mathbf{S}^A \llbracket P_0 \rrbracket$ is a correct approximation of phase semantics, we have that $P_0 \rightsquigarrow_A Q$ avoids false negatives, namely the abstract metamorphic test never misses a program that is a metamorphic variant of P_0 . The absence of false positives is not guaranteed in general and it may need a refinement of the abstract domain A [24]. In fact, due to the loss of precision of the abstraction process it may happen that the above test on the abstract metamorphic signature classifies a program as a metamorphic variant of the original malware P_0 even if it is not. For example, if we consider the most abstract domain $A = \{\top\}$ that maps every program and every phase semantics in \top we would have an abstract test for metamorphism that says that every program is a metamorphic variant of another one. Of course this result is sound in the sense that we have no false negatives, while it is very imprecise since we have the maximal amount of false positives. Indeed, there is a wide gamma of abstract domains between the concrete domain of programs (used to compute the precise phase semantics) and the abstract domain $A = \{\top\}$, and we have to carefully choose the abstract domain in order to gain decidability while keeping a good degree of precision. How to choose this abstract domains is a challenging task. Of course the decidability of the abstract test for metamorphism

depends on the choice of the abstract domain A . The example at the end of Section 6 shows another case of loss of precision due to abstraction.

Observe that we are interested in abstract domains representing code properties, namely in abstract domains that need to approximate properties of sequences of instructions. This is an unusual view of abstract domains, which are instead traditionally designed to approximate properties of the states computed by the program. Abstractions for approximating code properties can be achieved naturally by grammar-based, constraint-based and finite state automata abstractions [14]. This allows us to extract, by abstract interpretation, invariant properties of the code evolution in metamorphic code without any a priori knowledge about the obfuscations implemented in the metamorphic engine. This idea will be exploited in the following section, where we propose to abstract each phase by an FSA, describing the sequence of instructions (or approximate descriptions of instructions) that may be disassembled from the corresponding memory. Of course intermediate abstractions can be considered in case we have some partial knowledge on the structure of possible obfuscations implemented by the metamorphic engine. As observed in [16], the knowledge of some aspects of the obfuscation implemented by the metamorphic engine may induce an abstraction on traces which can be composed with the Galois connection identified by the adjoint maps α_{Ph} and γ_{Ph} in order to provide a more abstract basic representation of phases, on which checking whether $P_0 \rightsquigarrow_A Q$ is simpler.

5. Phase Semantics as Sequences of FSA

5.1. Phases as FSA

We introduce a representation of programs, where a program is specified by the sequences of possibly abstract instructions that may occur during its execution. Traditionally, the most commonly used program representation that expresses the possible control flow behaviors of a program, and hence the possible instruction sequences that may be obtained from its execution, is the *control flow graph*. In this representation, the vertices contain the instructions to be executed, and the edges represent possible control flow. For our purposes, it is convenient to consider a dual representation where vertices correspond to program locations and abstract instructions label edges. The resulting representation, which is clearly isomorphic to standard control flow graphs up-to memory locations, is an FSA over an alphabet of instructions. The instructions that define the alphabet of the FSA associated with a phase could be a simplification of ordinary IA-32 instructions, later called *abstract instructions*. Let M_P denote the FSA-representation of a given program P and let $\mathcal{L}(M_P)$ be the language it recognizes. The idea is that for each sequence in $\mathcal{L}(M_P)$ the order of the abstract instructions in the sequence corresponds to the execution order of the corresponding concrete instructions in at least one run of the control flow graph of P . Instructions are abstracted in order to provide a simplified alphabet and to be independent from low-level details that are not relevant when describing the malware metamorphic behaviour. Our construction is parametric in the instruction abstraction. In the rest of the paper, as a possible

abstraction of instructions, we consider function $\iota : \mathbb{I} \rightarrow \mathring{\mathbb{I}}$ defined as follows:

$$\iota(I) = \mathring{I} = \begin{cases} \text{call} & \text{if } I = \text{call } e \\ e_1 & \text{if } I = \text{if } e_1 \text{ goto } e_2 \\ \text{goto} & \text{if } I = \text{goto } e \\ I & \text{otherwise} \end{cases}$$

Observe that since \mathbb{I} is finite also the set $\mathring{\mathbb{I}}$ of abstract instructions is finite. The intuition beyond abstraction ι is to have a program representation that is independent from the particular memory locations used to store instructions, and this is the reason why we abstract from the specific expression denoting the destination of redirection of control flow. Let function $\text{Succ} : \mathbb{I} \times \text{Loc} \rightarrow \wp(\text{Loc})$ denote any sound control flow analysis that determines the locations of the possible successors of an instruction stored at a given location. Hence, $\text{Succ}(I, b)$ approximates the set of locations to which the control may flow after the execution of the instruction I stored at location b . We say that the control flow analysis Succ is sound if for every pair of states $\sigma_i = \langle a_i, m_i, \theta_i, \mathcal{J}_i \rangle$ and $\sigma_j = \langle a_j, m_j, \theta_j, \mathcal{J}_j \rangle$ such that $\sigma_j = \mathcal{T}(\sigma_i)$ then $a_j \in \text{Succ}(\text{decode}(m(a_i)), a_i)$.

Let \mathfrak{F} be the set of FSA over the alphabet $\mathring{\mathbb{I}}$ of abstract instructions where every state is considered to be final. Each FSA in \mathfrak{F} is specified as a graph $M = (Q, E, S)$ where $S \subseteq Q$ is the set of initial states and $E \subseteq Q \times \mathring{\mathbb{I}} \times Q$ is the set of edges labeled by abstract instructions. We define function $\hat{\alpha} : \mathbb{P} \rightarrow \mathfrak{F}$ that associates with each program its corresponding FSA-representation as follows:

$$\hat{\alpha}((a, m, \theta)) = (Q_P, E_P, \{a\})$$

where the set of states is given by $Q_P = \{b \in \text{Loc} \mid \text{decode}(m(b)) \in \mathbb{I}\}$ and the set of edges $E_P \subseteq Q_P \times \mathring{\mathbb{I}} \times Q_P$ is given by:

$$E_P = \left\{ (c, \mathring{I}, d) \mid \begin{array}{l} P = (a, m, \theta); c, d \in Q_P; \\ d \in \text{Succ}(\text{decode}(m(c)), c); \iota(\text{decode}(m(c))) = \mathring{I} \end{array} \right\}$$

As an example, in Figure 5 we show the automaton $\hat{\alpha}(P_0)$ corresponding to program P_0 of Figure 3.

Definition 5.1. We say that $\pi = a_0[\mathring{I}_0] \dots [\mathring{I}_{n-1}] a_n[\mathring{I}_n] a_{n+1}$ is a path of automaton $M = (Q, E, S)$ if $a_0 \in S$ and $\forall i \in [0, n] : (a_i, \mathring{I}_i, a_{i+1}) \in E$. We denote with $\Pi(M)$ the set of paths of M .

By point-wise extension of function $\hat{\alpha}$ we obtain the Galois connection $(\wp(\mathbb{P}), \hat{\alpha}, \hat{\gamma}, \wp(\mathfrak{F}))$.

5.2. Static Approximation of Phase Semantics on $\wp(\mathfrak{F}^*)$

In the previous section we have defined a function $\hat{\alpha} : \mathbb{P} \rightarrow \mathfrak{F}$ that approximates programs as FSA on the alphabet of abstract instructions. In this section, we want to approximate the computation of the phase semantics on the abstract domain of sets of sequences of FSA, i.e., $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$. To this end we define function $\alpha_{\mathfrak{F}} : \mathbb{P}^* \rightarrow \mathfrak{F}^*$ as the extension of $\hat{\alpha} : \mathbb{P} \rightarrow \mathfrak{F}$ to sequences:

$$\alpha_{\mathfrak{F}}(\epsilon) = \epsilon \quad \text{and} \quad \alpha_{\mathfrak{F}}(P_0 P_1 \dots P_n) = \hat{\alpha}(P_0) \alpha_{\mathfrak{F}}(P_1 \dots P_n)$$

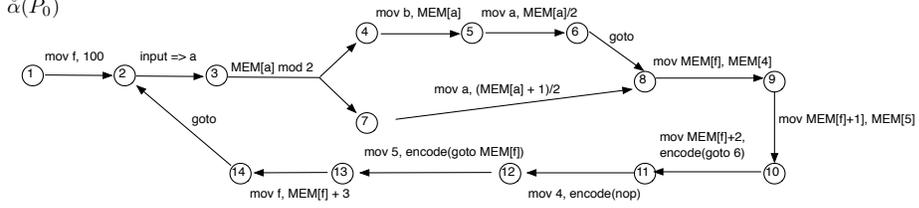


Figure 5: FSA $\hat{\alpha}(P_0)$ corresponding to program P_0 of Figure 3

and then we consider its point-wise extension to $\wp(\mathbb{P}^*)$ that gives rise to the Galois connection $(\wp(\mathbb{P}^*), \alpha_{\mathfrak{F}}, \gamma_{\mathfrak{F}}, \wp(\mathfrak{F}^*))$. In order to compute a correct approximation of the phase semantics on $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$, we need to define an abstract transition relation $\mathcal{T}^{\mathfrak{F}} : \wp(\mathfrak{F}) \rightarrow \wp(\mathfrak{F})$ on FSA that correctly approximates the transition relation on programs $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \rightarrow \wp(\mathbb{P})$, namely such that for every program P in \mathbb{P} we have that $\hat{\alpha}(\mathcal{T}^{Ph}(P)) \subseteq \mathcal{T}^{\mathfrak{F}}(\hat{\alpha}(P))$. We define $\mathcal{T}^{\mathfrak{F}}$ as the best correct approximation of \mathcal{T}^{Ph} on $\wp(\mathfrak{F})$, namely $\mathcal{T}^{\mathfrak{F}} = \hat{\alpha} \circ \mathcal{T}^{Ph} \circ \hat{\gamma}$:

$$\begin{aligned}
\mathcal{T}^{\mathfrak{F}}(K) &= \hat{\alpha}(\mathcal{T}^{Ph}(\hat{\gamma}(K))) \\
&= \{ \hat{\alpha}(P') \mid P' \in \mathcal{T}^{Ph}(P) \wedge P \in \hat{\gamma}(K) \} \\
&= \bigcup_{P \in \hat{\gamma}(K)} \{ \hat{\alpha}(P') \mid P' \in \mathcal{T}^{Ph}(P) \}
\end{aligned}$$

Given the transition relation on FSA we can define $\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0] : \wp(\mathfrak{F}^*) \rightarrow \wp(\mathfrak{F}^*)$ as:

$$\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0](K) = \hat{\alpha}(P_0) \cup \{ kM_i M_{i+1} \mid kM_i \in K, M_{i+1} \in \mathcal{T}^{\mathfrak{F}}(M_i) \}$$

The fix-point computation of $\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0]$, denoted as $\mathbf{S}^{\mathfrak{F}}[P_0] = \text{lfp} \mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0]$, approximates the phase semantics of P_0 on the abstract domain $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$. The correctness of the approximation $\mathbf{S}^{\mathfrak{F}}[P_0]$ follows from the correctness of $\mathcal{T}^{\mathfrak{F}}$ as shown by the following result.

Theorem 5.2. $\alpha_{\mathfrak{F}}(\text{lfp} \mathcal{F}_{\mathcal{T}^{Ph}}[P_0]) \subseteq \text{lfp} \mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0] = \mathbf{S}^{\mathfrak{F}}[P_0]$.

PROOF: Let us prove that $\omega \in \alpha_{\mathfrak{F}}(\text{lfp} \mathcal{F}_{\mathcal{T}^{Ph}}[P_0]) \Rightarrow \omega \in \text{lfp} \mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0]$. Let $M_0 \dots M_n \in \alpha_{\mathfrak{F}}(\text{lfp} \mathcal{F}_{\mathcal{T}^{Ph}}[P_0])$, this means that $\exists P_0 \dots P_n \in \text{lfp} \mathcal{F}_{\mathcal{T}^{Ph}}[P_0]$ such that $\alpha_{\mathfrak{F}}(P_0 \dots P_n) = M_0 \dots M_n$. This means that $\forall i \in [0, n[$: $P_{i+1} \in \mathcal{T}^{Ph}(P_i)$ and $\forall j \in [1, n]$: $\hat{\alpha}(P_j) = M_j$. From the correctness of $\mathcal{T}^{\mathfrak{F}}$ we have that $\forall P \in \mathbb{P}$: $\hat{\alpha}(\mathcal{T}^{Ph}(P)) \subseteq \mathcal{T}^{\mathfrak{F}}(\hat{\alpha}(P))$ and therefore: $\forall i \in [0, n[$: $\hat{\alpha}(P_{i+1}) \in \mathcal{T}^{\mathfrak{F}}(\hat{\alpha}(P_i))$ and $\forall j \in [0, n]$: $\hat{\alpha}(P_j) = M_j$. And this means that $M_0 \dots M_n \in \text{lfp} \mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[P_0]$.

$\mathbf{S}^{\mathfrak{F}}[P_0]$ approximates the phase semantics of program P_0 on $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$ by abstracting programs with FSA, while the transitions, i.e., the metamorphic engine, follow directly from \mathcal{T}^{Ph} and are not approximated. For this reason $\mathbf{S}^{\mathfrak{F}}[P_0]$ may still

have infinite traces of FSAs thus leading to an abstract test of metamorphism that is still undecidable in general. In the following we introduce a static computable approximation of the transition relation on FSA that allows us to obtain a static approximation $\mathbf{S}^\sharp[[P_0]]$ of the phase semantics of P_0 on $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$. $\mathbf{S}^\sharp[[P_0]]$ may play the role of *abstract metamorphic signature* of P_0 . To this end, we introduce the notion of *limits* of a path that approximates the notion of bounds of a trace, and the notion of *transition edge* that approximates the notion of mutating transition. Moreover, we assume to have access to the following sound program analyses:

- a stack analysis $StackVal : Loc \rightarrow \wp(Val)$ that approximates the set of possible values on the top of the stack when control reaches a given location (e.g. [3, 4]);
- a memory analysis $LocVal : Loc \times Loc \rightarrow \wp(Val)$ that approximates the set of possible values that can be stored in a memory location when the control reaches a given location (e.g. [3, 4]).

These analyses allow us to define the function $EVal : Loc \times \mathbb{E} \rightarrow \wp(Val)$, that approximates the evaluation of an expression in a given point:

$$\begin{aligned} EVal(b, n) &= \{n\} \\ EVal(b, MEM[e]) &= \{LocVal(b, l) \mid l \in EVal(b, e)\} \\ EVal(b, MEM[e_1] \text{op} MEM[e_2]) &= \{n_1 \text{op} n_2 \mid i \in \{1, 2\} : n_i \in EVal(b, MEM[e_i])\} \\ EVal(b, MEM[e] \text{op} n) &= \{n_1 \text{op} n \mid n_1 \in EVal(b, MEM[e])\} \end{aligned}$$

and a sound control flow analysis $Succ : \mathbb{I} \times Loc \rightarrow \wp(Loc)$:

$$Succ(I, b) = \begin{cases} EVal(b, e) & \text{if } I \in \{\text{call } e, \text{goto } e\} \\ StackVal(b) & \text{if } I = \text{ret} \\ \{b+1\} \cup EVal(b, e_2) & \text{if } I = \text{if } e_1 \text{ goto } e_2, \\ \emptyset & \text{if } I = \text{halt} \\ \{b+1\} & \text{otherwise} \end{cases}$$

Moreover, function $EVal$ allows us to define function $write : \mathbb{I} \times Loc \rightarrow \wp(Loc)$ that approximates the set of memory locations that may be modified by the execution of an abstract instruction stored at a given location:

$$write(\mathring{I}, b) = \begin{cases} EVal(b, e) & \text{if } \mathring{I} \in \{\text{mov } e, e', \text{input } \Rightarrow e, \text{pop } e\} \\ \emptyset & \text{otherwise} \end{cases}$$

With these assumptions, we define the *limits* of an execution path π as the nodes that are reached by an edge labeled by an abstract instruction that may modify the label of a future edge in π , namely an abstract instruction that occurs later in the same path. Given a path $\pi = a_0[\mathring{I}_0] \dots [\mathring{I}_{n-1}]a_n$ we have:

$$limit(\pi) = \{a_0\} \cup \{a_i \mid write(\mathring{I}_{i-1}, a_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}$$

Definition 5.3. Consider an automata $M = (Q, E, S)$ that represents a program. A pair of program locations (b, c) is a transition edge of $M = (Q, E, S)$, denoted $(b, c) \in TE(M)$, if there exists $a \in S$ and an execution path $\pi \in \Pi(M)$ such that $\pi = a[\mathring{I}_a] \dots [\mathring{I}_{b-1}]b[\mathring{I}_b]c$ and $c \in limit(\pi)$.

```

EXE( $M, \mathring{I}, b$ ) //  $M = (Q, E, S)$  is an FSA
 $Exe = \{M' = (Q, E, S') \mid S' = \{d \mid (b, \mathring{I}, d) \in E\}\}$ 
if  $\mathring{I} = \text{mov } e_1, e_2$ 
  then  $X = \text{write}(\mathring{I}, b)$ 
   $Y = \{n \mid n \in EVal(b, e_2), \text{decode}(n) \in \mathbb{I}\}$ 
   $Exe = Exe \cup \mathbf{NEXT}(X, Y, M, b)$ 
if  $\mathring{I} = \text{input} \Rightarrow e$ 
  then  $X = \text{write}(\mathring{I}, b)$ 
   $Y = \{n \mid n \text{ is an input}, \text{decode}(n) \in \mathbb{I}\}$ 
   $Exe = Exe \cup \mathbf{NEXT}(X, Y, M, b)$ 
if  $\mathring{I} = \text{pop } e$ 
  then  $X = \text{write}(\mathring{I}, b)$ 
   $Y = \{n \mid n \in StackVal(b), \text{decode}(n) \in \mathbb{I}\}$ 
   $Exe = Exe \cup \mathbf{NEXT}(X, Y, M, b)$ 
return  $Exe$ 

NEXT( $X, Y, M, b$ )
 $Next = \emptyset$ 
for each  $a_j \in X$  do
   $\hat{E} = E \setminus \{(a_j, \mathring{I}_j, c) \mid (a_j, \mathring{I}_j, c) \in E\}$ 
   $Next = Next \cup \bigcup_{n \in Y} \left\{ \hat{M} = (\hat{Q}, \hat{E}, \hat{S}) \mid \begin{array}{l} \hat{Q} = Q \cup \{a_j\} \cup \text{Succ}(\text{decode}(n), a_j) \\ \hat{E} = \hat{E} \cup \{(a_j, \iota(\text{decode}(n)), d) \mid d \in \text{Succ}(\text{decode}(n), a_j)\} \\ \hat{S} = \{d \mid (b, \mathring{I}, d) \in E\} \end{array} \right\}$ 
return  $Next$ 

```

Figure 6: Algorithm for statically executing instruction \mathring{I}

In the FSA of Figure 5 there are two transition edges: the one from 11 to 12 and the one from 12 to 13. These edges are labeled with instructions $\text{mov } 4, \text{encode}(\text{nop})$ and $\text{mov } 5, \text{encode}(\text{goto MEM}[f])$, and they both overwrite a location that is reachable in the future. Observe that also the instructions labeling the edges from 8 to 9, from 9 to 10, and from 10 to 11 write instructions in memory, but the locations that store these instructions are not reachable when considering the control flow of P_0 .

In order to statically compute the set of possible FSA evolutions of a given automaton $M = (Q, E, S)$ we need to statically execute the abstract instructions that may modify an FSA. Algorithm **EXE**(M, \mathring{I}, b) in Figure 6 returns the set Exe of all possible FSA that can be obtained by executing the abstract instruction \mathring{I} stored at location b of automaton M . The interesting cases of the algorithm are $\mathring{I} = \text{mov } e_1, e_2$, $\mathring{I} = \text{input} \Rightarrow e$ and $\mathring{I} = \text{pop } e$, since these are the only cases in which we might have a modification of the automaton.

The algorithm starts by initializing Exe to the FSA M' that has the same states and edges of M and whose possible initial states S' are the nodes reachable through the abstract instruction \mathring{I} stored at b in M . This ensures correctness when the execution of the abstract instruction \mathring{I} does not correspond to a real code mutation. Then if \mathring{I} writes in memory, namely if $\mathring{I} \in \{\text{mov } e_1, e_2, \text{input} \Rightarrow e, \text{pop } e\}$, we consider the set X of locations that can be modified by the execution of \mathring{I} , i.e., $X = \text{write}(\mathring{I}, b)$, and the set Y of possible instructions that can be written by \mathring{I} in a location of X . Next, we add to Exe the set **NEXT**(X, Y, M, b) of all possible automata that can be obtained by writing an instruction of Y in a memory location in X . In particular, **NEXT**(X, Y, M, b) is based on the following observation: For each location a_j in X and for each $n \in Y$ we have an automaton $\hat{M} = (\hat{Q}, \hat{E}, \hat{S})$ to add to Exe where \hat{Q} is obtained by adding to Q the lo-

ation a_j and the possible successors of the new instruction written by \mathring{I} at location a_j ; \mathring{E} is obtained from E by deleting all the edges that start from a_j (if any), and by adding for each $n \in Y$ the edges $\{(a_j, \iota_k(\text{decode}(n)), d) \mid d \in \text{Succ}(\text{decode}(n), a_j)\}$, \mathring{S} is given by the set of nodes reachable through \mathring{I} in the original automaton M .

Let $\text{Evol}(M)$ denote the possible evolutions of automaton M , namely the automata that can be obtained by the execution of the abstract instruction labeling the first transition edge of a path of M :

$$\text{Evol}(M) = \left\{ M' \mid \begin{array}{l} a_0[\mathring{I}_0] \dots [\mathring{I}_{l-1}] a_l[\mathring{I}_l] a_{l+1} \in \Pi(M), (a_l, a_{l+1}) \in \text{TE}(M), \\ \forall i \in [0, l]: (a_i, a_{i+1}) \notin \text{TE}(M), M' \in \mathbf{EXE}(M, \mathring{I}_l, a_l) \end{array} \right\}$$

We can now define the static transition $\mathcal{T}^\# : \wp(\mathfrak{F}) \rightarrow \wp(\mathfrak{F})$. The idea is that the possible static successors of an automaton M are all the automata in $\text{Evol}(M)$ together with all the automata M' that are different from M and that can be reached from M through a sequence of successive automata that differ from M only in the entry point. This ensures the correctness of $\mathcal{T}^\#$, i.e., $M_l \in \mathcal{T}^\#(M_0) \Rightarrow M_l \in \mathcal{T}^\#(M_0)$, even if between M_0 and M_l there are spurious transition edges, i.e., transition edges that do not correspond to any mutating transition.

Definition 5.4. Let $M = (Q, E, S)$. $\mathcal{T}^\# : \wp(\mathfrak{F}) \rightarrow \wp(\mathfrak{F})$ is given by the point-wise extension of:

$$\mathcal{T}^\#(M) = \text{Evol}(M) \cup \left\{ M' \mid \begin{array}{l} MM_1 \dots M_k M' : M_1 \in \text{Evol}(M), \forall i \in [1, k]: \\ M_{i+1} \in \text{Evol}(M_i), M' = (Q', E', S') \in \text{Evol}(M_k), \\ (E \neq E' \vee Q \neq Q'), \forall j \in [1, k]: M_j = (Q, E, S_j) \end{array} \right\}$$

Observe that $X \subseteq \text{Loc}$ and $Y \subseteq \text{Val}$ are finite sets and this ensures that the set of automata returned by the algorithm $\mathbf{EXE}(M, \mathring{I}, b)$ is finite and therefore the set of possible successors of any given automata M is finite. The following result shows that $\mathcal{T}^\#$ correctly approximates $\mathcal{T}^\#$.

Lemma 5.5. For any $M \in \mathfrak{F} : \mathcal{T}^\#(M) \subseteq \mathcal{T}^\#(M)$.

PROOF: Let $M_l \in \mathcal{T}^\#(M_0)$. By definition of $\mathcal{T}^\#$ this means $M_l \in \hat{\alpha}(\mathcal{T}^{Ph}(\hat{\gamma}(M_0)))$. Which means that $\exists P_0 = (a_0, m_0, \theta_0), P_l = (a_l, m_l, \theta_l) : \hat{\alpha}(P_0) = M_0$ and $\hat{\alpha}(P_l) = M_l$ such that $P_l \in \mathcal{T}^{Ph}(P_0)$. By definition of \mathcal{T}^{Ph} this means that there exists $P_0 = (a_0, m_0, \theta_0), P_l = (a_l, m_l, \theta_l) : \hat{\alpha}(P_0) = M_0$ and $\hat{\alpha}(P_l) = M_l$ such that $M_0 = (Q_0, E_0, S_0)$, $M_l = (Q_l, E_l, S_l)$ and $E_0 \neq E_l \vee Q_0 \neq Q_l$, moreover there exists $\sigma_0 \dots \sigma_{l-1} \sigma_l \in \mathbf{S}[\![P_0]\!]$ where $(\sigma_{l-1}, \sigma_l) \in \text{MT}(P_0)$ and $\forall i \in [0, l-2] : (\sigma_i, \sigma_{i+1}) \notin \text{MT}(P_0)$ with $\forall i \in [0, l] : \sigma_i = \langle a_i, m_i, \theta_i, \mathcal{I}_i \rangle$. Thanks to the soundness of the analyses this means that: $\exists P_0 = (a_0, m_0, \theta_0), P_l = (a_l, m_l, \theta_l) : \hat{\alpha}(P_0) = M_0$ and $\hat{\alpha}(P_l) = M_l$ such that $M_0 = (Q_0, E_0, S_0)$, $M_l = (Q_l, E_l, S_l)$ and $E_0 \neq E_l \vee Q_0 \neq Q_l$; moreover there exists $\pi = a_0 \xrightarrow{\mathring{I}_0} \dots \xrightarrow{\mathring{I}_{l-2}} a_{l-1} \xrightarrow{\mathring{I}_{l-1}} a_l \in \Pi(M_0)$ where $\forall i \in [0, l]: \iota(\text{decode}(m_i(a_i))) = \mathring{I}_i$ and $(a_{l-1}, a_l) \in \text{TE}(M_0)$. Here we have two possible cases:

- (1) $\forall j \in [0, l-2] : (a_j, a_{j+1}) \notin \text{TE}(M_0)$. In this case from the correctness of *write* and *EVal* it follows that $M_l \in \mathbf{EXE}(M_0, \mathring{I}_{l-1}, a_{l-1})$, and therefore that $M_l \in \text{Evol}(M_0)$ and hence that $M_l \in \mathcal{T}^\#(M_0)$.

- (2) $\exists j \in [0, l-2] : (a_j, a_{j+1}) \in \text{TE}(M_0)$. Observe that every transition edge of M_0 that precedes (a_{l-1}, a_l) in π does not correspond to a real mutating transition and it is therefore spurious, in fact they represent code modifications that seem to be possible because of the loss of information implicit in the static analysis of the code. Let k be the number of transition edges of M_0 that precede the transition edge (a_{l-1}, a_l) in π . Let us denote the set of transition edges of M_0 along the path π up to a_l as follows:

$$\text{TE}(M_0, \pi, a_l) = \{(a_{i_1}, a_{i_1+1}), (a_{i_2}, a_{i_2+1}), \dots, (a_{i_k}, a_{i_k+1}), (a_{l-1}, a_l)\}$$

Let $\pi|_{a_i}$ denote the suffix of path π starting from a_i . Let $M_0 = (Q_0, E_0, S_0)$, and $\forall j \in [1, k]$ let \mathring{I}_{i_j} be the abstract instruction labeling the transition edge (a_{i_j}, a_{i_j+1}) , then from the definition of algorithm **EXE** it follows that:

- $M_{i_1} = (Q_0, E_0, \{a_{i_1+1}\}) \in \mathbf{EXE}(M_0, \mathring{I}_{i_1}, a_{i_1})$ and thus $M_{i_1} \in \text{Evol}(M_0)$, moreover $\text{TE}(M_{i_1}, \pi|_{a_{i_1+1}}, a_l) = \{(a_{i_2}, a_{i_2+1}), \dots, (a_{i_k}, a_{i_k+1}), (a_{l-1}, a_l)\}$;
- $\forall j \in [2, k]$ we have that $M_{i_j} = (Q_0, E_0, \{a_{i_j+1}\}) \in \mathbf{EXE}(M_{i_{j-1}}, \mathring{I}_{i_j}, a_{i_j})$, thus $M_{i_j} \in \text{Evol}(M_{i_{j-1}})$, and $\text{TE}(M_{i_j}, \pi|_{a_{i_j+1}}, a_l) = \{(a_{i_{j+1}}, a_{i_{j+1}+1}), \dots, (a_{i_k}, a_{i_k+1}), (a_{l-1}, a_l)\}$.

In particular, we have that $\text{TE}(M_{i_k}, \pi|_{a_{i_k+1}}, a_l) = \{(a_{l-1}, a_l)\}$. Thanks to the correctness of *write* and *EVal* we have that $M_l \in \mathbf{EXE}(M_{i_k}, \mathring{I}_{l-1}, a_{l-1})$ and therefore $M_l \in \text{Evol}(M_{i_k})$, with $M_l = (Q_l, E_l, S_l)$ and $Q_0 \neq Q_l \vee E_0 \neq E_l$. Thus, we have that $M_0 M_{i_1} \dots M_{i_k} M_l$ is a sequence of FSA such that: $M_1 \in \text{Evol}(M_0)$, $\forall j \in [1, k]$: $M_{i_{j+1}} \in \text{Evol}(M_{i_j})$, $M_l \in \text{Evol}(M_{i_k})$ with $M_l = (Q_l, E_l, S_l)$ and $Q_0 \neq Q_l \vee E_0 \neq E_l$, and $\forall h \in [1, k] : M_{i_h} = (Q_0, E_0, \{a_{i_h+1}\})$ and from the definition of \mathcal{T}^\sharp this means that $M_l \in \mathcal{T}^\sharp(M_0)$.

We can now define function $\mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket : \wp(\mathfrak{F}^*) \rightarrow \wp(\mathfrak{F}^*)$ that statically approximates the iterative computation of phase semantics on the abstract domain $(\wp(\mathfrak{F}^*), \subseteq)$:

$$\mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket (K) = \hat{\alpha}(P_0) \cup \{k M_i M_j \mid (M_i, M_j) \in \mathcal{T}^\sharp, k M_i \in K\}$$

From the correctness of \mathcal{T}^\sharp it follows the correctness of $\mathbf{S}^\sharp \llbracket P_0 \rrbracket = \text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket$, as shown by the following result.

Theorem 5.6. $\alpha_{\mathfrak{F}}(\text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket) \subseteq \text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket$.

PROOF: From Theorem 5.2 we have that $\alpha_{\mathfrak{F}}(\text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket) \subseteq \text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket$. Thus it is enough to prove that $\text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket \subseteq \text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket$. Let $M_0 \dots M_n \in \text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket$. This means that $\forall i \in [0, n]$: $M_{i+1} \in \mathcal{T}^\sharp(M_i)$. From Lemma 5.5 this means that $\forall i \in [0, n]$: $M_{i+1} \in \mathcal{T}^\sharp(M_i)$ and therefore $M_0 \dots M_n \in \text{lfp} \mathcal{F}_{\mathcal{T}^\sharp} \llbracket P_0 \rrbracket$.

Thus, $\mathbf{S}^\sharp \llbracket P_0 \rrbracket$ provides a correct approximation of the phase semantic of a metamorphic program on the abstract domain of set of sequences of FSA, where both programs and transitions are approximated. However, it is still possible to have infinite traces of FSA and thus the abstract test for metamorphism defined by $\mathbf{S}^\sharp \llbracket P_0 \rrbracket$ may still be undecidable. In the next section we propose a further abstraction of $\mathbf{S}^\sharp \llbracket P_0 \rrbracket$ that provides an

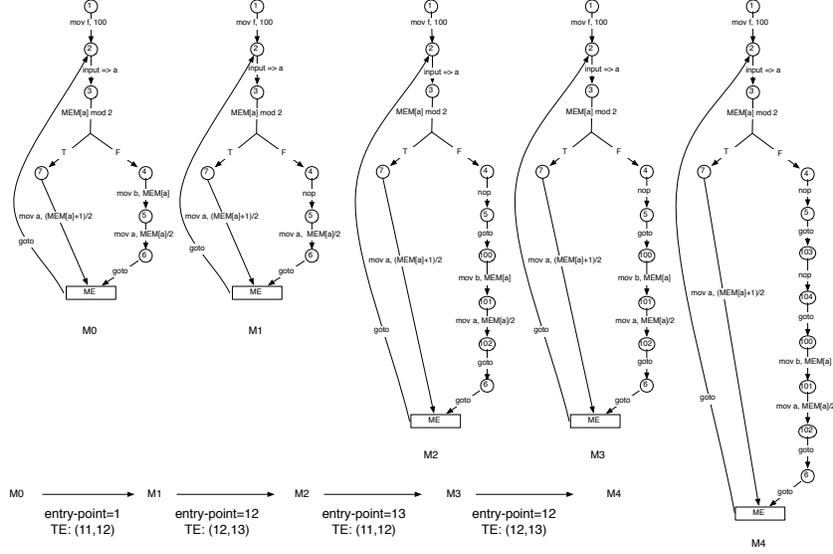


Figure 7: Some metamorphic variants of program P_0 of Figure 3, where the metamorphic engine, namely the instructions stored at locations from 8 to 14, is briefly represented by the box marked ME. In the graphic representation of automata we omit to show the nodes that are not reachable.

approximation of the phase semantics that leads to a decidable test for metamorphism for every metamorphic program.

In Figure 7 we report a possible sequence of FSA that can be generated during the execution of program P_0 of Figure 3. In this case, thanks to the simplicity of the example, it is possible to use the (concrete) transition relation over FSA defined by $\mathcal{T}^{\mathfrak{F}}$.

6. Regular Metamorphism

Regular metamorphism models the metamorphic behavior as a regular language of abstract instructions. This can be achieved by approximating sequences of FSA into a single FSA, denoted $\mathbf{W}[[P_0]]$. $\mathbf{W}[[P_0]]$ represents all possible (regular) program evolutions of P_0 , i.e., it recognizes all the sequences of instructions that correspond to a run of at least one metamorphic variant of P_0 . This abstraction of course is able to precisely model metamorphic engines implemented as FSA of basic code replacement as well as it may provide a regular language-based approximation for any metamorphic engine, by extracting the *regular* invariant of their behavior.

We define an ordering relation on FSA according to the language they recognize: Given two FSA M_1 and M_2 we say that $M_1 \sqsubseteq_{\mathfrak{F}} M_2$ if $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$. Observe that $\sqsubseteq_{\mathfrak{F}}$ is reflexive and transitive but not antisymmetric and it is therefore a pre-order. Moreover, according to this ordering, a unique least upper bound of two automata M_1 and M_2 does not always exist, since there is an infinite number of automata that recognize the language $\mathcal{L}(M_1) \cup \mathcal{L}(M_2)$. Given two automata $M_1 = (Q_1, \delta_1, S_1, F_1, A)$

and $M_2 = (Q_2, \delta_2, S_2, F_2, A)$ on the same alphabet A , we approximate their least upper bound with:

$$M_1 \uplus M_2 = (Q_1 \cup Q_2, \hat{\delta}, S_1 \cup S_2, F_1 \cup F_2, A)$$

where the transition relation $\hat{\delta} : (Q_1 \cup Q_2) \times A \rightarrow \wp(Q_1 \cup Q_2)$ is defined as $\hat{\delta}(q, s) = \delta_1(q, s) \cup \delta_2(q, s)$. FSA are \uplus -closed for finite sets, and the following result shows that \uplus approximates any upper bound with respect to the ordering $\sqsubseteq_{\mathfrak{F}}$.

Lemma 6.1. *Given two FSA M_1 and M_2 we have: $\mathcal{L}(M_1) \cup \mathcal{L}(M_2) \subseteq \mathcal{L}(M_1 \uplus M_2)$.*

PROOF: Let us show by induction on the length of $\omega \in A^*$ that for every $q_1 \in S_1$ it holds that $q \in \delta_1(q_1, \omega) \Rightarrow q \in \hat{\delta}(q_1, \omega)$. Base: When $|\omega| = 1$ we have that $\omega = s \in A$ and therefore $\delta_1(q_1, s) \subseteq \delta_1(q_1, s) \cup \delta_2(q_1, s) = \hat{\delta}(q_1, s)$. Assume that it holds for strings of length n and let us prove that it holds for strings of length $n + 1$. Let $\omega = s_0 \dots s_{n-1} s_n$. By definition we have that:

$$\begin{aligned} \delta_1^*(q_1, s_0 \dots s_{n-1} s_n) &= \bigcup_{p \in \delta_1^*(q_1, s_0 \dots s_{n-1})} \hat{\delta}(p, s_n) \\ \hat{\delta}^*(q_1, s_0 \dots s_{n-1} s_n) &= \bigcup_{p \in \hat{\delta}^*(q_1, s_0 \dots s_{n-1})} \hat{\delta}(p, s_n) \end{aligned}$$

By induction hypothesis $\delta_1^*(q_1, s_0 \dots s_{n-1}) \subseteq \hat{\delta}^*(q_1, s_0 \dots s_{n-1})$, and since $\hat{\delta}(p, s_n) = \delta_1(p, s_n) \cup \delta_2(p, s_n)$, we have $\delta_1^*(q_1, s_0 \dots s_{n-1} s_n) \subseteq \hat{\delta}^*(q_1, s_0 \dots s_{n-1} s_n)$. Moreover the final states of $M_1 \uplus M_2$ are $F_1 \cup F_2$ and therefore: $\delta_1^*(q_1, \omega) \cap F_1 \neq \emptyset \Rightarrow \hat{\delta}^*(q_1, \omega) \cap (F_1 \cup F_2) \neq \emptyset$. The proof that $\forall q_2 \in S_2 : q \in \delta_2(q_2, \omega) \Rightarrow q \in \hat{\delta}(q_2, \omega)$ is analogous.

We can now define the function $\mathcal{F}_{\mathcal{T}^\#}^{\uplus} \llbracket P_0 \rrbracket : \mathfrak{F} \rightarrow \mathfrak{F}$ as follows:

$$\mathcal{F}_{\mathcal{T}^\#}^{\uplus} \llbracket P_0 \rrbracket (M) = \hat{\alpha}(P_0) \uplus M \uplus (\uplus \{M' \mid M' \in \mathcal{T}^\#(M)\})$$

Observe that the set of possible successors of a given automaton M , i.e., $\mathcal{T}^\#(M)$, is finite since we have a (finite family of) successor for every transition edge of M and M has a finite set of edges. Since FSA are \uplus -closed for finite sets, the function $\mathcal{F}_{\mathcal{T}^\#}^{\uplus} \llbracket P_0 \rrbracket$ is well defined and returns an FSA. Let $\wp_F(\mathfrak{F}^*)$ denote the domain of finite sets of strings of FSA, let $K \in \wp_F(\mathfrak{F}^*)$ and let us define the function $\alpha_S : \wp_F(\mathfrak{F}^*) \rightarrow \mathfrak{F}$ as:

$$\begin{aligned} \alpha_S(M_0 \dots M_k) &= \uplus \{M_i \mid 0 \leq i \leq k\} \\ \alpha_S(K) &= \uplus \{\alpha_S(M_0 \dots M_k) \mid M_0 \dots M_k \in K\} \end{aligned}$$

Function α_S is additive and thus it defines a Galois connection $(\wp_F(\mathfrak{F}^*), \alpha_S, \gamma_S, \mathfrak{F})$. The following result shows that, when considering finite sets of sequences of FSA, the abstract function $\mathcal{F}_{\mathcal{T}^\#}^{\uplus} \llbracket P_0 \rrbracket$ correctly approximates function $\mathcal{F}_{\mathcal{T}^\#} \llbracket P_0 \rrbracket$ on \mathfrak{F} .

Theorem 6.2. *For any $K \in \wp_F(\mathfrak{F}^*)$: $\alpha_S(\mathcal{F}_{\mathcal{T}^\#} \llbracket P_0 \rrbracket (K)) \sqsubseteq_{\mathfrak{F}} \mathcal{F}_{\mathcal{T}^\#}^{\uplus} \llbracket P_0 \rrbracket (\alpha_S(K))$.*

PROOF: We prove $\mathcal{L}(\alpha_S(\mathcal{F}_{\mathcal{T}^\#}[[P_0]](K))) \subseteq \mathcal{L}(\mathcal{F}_{\mathcal{T}^\#}^\cup[[P_0]](\alpha_S(K)))$. Let $\dot{I}_0 \dots \dot{I}_n \in \mathcal{L}(\alpha_S(\mathcal{F}_{\mathcal{T}^\#}[[P_0]](K)))$. This means $\exists M_0 \dots M_h \in \mathcal{F}_{\mathcal{T}^\#}[[P_0]](K)$ with $\dot{\alpha}(P_0) = M_0$ s.t. $\dot{I}_0 \dots \dot{I}_n \in \mathcal{L}(\alpha_S(M_0 \dots M_h))$. Let $\forall i \in [0, h] : M_i = (Q_i, E_i, S_i)$. Therefore, by definition of $\mathcal{F}_{\mathcal{T}^\#}[[P_0]]$ and of \cup , we have that $\exists M_0 \dots M_{h-1} \in K$ with $\dot{\alpha}(P_0) = M_0$ and $M_h \in \mathcal{T}^\#(M_{h-1})$ such that $\dot{I}_0 \dots \dot{I}_n \in \mathcal{L}((Q_0 \cup \dots \cup Q_h), (E_0 \cup \dots \cup E_h), (S_0 \cup \dots \cup S_h))$. Since $M_0 \dots M_{h-1} \in K$ we have that $\alpha_S(K) = (Q_K, E_K, S_K)$ where $Q_0 \cup \dots \cup Q_{h-1} \subseteq Q_K$, $E_0 \cup \dots \cup E_{h-1} \subseteq E_K$ and $S_0 \cup \dots \cup S_{h-1} \subseteq S_K$. This implies that $M_h \in \mathcal{T}^\#(\alpha_S(K))$. By definition $\mathcal{F}_{\mathcal{T}^\#}^\cup[[P_0]](\alpha_S(K)) = \dot{\alpha}(P_0) \cup \alpha_S(K) \cup (\cup\{M' \mid M' \in \mathcal{T}^\#(\alpha_S(K))\})$, and therefore we have $\mathcal{F}_{\mathcal{T}^\#}^\cup[[P_0]](\alpha_S(K)) = (Q', E', S')$ where $Q_0 \cup \dots \cup Q_{h-1} \cup Q_h \subseteq Q'$, $E_0 \cup \dots \cup E_{h-1} \cup E_h \subseteq E'$ and $S_0 \cup \dots \cup S_{h-1} \cup S_h \subseteq S'$ and therefore $\dot{I}_0 \dots \dot{I}_n \in \mathcal{L}(\mathcal{F}_{\mathcal{T}^\#}^\cup[[P_0]](\alpha_S(K)))$.

Observe that, thanks to the fact that $\mathcal{T}^\#(M)$ returns a finite number of possible successors of M , then at each step of the fix-point computation of $\mathcal{F}_{\mathcal{T}^\#}$ the function $\mathcal{F}_{\mathcal{T}^\#}$ is applied to a finite set of traces of FSA. This means Theorem 6.2 can be applied to each step of the fix-point computation of $\mathcal{F}_{\mathcal{T}^\#}$.

The domain $\langle \mathfrak{F}, \sqsubseteq_{\mathfrak{F}} \rangle$ has infinite ascending chains, which means that, in general, the fix-point computation of $\mathcal{F}_{\mathcal{T}^\#}^\cup[[P_0]]$ on \mathfrak{F} may not converge. A typical solution for this situation is the use of a widening operator which forces convergence towards an upper approximation of all intermediate computations along the fix-point iteration, i.e., an element in \mathfrak{F} which upper approximates the iterations of the fix-point semantic operator. The widening operator has for example been used to approximate fix-point computations in possibly non-complete lattices, e.g., in the case of convex polyhedra [15]. We refer to the widening operation over FSA described by D'Silva [22]. Here the author considers an increasing sequence $M_0 M_1 \dots M_k$ of FSA in a fix-point computation of a function \mathcal{H} on automata, where $\mathcal{L}(M_{i+1}) = \mathcal{L}(M_i) \cup \mathcal{L}(\mathcal{H}(M_i))$. Given 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 $i < j$, 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 is then given by the quotient of M_j with respect to the partition induced by \equiv_R :

$$M_i \nabla M_j = M_j / \equiv_R$$

By changing the widening seed, i.e., the equivalence relation R , we obtain different widening operators. It has been proved 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 [22]. 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 $\exists r \in Q_i : (r, q) \in R_n$ 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

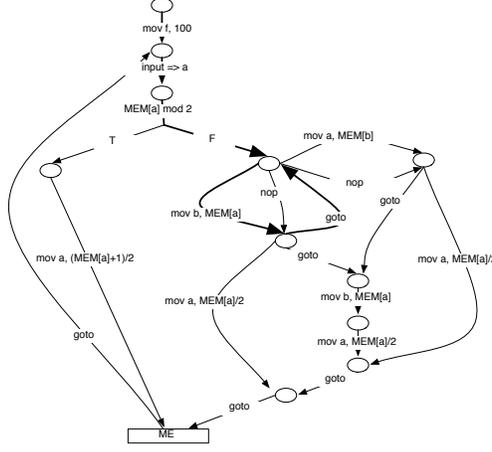


Figure 8: Widened phase semantics

widening. We denote with ∇_n the widening operator that uses R_n as widening seed. ∇_n is well defined since \mathbb{I} is finite.

The widening operator ∇_n allows us to approximate the least fix-point of $\mathcal{F}_{\mathcal{T}^\sharp}^\omega[[P_0]]$ on $\langle \mathfrak{F}_k, \sqsubseteq_{\mathfrak{F}} \rangle$ with the limit $\mathbf{W}[[P_0]]$ of the following widening sequence:

$$W_0 = \hat{\alpha}(P_0) \quad W_{i+1} = W_i \nabla_n \mathcal{F}_{\mathcal{T}^\sharp}^\omega[[P_0]](W_i)$$

In the following we refer to $\mathbf{W}[[P_0]]$ as the *widened fix-point* of $\mathcal{F}_{\mathcal{T}^\sharp}^\omega[[P_0]]$ and to the sequence $W_0 W_1, \dots$ as the *widening sequence* of $\mathcal{F}_{\mathcal{T}^\sharp}^\omega[[P_0]]$, namely the sequence of automata generated in the above widening fix-point computation. From the correctness of the widening operator ∇_n and by Theorem 6.2, it follows that the widening sequence $W_0 W_1 \dots$ converges to an upper-approximation of the least fix-point of $\mathcal{F}_{\mathcal{T}^\sharp}^\omega[[P_0]]$, namely any automata modeling a possible static variant of P_0 is correctly approximated by $\mathbf{W}[[P_0]]$. This means that for every M_i we have that:

$$\dots M_i \dots \in \text{Lfp}^\subseteq \mathcal{F}_{\mathcal{T}^\sharp}^\omega[[P_0]] \Rightarrow M_i \sqsubseteq_{\mathfrak{F}} \mathbf{W}[[P_0]]$$

Therefore, the language $\mathcal{L}(\mathbf{W}[[P_0]])$ recognized by the limit of the widening sequence contains all the possible sequences of abstract instructions that can be executed by a metamorphic variant of the original program P_0 . As a consequence, a program Q is a regular (abstract) metamorphic variant of P_0 if $\mathbf{W}[[P_0]]$ recognizes all the sequences of abstract instructions that correspond to the runs of Q . This leads to the following abstract test for metamorphism:

$$P_0 \rightsquigarrow_{\mathfrak{F}} Q \Leftrightarrow \hat{\alpha}(Q) \sqsubseteq_{\mathfrak{F}} \mathbf{W}[[P_0]] \Leftrightarrow \mathcal{L}(\hat{\alpha}(Q)) \subseteq \mathcal{L}(\mathbf{W}[[P_0]])$$

In this abstract model it is possible to decide if a program is a metamorphic variant of another one since language containment is decidable in FSA. The language

$\mathcal{L}(\mathbf{W}[[P_0]])$ represents the regular metamorphic signature for the metamorphic malware P_0 . On the other side, the automaton $\mathbf{W}[[P_0]]$ represents the mechanism of generation of the metamorphic variants and therefore it provides a model of the metamorphic engine of P_0 . Figure 8 shows the widened fix-point $\mathbf{W}[[P_0]]$ of the widening sequence of program P_0 reported in Figure 3, where the widening seed is R_2 . This automaton recognizes any possible program that can be obtained during the execution of P_0 . Note that, the loss of precision introduced by the abstraction and the widening, may lead to false positives, as for example the sequences of instructions along the bold path `mov f, 100; input \Rightarrow a; MEM[a] mod 2 = 0; mov b, MEM[a]; goto; mov b, MEM[a]; goto; ...` that is not a run of any of the metamorphic variants of P_0 .

7. Case Study: The malware MetaPHOR

In this section we discuss how the proposed approach could be applied to a well-known metamorphic malware called MetaPHOR, also known as Win32/Smile or ETAP. In particular, we will show how the knowledge of some implementation details of the metamorphic engine of MetaPHOR could be used for designing an abstraction of phase semantics that observes only the “final” metamorphic variants of the malware while abstracting from the intermediate phases that lead to the generation of the new code variant. Moreover, at the end of this section we provide a very simple example that shows how the knowledge of specific features of the metamorphic engine can be used for learning an approximated specification of the transformations used by the metamorphic engine. The proposed example is very simple but it provides good insights on how further knowledge of the metamorphic code can be exploited for extracting properties of the metamorphic engine from the phase semantics.

We have decided to focus on MetaPHOR because it is a real metamorphic virus that, according to [5], uses highly advanced metamorphic techniques which combine the majority of the techniques used by its predecessor. The virus, written by the virus writer Mental Driller, was released in the most recent version in early March 2002. The original version of this virus infects only Windows 32-bit files, but a later variant of the virus was a cross-platform infector capable of infecting also Linux ELF files. The Mental Driller named it MetaPHOR from the words “Metamorphic Permutating High-Obfuscating Reassembler”, which accurately describe this virus. A detailed description of the virus can be found at [5, 21]².

Let us describe the main stages of the life-cycle behavior of MetaPHOR. *Polymorphic Decryption*: MetaPHOR starts by running a sophisticated polymorphic decryptor that hides from decryption heuristic scanners. Sometimes this step is not necessary because the virus could be unencrypted, as MetaPHOR is programmed to produce an unencrypted copy every few infections. *Payload*: MetaPHOR performs the intended payload that consists in displaying a specific message according to the current date. *Re-building by metamorphism*: MetaPHOR builds a new virus body in memory at each generation. The metamorphic process is very complex and accounts for around

²The assembly code of the virus can be found at: <http://vx.netlux.org/29a/29a-6/29a-6.602>.

70% of the viral code. *Polymorphism*: MetaPHOR uses advanced polymorphic techniques such as the branching technique, pseudo-random index decryption technique (PRIDE) and entry point obscuring technique (EPO), see [5, 19, 20] for details. *Infection*: MetaPHOR searches for Win32 PE executable files in the current directory and in the directories located in the three levels above the current directory. The virus checks several things before infecting a file: For example it avoids to infect “goat” or “bait” files – files that are created by anti-virus programs.

7.1. The re-building engine of MetaPHOR

An innovative feature of the metamorphic re-building engine of MetaPHOR is the use of an *intermediate representation* which allows to abstract from the complexity of the underlying processors instruction-set and to simplify the metamorphic transformations. In particular, the instruction-set of the intermediate representation includes: base instructions with 2 operands, such as: `add`, `sub`, `or`, `and`, `mov`, ..., base instructions with 1 operand, such as: `push`, `pop`, `call`, ..., other instructions, such as: `ret`, `lea`, ... and macro-instructions that represent for example the instruction sequences which are used when calling a Windows API (such as `apicall-begin`, `apicall-end`), or a system call, or a system call followed by the results saving and so on. Once MetaPHOR has decrypted its own code, it gives control to it and, after the potential payload actions, it re-builds its own code by applying metamorphic transformations. The re-building process is organized as follows:

- *Disassembly-Depermutation*: The x86 code is first disassembled into the intermediate representation presented above. The disassembly algorithm builds the intermediate code in a linear way and this ensures both the depermutation of the resulting code and the removal of the inaccessible code (dead code).
- *Compression*: This step is necessary in order to avoid continuous growth of the viral code that would lead to a virus of many mega-bytes in very few generations. The basic idea of this process is to compress in one instruction what the expansion process codes in many. This is achieved by applying transformations that are the inverse of the ones used by the expansion process. Examples of compressing transformations are:

1. Rules that transform one instruction into an equivalent one, as for example:

- `sub e, imm` \rightarrow `add e, -imm`
- `or e, 0` \rightarrow `nop`

2. Rules that transform a sequence of two instructions into one instruction, as for example:

- `push imm; pop e` \rightarrow `mov e, imm`
- `mov e, imm; push e` \rightarrow `push imm`
- `op e, imm; op e, imm2` \rightarrow `op e, (imm op imm2)`

3. Rules that transform a sequence of three instructions into one instruction, as for example:

- `mov e1, e2; op e1, e3; mov e2, e1` \rightarrow `op e2, e3`

Compression corresponds to rewriting code in such a way that we substitute the first instructions by their equivalent one according to the rule and we overwrite the remaining instructions with `nop`. The compression algorithm compresses the code as much as possible by iterating the code rewriting process. A complete list of the compression rules used by MetaPHOR can be found in [21]. In the following we report an example of what a compressor does in few iterations:

<code>mov [var1], esi</code>	<code>mov eax, esi</code>
<code>push [var1]</code>	<code>nop</code>
<code>pop eax</code>	<code>nop</code>
<code>push ebx</code>	<code>add eax, ebx</code>
<code>pop [var2]</code>	<code>nop</code>
<code>add eax, [var2]</code>	<code>nop</code>

- *Permutation*: The code is permuted by splitting it into blocks of random size. Once the blocks 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. Despite its simplicity, permutation is very powerful since it breaks all the scan strings that can be used to detect the virus.
- *Expansion*: This phase consists in applying, randomly, the inverse of the compression rules. In order to control the growth of the code size, a maximum level of recursion is usually set to 3. This means that every time that an expansion rule is applied to an instruction its recursion level, initially set to 0, is incremented until it reaches the value 3. Moreover, when an instruction uses an immediate value, this can be computationally decomposed into a sequence of operations that compute it. For example, the instruction `mov e1, imm` can be replaced with the following sequence `random ⇒ v1; mov e1, v1; add e1, (imm - v1)`, where instruction `random ⇒ v1` assigns a random value to `v1`. Additionally, the expansion process applies the insertion of dead code, with probability 1/16, after the transformation of each instruction of the compressed code. Examples of dead code inserted by MetaPHOR are instructions that do nothing, like `mov e,e; add e,0; sub e,0; nop`.
- *Reassembly*: The intermediate representation of the expanded, i.e., obfuscated, code is reassembled into the valid x86 assembly language. When several translations are possible, the algorithm chooses one at random. The code is now ready for encryption (polymorphism).

7.2. Applying regular metamorphism to MetaPHOR

Given the description of the life-cycle behavior of MetaPHOR, we want to study the phase semantics of its metamorphic engine. We identify the metamorphic engine of MetaPHOR with the manipulation of the intermediate representation of the code that transforms a MetaPHOR variant into a new one. This means that the metamorphic engine of MetaPHOR consists of the three stages named: *Compression*, *permutation*, and *expansion*; while the other stages (disassembly/reassembly and encryption/decryption)

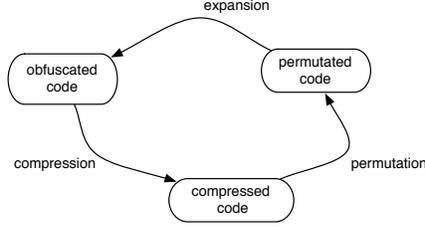


Figure 9: MetaPHOR metamorphic engine behavior

do not deal with metamorphism and will not be considered here. Hence, the metamorphic engine of MetaPHOR takes an obfuscated version of the virus in its intermediate representation and generates a (set of) metamorphic variant in intermediate form. Figure 9 reports the structure of the metamorphic engine of MetaPHOR that will be considered in our analysis.

In the following we assume to have no a priori knowledge about the compression and expansion processes, namely we ignore the set of compression rules used for shrinking code. This information is indeed critical for the effectiveness of MetaPHOR, being the core of its metamorphic engine. Therefore, we assume that the compression rules are secret and the only observable property of MetaPHOR consists in its alternate use of the compression and expansion processes, a feature achievable by tracing its execution. This property of the MetaPHOR behavior makes the virus particularly vulnerable because the compression process shrinks the code into a form that is roughly the same from one generation to another. Indeed, every evolution trace in the phase semantics $\mathbf{S}^{Ph}[[Q]]$ of a given a metamorphic variant Q of MetaPHOR will start by applying several compression rules and obtaining the compressed version C_1 . C_1 clearly has a number of instructions smaller than Q . Of course between Q and C_1 we have many intermediate phases, one for each compression rule applied. Then C_1 will be obfuscated again by applying several expansion rules that lead to the final metamorphic variant Q_1 , whose number of instructions is greater than the one of C_1 . Also in this case between C_1 and Q_1 we have many intermediate phases, possibly one for each expansion rule applied. This process keeps on being repeated generating a sequence of code variants where compressed and final metamorphic variants alternate. Thus, the sequence of phases generated by the execution of a variant Q of MetaPHOR has the following structure:

$$Q \dots C_1 \dots Q_1 \dots C_2 \dots Q_2 \dots C_3 \dots Q_3 \dots C_4 \dots Q_4 \dots$$

where C_1, C_2, \dots denote the compressed code variants, and Q_1, Q_2, \dots the final metamorphic variants of Q .

In the following we show how it is possible to exploit this knowledge of the particular structure of the metamorphic engine of MetaPHOR that alternates code expansion and code compression, in order to design an abstraction of phase semantics that observes only the compressed code variants C_i and the final metamorphic variants Q_i of

the code evolution of MetaPHOR. This shows how the knowledge of some implementation details of the metamorphic engine can be used for abstracting from the intermediate phases that model the intermediate steps of generation of a new metamorphic variant. Given a program $P = (a, m, \theta)$ we consider the code property $\|P\| \in \mathbb{N}$ that observes the number of instructions of P that are different from `nop`:

$$\|P\| = |\{n \mid \text{decode}(m(n)) \in \mathbb{I}, \text{decode}(m(n)) \neq \text{nop}\}|$$

Given a trace $Q \dots C_1 \dots Q_1 \dots C_2 \dots Q_2 \dots$ of the phase semantics $\mathbf{S}^{Ph} \llbracket Q \rrbracket$, we have that for every i the number of `no-nop` instructions decreases at each step between Q_i and C_{i+1} (namely during the compression process), and increases at each step between C_i and Q_i (namely during the permutation and expansion process). This is a property satisfied by all the traces in the phase semantics $\mathbf{S}^{Ph} \llbracket Q \rrbracket$. By knowing this, we can design an abstraction of the phase semantics that observes only the compressed code and the maximal metamorphic variants and ignores the intermediate phases. To this end we define a function $t^b : \mathbb{P}^* \rightarrow \wp(\mathbb{P})$ that extracts the code snapshots with the local minimum and local maximum number of instructions from a trace of programs:

$$\begin{aligned} t^b(\epsilon) &= \emptyset & t^b(P_0 \dots P_n) &= \{P_0, P_n\} \cup \mu(P_0 \dots P_n) \\ \mu(\epsilon) &= \emptyset & \mu(P) &= \emptyset & \mu(QP) &= \emptyset \end{aligned}$$

$$\begin{aligned} \mu(P_0 P_1 P_2 \dots P_n) &= \\ \left\{ \begin{array}{ll} \mu(P_1 P_2 \dots P_n) & \text{if } (\|P_0\| < \|P_1\| < \|P_2\|) \vee (\|P_0\| > \|P_1\| > \|P_2\|) \\ P_1 \cup \mu(P_1 P_2 \dots P_n) & \text{if } (\|P_0\| < \|P_1\| > \|P_2\|) \vee (\|P_0\| > \|P_1\| < \|P_2\|) \end{array} \right. \end{aligned}$$

Function t^b can be lifted point-wise to $\wp(\mathbb{P}^*)$ and this gives rise to the abstraction $\alpha^b : \wp(\mathbb{P}^*) \rightarrow \wp(\mathbb{P})$ and to the Galois connection $(\wp(\mathbb{P}^*), \alpha^b, \gamma^b, \wp(\mathbb{P}))$. Given the particular structure of the metamorphic engine of MetaPHOR we have that, given a metamorphic variant Q of MetaPHOR: $\alpha^b(\mathbf{S}^{Ph} \llbracket Q \rrbracket) = \{Q, C_1, Q_1, C_2, Q_2 \dots Q_n\}$ where C_i are the compressed versions of Q and Q_i are the possible maximal metamorphic variants of Q . Hence, abstraction α^b of the phase semantics allows us to ignore intermediate code evolutions. We apply the above abstraction to the static approximation of the phase semantics as sequences of FSA, namely:

- We define $\alpha_{\mathfrak{F}}^b : \wp(\mathfrak{F}^*) \rightarrow \wp(\mathfrak{F})$ that keeps only the FSA corresponding to the minimal and the maximal code in each trace of FSA: $\alpha_{\mathfrak{F}}^b = \alpha_{\mathfrak{F}} \circ \alpha^b \circ \gamma_{\mathfrak{F}}$.
- Given a metamorphic variant Q of MetaPHOR, compute the static approximation of its phase semantics $\mathbf{S}^\# \llbracket Q \rrbracket$.
- Extract from $\mathbf{S}^\# \llbracket Q \rrbracket$ the set of FSA representing the compressed code and the maximal metamorphic variants of Q : $\alpha_{\mathfrak{F}}^b(\mathbf{S}^\# \llbracket Q \rrbracket) \in \wp(\mathfrak{F})$.
- Fix an order \triangleleft^b on the elements in $\alpha_{\mathfrak{F}}^b(\mathbf{S}^\# \llbracket Q \rrbracket) \in \wp(\mathfrak{F})$ and assume that the corresponding ordered set $\alpha_{\mathfrak{F}}^b(\mathbf{S}^\# \llbracket Q \rrbracket)$ is $\{M_0, M_1 \dots M_n\}$.
- Compute the limit \mathbf{W}^b of the following widening sequence with widening seed R_n :

$$W_0 = M_0 \quad W_{i+1} = W_i \nabla_n W_i \uplus M_i$$

The metamorphic signature \mathbf{W}^b approximates the possible metamorphic variants of the virus MetaPHOR. Indeed, by construction, the language $\mathcal{L}(\mathbf{W}^b)$ contains all the possible sequences of instructions that can be executed by a metamorphic variants of MetaPHOR. Moreover, \mathbf{W}^b is built on the FSAs observed by the abstraction $\alpha_{\mathfrak{S}}^b$ of the abstract phase semantics ($\mathbf{S}^\# \llbracket Q \rrbracket$), namely \mathbf{W}^b is build considering only the final metamorphic variants of Q and not all the intermediate phases. For this reason \mathbf{W}^b provides an abstract signature of MetaPHOR that is more precise than the one that we would obtain by computing the widening on all the steps of the phase semantics (as proposed in Section 6).

It is interesting to observe that while $\mathcal{L}(\mathbf{W}^b)$ models the possible code evolutions, the FSA \mathbf{W}^b describes the mechanism of generation of such variants, namely the rewriting rules used by the engine. The idea is that, by analyzing the multiple paths that connect relevant pairs of nodes in \mathbf{W}^b , i.e., the pairs of nodes which are common between any compressed code $\hat{\alpha}(C_i)$ and \mathbf{W}^b , it is possible to extract the equivalent sequences of instructions generated during the metamorphic behavior, and therefore an approximate description of the rewriting rules. This means that in this case the proposed methodology provides a systematic way for extracting a set of metamorphic transformations that approximates the ones used by MetaPHOR. This is a very important and promising feature of our model, since deriving the code transformations used by metamorphic malware is typically a manual and time consuming task.

In the following we provide a very simple example that hows how it can be possible to exploit the knowledge of some details of the implementation of the metamorphic engine in order to to systematically learn the metamorphic transformations used by abstract interpretation of its phase semantics.

Example. Given the analogy between the intermediate representation of MetaPHOR and our abstract assembly language described in Section 3.1, in the following we use our abstract assembly language to present an example of how the MetaPHOR metamorphic process could modify a simple program and how regular metamorphism allows us to approximate the secret rewriting rules used for its morphing. We consider a very simple program made of only one instruction:

$$P = \text{mov } e, 10$$

We consider the following secret compression rules:

$$\begin{aligned} \text{Rule}_1 &: \text{push } e_2; \text{pop } e_1 \longrightarrow \text{mov } e_1, e_2 \\ \text{Rule}_2 &: \text{mov } e_2, e_1; \text{push } e_2 \longrightarrow \text{push } e_1 \\ \text{Rule}_3 &: \text{pop } e_2; \text{mov } e_1, e_2 \longrightarrow \text{pop } e_1 \end{aligned}$$

Given the simplicity of our example we set the maximal recursion level to 4. It is clear that this simple example only deals with instructions $\text{mov } e_1, e_2$, $\text{push } e$ and $\text{pop } e$. Our proposed framework is parametric on instruction abstraction and in this case we consider the following abstraction $\iota(\text{mov } e_1, e_2) = \text{mov}$, $\iota(\text{push } e_1, e_2) = \text{push}$ and $\iota(\text{pop } e) = \text{pop}$. We have chosen this abstraction in order to be independent from the particular locations used in the expansion process (location renaming can be handled

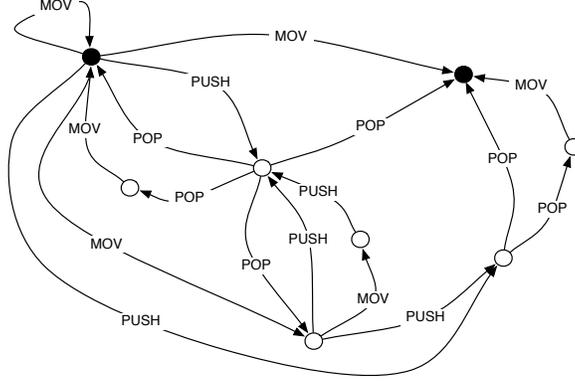


Figure 10: Possible metamorphic variant of $P = \text{mov } l, 10$ generated by the MetaPHOR engine

by using symbolic names). As observed above, from the phase semantics of a program mutated through the MetaPHOR engine we can extract the set of automata representing the possible metamorphic variants through function $\alpha_{\mathcal{S}}^b$. Thus, we consider the automata $\hat{\alpha}(P)$ and the automata representing the possible evolutions of program P that we can obtain by repetitively applying the three rules above to program P and by stopping when the maximal recursion level reaches 4. In order to define the widening sequence we need to fix an order among the possible variants of P . Let $M_0 \dots M_n$ be the ordered sequence of FSA that represent possible evolutions of P , for example we can order the automata with respect to the number of their instructions (choose an order for those evolutions having the same number of instructions). Then we compute the following widening sequence, with widening seed R_2 :

$$W_0 = \hat{\alpha}(P) \quad W_{i+1} = W_i \nabla_2 W_i \uplus M_i$$

The automata \mathbf{W}^b in Figure 10 is built as the limit of the above widening sequence and it recognizes all the possible metamorphic variants of P . As observed above, we can extract an approximated set of rewriting rules from \mathbf{W}^b by considering all paths connecting the common nodes between $\hat{\alpha}(\{\text{mov } e, 10\})$ and \mathbf{W}^b (depicted in black). In this case, a clause of the form $I_1, \dots, I_n \rightarrow \text{mov}$ can be derived from each path with instruction-labels I_1, \dots, I_n connecting black nodes. The following set \mathcal{L} of clauses subsumes all clauses which can be generated in this way from \mathbf{W}^b :

$$\begin{aligned} \text{push; pop} &\rightarrow \text{mov} \\ \text{mov; mov} &\rightarrow \text{mov} \end{aligned}$$

The language generated by the rewriting rules \mathcal{L} includes all the metamorphic variants of mov generated by the rules $\{\text{Rule}_1, \text{Rule}_2, \text{Rule}_3\}$ introduced above, including spurious sequences like mov mov mov mov induced by the widening approximation. This shows how the analysis of the phase semantics of a metamorphic virus can provide important information about the implementation of its metamorphic engine.

8. Related Works

Existing malware detectors typically use signature-based schemes, which are inherently syntactic in nature [34] and that can be easily fooled using simple code obfuscations [10]. In order to face this problem, researchers have recently started to consider semantic approaches to malware detection in order to deal with metamorphism, i.e., obfuscation, (e.g., see [9, 11, 16, 29, 26, 27, 28, 32]).

In [16] the authors use trace semantics to characterize the behaviors of the malware and of the potentially infected program, and use abstract interpretation to “hide” irrelevant aspects of these behaviors. In this setting, a program is infected by a malware if their behaviors are indistinguishable up to a certain abstraction, which corresponds to some obfuscations. A significant limitation of this work, however, is that it does not give any systematic method to derive abstractions from obfuscations. The knowledge of the obfuscation is essential in order to derive these abstractions.

In [27] the authors model the malware as a formula in the new logic CTPL, which is an extension of CTL able to handle register renaming. They develop a model checking algorithm for CTPL and use it to verify infection. A program P is infected by malware M , if P satisfies the CTPL formula that models M . By knowing the obfuscations used by malware M it is possible to design CTPL specifications that recognise several metamorphic variants of M . Once again the knowledge of the obfuscations is essential in order to check infection.

In [11] the idea is to model the malware as a template that expresses its malicious intent, while no distinction is made among their different metamorphic variants. Also in this case the definition of the template is driven by the knowledge of the obfuscations commonly used by malware: It uses symbolic variable/constants to handle variable and register renaming, and it is related to the malware control flow graph in order to deal with code reordering. The authors provide an algorithm that verifies whether a program presents the considered template malicious behavior, and in the case of positive answer they classify the program as infected.

In [28] the authors propose a methodology for making context-sensitive analysis of assembly programs even when the call and ret instructions are obfuscated. In particular, they define a general framework where they formalize the notion of context-trace semantics. They show how, by successive abstractions of the context-trace semantics, it is possible to derive by abstract interpretation the context-sensitive version of any context insensitive analysis that was obtained by approximating the context insensitive trace semantics.

Some researchers have tried to detect metamorphic malware by modeling the metamorphic engine as formal grammars and automata [23, 31, 35]. These works are promising but the design of the grammar and automata is based on the knowledge of the metamorphic transformations used, and none of them provides a methodology for extracting a grammar or an automata from a given metamorphic malware.

In [6] the authors use FSA to approximate programs in a setting that is different from ours. In particular, Beaucamps et al. approximate the set of possible execution traces of a program with a regular language \mathcal{P} on a given alphabet Σ that expresses some instruction properties (like being a system call). In order to be independent from implementation details (and therefore cope with metamorphism) the regular language

\mathcal{P} is abstracted with respect to predefined behavioral patterns. A behavioral pattern is formalized as an automata whose regular language $\mathcal{B}_\lambda \in \wp(\Sigma^*)$ describes the possible sequences in Σ^* that satisfy a high level property λ . The abstraction of \mathcal{P} with respect to \mathcal{B}_λ is obtained by replacing in each trace of \mathcal{P} all the occurrences of the patterns of \mathcal{B}_λ with λ . This abstraction can be carried out with respect to all the behavioral patterns $\mathcal{B}_{\lambda_1} \dots \mathcal{B}_{\lambda_n}$ representing high level properties of interest. This leads to a description of the original program as a regular language of abstract symbols $\Gamma = \{\lambda_1 \dots \lambda_n\}$ that can be compared to the abstract description of known malware in order to detect infection. This detection strategy is able to recognize all the metamorphic variants that are considered by the behavioral patterns. Thus, in order to design efficient behavioral patterns we need to know in advance the metamorphic transformations used by malware. In the experiments in [6] the authors construct the behavioral patterns from the observation of some known malicious execution traces. We believe that it is possible to use our method based on abstracting the phase semantics to systematically derive the FSA for \mathcal{B}_λ .

In [1, 2] the authors model malware as tree automata and then use tree automata inference for capturing the essence of being malicious. Their idea is to focus on the system call behaviour of the malware and to use dynamic analysis in order to extract the data flow dependences between system calls. These dependences are then represented as a graph that provides an abstract model of the malicious behaviour. Next, given a set of these system call dependences graphs, they use tree automata inference to derive (learn) an automata that is able to recognize them. This automata should ideally capture the essence of the malicious behavior and should be able to detect also unknown variants of the considered malware. In our framework we would say that the learned automata acts like a metamorphic signature for the set of malware variants from which it has been generated. However, it cannot be ensured that the so obtained metamorphic signature is able to capture all the possible variants of a given malware. Indeed the efficiency of the automata in capturing unknown malware depends on the malware set used for training the inference algorithm.

All these approaches provide a model of the metamorphic behavior that is based on the knowledge of the metamorphic transformations, i.e., obfuscations, that malware typically use. By knowing how code mutates, it is possible to specify suitable (semantics-based) equivalence relations which trace code evolution and detect malware. 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. These abstract specifications of the malicious behavior are designed based on the obfuscations used by malware and can therefore be potentially bypassed by the design of a variety of new evasion techniques, yet keeping the basic attack model unchanged.

A work closed to our is the one proposed by Guizani et al. [25]. Here the authors propose a theoretical framework for modeling self-modifying programs. They introduce the notion of code waves which are very similar to our phases. The main difference is that their approach for modeling code evolution is dynamic while our one is static. So they can be more precise on the analysis of the considered execution traces but they have to face the problem of behavioral coverage typical of dynamic

analysis.

To the best of our knowledge, we are not aware of any work for statically modeling metamorphism without any a priori knowledge of the transformations used by the metamorphic engine.

Another important work that formally addresses the analysis of self-modifying code is the one of Cai et al. [8]. In this work the authors propose a general framework based on Hoare logic for the modular verification of self-modifying programs. In particular, in order to reason on self-modifying code Cai et al. treat code as data and apply separation logic to support local reasoning in order to verify code modules and then combine them through the frame rule. A related recent work in the verification of self-modifying x86 code based on Hoare logic is the one of Myreen [30]. Myreen proposes a formal semantics for x86 that models possible out-of-date instruction cache and defines a Hoare logic on this semantics that allows to reason about self-modifying code. However, their goals and results are very different from ours: Cai et al. and Myreen propose a general framework based on Hoare logic to verify self-modifying code, while we use program semantics and abstract interpretation to extract metamorphic signature from malicious self-modifying code.

9. Discussion

While traditional static models of programs consider programs that have a constant structure during execution, 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. For this reason we have introduced the notion of phase semantics, that precisely describes the evolution of any metamorphic program during execution. Phase semantics allows us to model the self modifying behavior of any metamorphic malware and it does not need any a priori knowledge of the metamorphic transformations. The key contribution relies upon the idea that abstract interpretation of phase semantics provides useful information about the way code changes, i.e., about the metamorphic engine itself, without any a priori knowledge on its concrete implementation. Phase semantics provides a very precise and low level description of the metamorphic behavior, and due to the presence of possible infinite sequences of code evolution it may not be possible to decide whether a program appears in the phase semantics of another one, namely whether a program is a metamorphic variant of another one. Since we are interested in deciding this fact we recurred to abstract interpretation in order to loose some of the precision of phase semantics in favor of decidability. In particular, the abstract interpretation of the phase semantics of a metamorphic program P on the abstract domain of FSA, i.e., automaton $\mathbf{W}[[P]]$, provides an abstract metamorphic signature that can be used to verify if a program is a metamorphic variant of P . The idea is that the language recognized by the resulting automaton represents the sequences of all possible instructions that can be encountered during the execution of a variant of the original code. A priori knowledge of the transformations applied by the metamorphic engine can be used to further abstract the FSA-representation and to obtain a FSA that models in a more concise way all the possible metamorphic variants.

For example, by abstracting from the intermediate phases that model the intermediate steps of code mutation that lead to the generation of a metamorphic code variant.

Interestingly, the language recognized by $\mathbf{W}[[P]]$ provides an upper-approximation of the possible metamorphic variants of the original malware, while the automaton itself models the mechanism of generation of such variants, i.e., an approximation of the metamorphic engine. This means that with our approach it should be possible to extract properties of the implementation of the metamorphic engine by abstract interpretation of the phase semantics. In Section 7 we have considered the malware MetaPHOR and we have shown how to learn the metamorphic engine from phase semantics on toy example. Besides its simplicity, this example provides good insights on how to proceed for the extraction of properties of code evolution from phase semantics. The idea is that phase semantics captures all the details of code evolution and in order to extract properties of this evolution we need to know what to look at, namely what to observe and what to abstract from. Indeed, we need to design a proper abstraction of code that it is suitable for the property of interest. Any information regarding features of the metamorphic engine can be useful in the design of such an abstraction.

The phase semantics provides a semantics-based model for code metamorphism where abstract interpretation is used for reasoning about code layout. This opens an interesting new field that may represent a future challenge for abstract interpretation: *The abstraction of code layout*, where the code is the object of abstraction and the way it is generated is the object of abstract interpretation. The computation of the phase semantics on these abstract domains returns an approximation of the possible evolutions of the malicious code, where the code variants are approximated with objects of the abstract domain. As future work we are interested in using these approximated code variants in order to extract a formal specification of all possible code variants that could be generated and of the metamorphic engine. The idea is to consider known algorithms for learning from positive examples and to adapt them to the extraction of common features among variants of metamorphic malware. For example we could model the metamorphic engine as a rewriting system, an automata, a grammar or a logic program and then develop a learning strategy for the extraction of upper approximating term rewriting systems, automata and grammars incorporating the provided positive examples yet inducing an approximation of the metamorphic mutation engine. Particular emphasis will be devoted to extract context-free grammars modeling code mutations driven by context free rewriting rules. This involves the definition of widening operators, namely a transformation which steady unstable substrings characterizing exactly the invariant part of the unstable strings of a self-mutating malware. We then plan to validate these learning methodology on real malware.

We believe that the possibility of understanding properties of the implementation of the metamorphic engine, by analyzing the structure of an approximate specification derived by abstract interpretation from the phase semantics of a malware (e.g., the FSA $\mathbf{W}[[P]]$), will become an essential tool for a quick tracking of malicious code and for understanding new obfuscation strategies in the long standing battle between hiding and seeking in computer security.

10. Acknowledgments

The work of Mila Dalla Preda was partially supported by the MIUR project FACE (Formal Avenue for Chasing malwarE).

We would like to thank the anonymous referees for their useful comments that have helped us in improving the paper.

References

- [1] D. Babic, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *Proceedings of the Computer Aided Verification - 23rd International Conference, CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2011.
- [2] D. Babic, D. Reynaud, and D. Song. Recognizing malicious software behaviors with tree automata inference. *Formal Methods in System Design*, 41(1):107–128, 2012.
- [3] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *Proceedings of the 14th International Conference on Compiler Construction (CC'05)*, pages 250–254, 2005.
- [4] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC'04)*, pages 5–23, 2004.
- [5] P. Beaucamps. Advanced metamorphic techniques in computer viruses. 2008. available at VX Heavens: <http://vxheavens.com/lib/apb01.html>.
- [6] P. Beaucamps, I. Gnaedig, and J. Marion. Behavior abstraction in malware analysis. In *Proceedings of Runtime Verification - First International Conference, RV10*, volume 6418 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2010.
- [7] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, 2007.
- [8] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 66–77, New York, NY, USA, 2007. ACM.
- [9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security '03)*, pages 169–186, 2003.
- [10] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 34–44. ACM, 2004.

- [11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, 2005.
- [12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symp. on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, New York, 1977.
- [13] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symp. on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, New York, 1979.
- [14] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 25–28 June 1995. ACM Press, New York, NY.
- [15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
- [16] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5):1–54, 2008.
- [17] M. Dalla Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. Townsend. Modelling metamorphism by abstract interpretation. In *Proceedings of the 17th International Static Analysis Symposium, SAS10*, volume 6337 of *Lecture Notes in Computer Science*, pages 218–235. Springer-Verlag, 2010.
- [18] M. Dalla Preda. The Grand Challenge in Metamorphic Analysis. In *Proceedings of the 6th International Conference on Information Systems, Technology and Management, ICISTM'12*, volume 285 of *Communications in Computer and Information Science*, pages 439–444. Springer, 2012.
- [19] Mental Driller. Advanced polymorphic engine construction. *29A E-zine*, 2000. available at <http://vx.netlux.org/lib/vmd03.html>.
- [20] Mental Driller. Tuareg details and source code. *29A E-zine*, 2000. available at <http://vx.org.ua/29a/29A-5.html>.
- [21] Mental Driller. How i made metaphor and what i've learnt. 2002. available at VX Heavens: <http://vxheavens.com/lib/vmd01.html>.
- [22] V. D'Silva. Widening for automata. Diploma Thesis, Institut Fur Informatick, Universitat Zurich, 2006.

- [23] E. Filiol. Metamorphism, formal grammars and undecidable code mutation. In *Proceedings of World Academy of Science, Engineering and Technology (PWASET)*, volume 20, 2007.
- [24] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.
- [25] W. Guizani, J. Marion, and D. Reynaud-Plantey. Server-side dynamic code analysis. In *Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (MALWARE'09)*, pages 55–62, 2009.
- [26] A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *Proceedings of the 11th International Conference on Computer Aided System Theory (EUROCAST'07)*, volume 4739 of LNCS, pages 497–504, 2007.
- [27] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of LNCS, pages 174–187, 2005.
- [28] A. Lakhota, D. Boccardo, A. Singh, and A. Manacero. Context-sensitive analysis of obfuscated x86 executables. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*, pages 131–140. ACM, 2010.
- [29] G. Lu and S. Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *Proceedings of the Sixth International Conference on Software Security and Reliability, SERE 2012*, pages 31–40. IEEE, 2012.
- [30] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 107–118. ACM, 2010.
- [31] Qozah. Polymorphism and grammars. *29A E-zine*, 2009.
- [32] P. Singh and A. Lakhota. Static verification of worm and virus behaviour in binary executables using model checking. In *Proceedings of the 4th IEEE Information Assurance Workshop*, 2003.
- [33] P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of the Virus Bulletin Conference*, pages 123–144. Virus Bulletin Ltd, 2001.
- [34] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Boston, MA, USA, 2005.
- [35] P. Zbitskiy. Code mutation techniques by means of formal grammars and automata. *Journal in Computer Virology*, 2009.