Modelling Metamorphism by Abstract Interpretation

Mila Dalla Preda[1], Roberto Giacobazzi[1], Saumya Debray[2], Kevin Coogan[2], and Gregg Townsend[2]

[1] Dipartimento di Informatica, Università di Verona
{mila.dallapreda,roberto.giacobazzi}@univr.it
[2] Department of Computer Science, University of Arizona
{debray,kpcoogan,gmt}@cs.arizona.edu

Abstract. Metamorphic malware apply semantics-preserving transformations to their own code in order to foil detection systems based on signature matching. In this paper we consider the problem of automatically extract metamorphic signatures from these malware. 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, and that regular metamorphism can be modelled as finite state automata abstraction of the phase semantics.
Keywords: Abstract interpretation, malware detection, metamorphic code, program transformation, static analysis, security, semantics.

1 Introduction

Challenges and insights. 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* [22]. Malware writers have responded by using a variety of techniques in order to avoid detection: Encryption, oligomorphism with mutational decryptor 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 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* [23], 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 to a syntactically different but semantically equivalent variant in order to avoid detection. 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 [6]. 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 exiting and commercial detectors. The key for identifying metamorphic malware lies, instead, in a deeper understanding of their semantics. Still a major drawback of existing semantics-based methods (e.g., see [13, 19]) relies upon the *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.

Contributions. We proposes a different approach to metamorphic malware detection based on the idea that *extracting metamorphic signatures is approximating malware semantics*. A *metamorphic signature* is therefore any (possibly decidable) approximation of the properties of code evolution. The 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, yet catching properties of this mutation, without any a priori knowledge about the way the metamorphic transformations are implemented. 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. Abstract interpretation is then used 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 a computable 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. 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 [16, 20, 25]. Our approach is general and language independent, providing a systematic method for extracting approximate metamorphic signatures from any metamorphic malware P, in such a way that checking whether a program is a metamorphic variant of P is decidable.

2 Background

Mathematical notation. Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \smallsetminus T$ 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 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) $\perp$ is denoted by $\langle P, \leq, \vee, \wedge, \top, \perp \rangle$. $\sqsubseteq$ denotes pointwise ordering between functions. If $f : S \to T$ and $g : T \to Q$ then $g \circ f : S \to Q$ denotes the composition of f and g, i.e., $g \circ f = \lambda x.g(f(x))$. $f : P \to Q$ on posets is (Scott)-continuous when f preserves lub of countable chains in P. $f : C \to D$ on complete lattices is additive (co-additive) when for any $Y \subseteq C, f(\vee_C Y) = \vee_D f(Y)$ ($f(\wedge_C Y) = \wedge_D f(Y)$). 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 $s_0 \ldots s_h$ is a subsequence of a string $t_0 \ldots t_n$, denoted $s_0 \ldots s_h \preceq t_0 t_1 \ldots t_n$, if $\exists l \in [1, n] : \forall i \in [0, h] : \ s_i = t_{l+i}$.

Finite State Automata (FSA). An FSA M is a tuple (Q, δ, S, F, A), where Q is the set of states, $\delta : Q \times A \to \wp(Q)$ is the transition relation, $S \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states and A is the finite alphabet of symbols. Let $\omega \in A^*$, function $\delta^* : Q \times A^* \to \wp(Q)$ denotes 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 $\mathscr{L}(M)$ 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* $M/\pi = (Q', \delta', S', F', A)$ is defined as follows: $Q' = \{[q]_\pi \mid q \in Q\}$, $\delta' : Q' \times A \to \wp(Q')$ 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 behaviour of a program at different levels of abstraction is an approximation of its (concrete) semantics [8, 9]. The concrete program semantics is computed on the concrete domain $\langle C, \leq_C \rangle$, while approximation is encoded by an abstract domain $\langle A, \leq_A \rangle$. In abstract interpretation abstraction is specified as a Galois connection (GC) (C, α, γ, A) , i.e., an adjunction [8, 9], namely as an abstraction map $\alpha : C \to A$ and a concretization map $\gamma : A \to C$ such that: $\forall a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Let A_1 and A_2 be abstract domains of the concrete domain C: A_1 is more precise than A_2 when $\gamma_2(A_2) \subseteq \gamma_1(A_1)$. Given a GC (C, α, γ, A) and a concrete predicate transformer (semantics) $F : C \to C$, we say that $F^\sharp : A \to A$ is a *sound* approximation of F in A if $\forall c \in C$, $\alpha(F(c)) \leq_A F^\sharp(\alpha(c))$. 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 $F : C \to C$ in A, not all abstract domains induce a complete abstraction [17]. The least fixpoint (lfp) of an operator F on a poset $\langle P, \leq \rangle$, when it exists, is denoted by $lfp^{\leq} F$, or by $lfp F$ when $\leq$ is clear. Any continuous operator $F : C \to C$ on a complete lattice $C = \langle C, \leq_C, \vee_C, \wedge_C, \top_C, \perp_C \rangle$ admits a lfp: $lfp^{\leq_C} F = \bigvee_{n \in \mathbb{N}} F^i(\perp_C)$, where for any $i \in \mathbb{N}$ and $x \in C$: $F^0(x) = x$; $F^{i+1}(x) = F(F^i(x))$. If $F^\sharp : A \to A$ is a correct approximation of $F : C \to C$ on

Syntactic categories:

$n, a \in \mathbb{N}$	(naturals)
$e \in \mathbb{E}$	(expressions)
$I \in \mathbb{I}$	(instructions)
$\mathtt{m} \in \mathcal{M} : \mathbb{N} \to \mathbb{N}_\perp$	(memory map)
$P \in \mathcal{M} \times \mathbb{N} = \mathbb{P}$	(programs)

Expressions:

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

Instructions:

$I ::= \mathtt{call}\ e \mid \mathtt{ret} \mid \mathtt{pop}\ e \mid \mathtt{push}\ e \mid \mathtt{nop} \mid$
$\quad \mathtt{MEM}[e_1] := e_2 \mid \mathtt{input} \Rightarrow \mathtt{MEM}[e] \mid$
$\quad \mathtt{if}\ e_1\ \mathtt{goto}\ e_2 \mid \mathtt{goto}\ e \mid \mathtt{halt}$

Fig. 1. Syntax of an abstract assembly language

$\langle A, \leq_A \rangle$, then $\alpha(lfp^{\leq_C} F) \leq_A lfp^{\leq_A} F^\sharp$. Convergence can be ensured through *widening* iterations along increasing chains [8]. A widening operator $\triangledown : P \times P \to P$ approximates the lub, i.e., $\forall X, Y \in P : X \leq_P (X \triangledown Y)$ and $Y \leq_P (X \triangledown Y)$, and it is such that the increasing chain W^i, where $W^0 = \perp$ and $W^{i+1} = W^i \triangledown F(W^i)$ is not strictly increasing for $\leq_P$. The limit of the sequence W^i provides an upper fixpoint approximation of F on P, i.e., $lfp^{\leq_P} F \leq_P lim_{i \to \infty} W^i$.

3 Modelling metamorphism

Abstract assembly language. Executable programs make no fundamental distinction between code and data. 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 interpreting the result as code and executing it. To model this aspect, we define a program to be a pair $P = (\mathtt{m}, a)$, where $\mathtt{m}$ specifies the contents of a *memory* (both code and data) and a denotes the *entry point* of P, namely the address of the first instruction of P. Since a memory location contains a natural number that can be interpreted either as data or as instruction[3] we use an injective function $\mathtt{encode} : \mathbb{I} \to \mathbb{N}$ that, given an instruction $I \in \mathbb{I}$, returns its binary $\mathtt{encode}(I) \in \mathbb{N}$, and a function $\mathtt{decode} : \mathbb{N} \to \mathbb{I}_\perp$ that given a natural number n returns I if $\mathtt{encode}(I) = n$ otherwise $\perp$. Fig. 1 shows the syntax of our abstract assembly language. The semantics of expressions is specified by a function $\mathcal{E} : \mathbb{E} \times \mathcal{M} \to \mathbb{N}$:

$\mathcal{E}[\![n]\!]\mathtt{m} = n$
$\mathcal{E}[\![\mathtt{MEM}[e]]\!]\mathtt{m} = \mathtt{m}(\mathcal{E}[\![e]\!]\mathtt{m})$
$\mathcal{E}[\![\mathtt{MEM}[e_1]\ \mathtt{op}\ \mathtt{MEM}[e_2]]\!]\mathtt{m} = \mathcal{E}[\![\mathtt{MEM}[e_1]]\!]\mathtt{m}\ \mathtt{op}\ \mathcal{E}[\![\mathtt{MEM}[e_2]]\!]\mathtt{m}$
$\mathcal{E}[\![\mathtt{MEM}[e_1]\ \mathtt{op}\ n]\!]\mathtt{m} = \mathcal{E}[\![\mathtt{MEM}[e_1]]\!]\mathtt{m}\ \mathtt{op}\ n$

and the semantics of instructions by a function $\mathcal{I} : \mathbb{I} \times \Sigma \to \Sigma$:

$\mathcal{I}[\![\mathtt{call}\ e]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I} \rangle = \langle \mathcal{E}[\![e]\!]\mathtt{m}, \mathtt{m}, (a+1) :: \theta, \mathfrak{I} \rangle$
$\mathcal{I}[\![\mathtt{ret}]\!]\langle a, \mathtt{m}, n :: \theta, \mathfrak{I} \rangle = \langle n, \mathtt{m}, \theta, \mathfrak{I} \rangle$
$\mathcal{I}[\![\mathtt{MEM}[e_1] := e_2]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I} \rangle = \langle a+1, \mathtt{m}[\mathcal{E}[\![e_1]\!]\mathtt{m} \leftarrow \mathcal{E}[\![e_2]\!]\mathtt{m}], \theta, \mathfrak{I} \rangle$
$\mathcal{I}[\![\mathtt{input} \Rightarrow \mathtt{MEM}[e]]\!]\langle a, \mathtt{m}, \theta, n :: \mathfrak{I} \rangle = \langle a+1, \mathtt{m}[\mathcal{E}[\![e]\!]\mathtt{m} \leftarrow n], \theta, \mathfrak{I} \rangle$

[3] For simplicity, we assume that each instruction occupies a single location in memory, because the issues raised by variable-length instructions are orthogonal to the topic of this paper, and do not affect any of our results.

$$
\begin{aligned}
&\mathcal{I}[\![\texttt{if}\ e_1\ \texttt{goto}\ e_2]\!]\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle = \begin{cases}\langle \mathcal{E}[\![e_2]\!]\mathtt{m},\mathtt{m},\theta,\mathfrak{I}\rangle & \text{if } \mathcal{E}[\![e_1]\!]\mathtt{m}\neq 0\\ \langle a+1,\mathtt{m},\theta,\mathfrak{I}\rangle & \text{otherwise}\end{cases}\\
&\mathcal{I}[\![\texttt{pop}\ e]\!]\langle a,\mathtt{m},n::\theta,\mathfrak{I}\rangle = \langle a+1,\mathtt{m}[\mathcal{E}[\![e]\!]m \leftarrow n],\theta,\mathfrak{I}\rangle\\
&\mathcal{I}[\![\texttt{goto}\ e]\!]\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle = \langle \mathcal{E}[\![e]\!]\mathtt{m},\mathtt{m},\theta,\mathfrak{I}\rangle\\
&\mathcal{I}[\![\texttt{push}\ e]\!]\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle = \langle a+1,\mathtt{m},\mathcal{E}[\![e]\!]m::\theta,\mathfrak{I}\rangle\\
&\mathcal{I}[\![\texttt{halt}]\!]\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle = \langle \bot,\mathtt{m},\theta,\mathfrak{I}\rangle\\
&\mathcal{I}[\![\texttt{nop}]\!]\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle = \langle a+1,\mathtt{m},\theta,\mathfrak{I}\rangle
\end{aligned}
$$

A program *state* is a tuple $\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle$ where $\mathtt{m}$ is the memory map, a is the address of the next instruction to be executed, $\theta \in \mathbb{N}^*$ is the stack and $\mathfrak{I} \in \mathbb{N}^*$ is the input string. Let $\Sigma = \mathbb{N}_\bot \times \mathcal{M} \times \mathbb{N}^* \times \mathbb{N}^*$ be the set of possible program states and $\mathcal{T} : \wp(\Sigma) \to \wp(\Sigma)$ be the *transition relation* between states, which is given by the point-wise extension of $\mathcal{T}(\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle) = \mathcal{I}[\![\texttt{decode}(\mathtt{m}(a))]\!]\langle a,\mathtt{m},\theta,\mathfrak{I}\rangle$. As usual [11], the *maximal finite trace semantics* $\mathbf{S}[\![P]\!] \in \wp(\Sigma^*)$ of a program $P = (\mathtt{m}, a)$ is given by the least fixpoint of $\mathcal{F}_\mathcal{T}[\![P]\!] : \wp(\Sigma^*) \to \wp(\Sigma^*)$ where $\mathit{Init}[\![P]\!] = \{\langle a,\mathtt{m},\epsilon,\mathfrak{I}\rangle \mid \mathfrak{I} \text{ is an input stream}\}$ and $\mathcal{F}_\mathcal{T}[\![P]\!](X) = \mathit{Init}[\![P]\!] \cup \{\sigma\sigma_i\sigma_j \mid \sigma_j \in \mathcal{T}(\sigma_i), \sigma\sigma_i \in X\}$.

Phase Semantics. Intuitively, 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 $\sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i\rangle$ 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 consider the set $\mathit{mod}(\sigma_i) \subseteq \mathbb{N}$ of memory locations that are modified by the instruction executed in state σ_i:

$$
\mathit{mod}(\sigma_i) = \begin{cases}\{\mathcal{E}[\![e_1]\!]\mathtt{m}\} & \text{if } \texttt{decode}(\mathtt{m}_i(a_i)) = \texttt{MEM}[e_1] := e_2\\ \{\mathcal{E}[\![e]\!]\mathtt{m}\} & \text{if } \texttt{decode}(\mathtt{m}_i(a_i)) \in \{\texttt{input} \Rightarrow \texttt{MEM}[e], \texttt{pop}\ \ e\}\\ \emptyset & \text{otherwise}\end{cases}
$$

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

Definition 1 *The set of phase boundaries of $\sigma = \sigma_0 \dots \sigma_n \in \Sigma^*$, where $\forall i \in [0, n]$: $\sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i\rangle$, is: $\mathit{bound}(\sigma) = \{\sigma_0\} \cup \{\sigma_i \mid \mathit{mod}(\sigma_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}$. The set of phases of a trace $\sigma \in \Sigma^*$ is:*

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

Observe that, by definition, the memory map of the first state of a phase always specifies the code snapshot that is executed in the same phase. Hence, the sequence of the initial states of the phases of a trace highlights the different code snapshots encountered during code execution. In general, different executions of a program give rise to different sequences of code snapshots. A complete characterization of all code snapshots of a self-modifying program can be obtained by organizing phases in a *program evolution graph*. Here, each vertex is a code snapshot P_i corresponding to a phase, and an edge $P_i \to P_j$ 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 2 *The program evolution graph of a program P_0 is $\mathbf{G}[\![P_0]\!] = (V, E)$:*

$$V = \{P_i = (\mathtt{m}_i, a_i) \mid \sigma = \sigma_0..\sigma_i..\sigma_n \in \mathbf{S}[\![P_0]\!] : \sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i\rangle \in \mathit{bound}(\sigma)\}$$

$$E = \left\{ (P_i, P_j) \,\middle|\, \begin{array}{l} P_i = (\mathtt{m}_i, a_i), P_j = (\mathtt{m}_j, a_j), \sigma = \sigma_0..\sigma_i..\sigma_{j-1}\sigma_j..\sigma_n \in \mathbf{S}[\![P_0]\!] : \\ \sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i\rangle, \sigma_j = \langle a_j, \mathtt{m}_j, \theta_j, \mathfrak{I}_j\rangle, \sigma_i \ldots \sigma_{j-1} \in \mathit{phases}(\sigma) \end{array} \right\}$$

A path in $\mathbf{G}[\![P_0]\!]$ is therefore a sequence of programs $P_0 \ldots P_n$ such that for every $i \in [0, n[$ we have that $(P_i, P_{i+1}) \in E$. 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 $\mathbf{S}^{Ph}[\![P_0]\!]$: $\mathbf{S}^{Ph}[\![P_0]\!] = \{P_0 \ldots P_n \mid P_0 \ldots P_n \text{ is a path in } \mathbf{G}[\![P_0]\!]\}$.

```
P0   1: MEM[f] := 100                     8: MEM[MEM[f]] := MEM[4]
     2: input ⇒ MEM[a]                    9: MEM[MEM[f] + 1] := MEM[5]
     3: if (MEM[a] mod 2) goto 7         10: MEM[MEM[f] + 2] := encode(goto 6)
     4: MEM[b] := MEM[a]                 11: MEM[4] := encode(nop)
     5: MEM[a] := MEM[a]/2               12: MEM[5] := encode(goto MEM[f])
     6: goto 8                           13: MEM[f] := MEM[f] + 3
     7: MEM[a] := (MEM[a] + 1)/2         14: goto 2
```

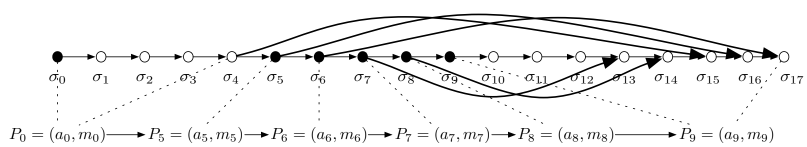


Fig. 2. A metamorphic program P_0 and the phases of one of its traces.

Consider for instance the metamorphic program P_0 of Fig. 2. The metamorphic engine of P_0, which is stored at memory locations from 8 to 13, writes a `nop` at memory location 4 and copies the original content of this location to the free location identified by $\mathtt{MEM}[f]$; then it adds some `goto` instructions to preserve the original semantics. We consider the execution trace $\sigma = \sigma_0\sigma_1 \ldots \sigma_{17}$ of program P_0 corresponding to the input sequence $\mathfrak{I} = 7 :: 6$, in particular $\sigma = \langle 1, \mathtt{m}_0, \epsilon, 7 :: 6\rangle\langle 2, \mathtt{m}_1 = \mathtt{m}_0[f \leftarrow 100], \epsilon, 7 :: 6\rangle\langle 3, \mathtt{m}_2 = \mathtt{m}_1[a \leftarrow 7], \epsilon, 6\rangle\langle 7, \mathtt{m}_3 = \mathtt{m}_2, \epsilon, 6\rangle\langle 8, \mathtt{m}_4 = \mathtt{m}_3[a \leftarrow 4], \epsilon, 6\rangle\langle 9, \mathtt{m}_5 = \mathtt{m}_4[100 \leftarrow \mathtt{encode}(\mathtt{MEM}[b] := \mathtt{MEM}[a])], \epsilon, 6\rangle \ldots \langle 17, \mathtt{m}_{17} = \mathtt{m}_{16}[a \leftarrow 3], \epsilon, \epsilon\rangle$. Fig. 2 shows the considered execution trace σ where: the bold arrows denote the modifications of instructions that will be later executed, for example the bold arrow from $\sigma_4 = \langle a_4, \mathtt{m}_4, \theta_4, \mathfrak{I}_4\rangle$ to $\sigma_{15} = \langle a_{15}, \mathtt{m}_{15}, \theta_{15}, \mathfrak{I}_{15}\rangle$ means that location a_{15} is overwritten by the execution of instruction $\mathtt{decode}(\mathtt{m}_4(a_4))$ at state σ_4, i.e., $a_{15} \in \mathit{mod}(\sigma_4)$; and the black dots identify the states that are phase boundaries.

Fixpoint phase semantics. We introduce the notion of *mutating transition*, i.e., a transition between two states that leads to a state which is a phase boundary. We say that a pair of states (σ_i, σ_j) is a mutating transition of P_0, denoted $(\sigma_i, \sigma_j) \in \mathtt{MT}(P_0)$, if there exists a trace $\sigma = \sigma_0 \ldots \sigma_i\sigma_j \ldots \sigma_n \in \mathbf{S}[\![P_0]\!]$ such that $\sigma_j \in \mathit{bound}(\sigma)$. This allows us to define the code transformer $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ that associates with each set of programs the set of their possible metamorphic variants: $P_j \in \mathcal{T}^{Ph}(P_i)$ means that during execution program P_i can be transformed into program P_j.

Definition 3 $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ *is given by the point-wise extension of:*

$$\mathcal{T}^{Ph}(P_0) = \left\{ P_l \;\middle|\; \begin{array}{l} P_l = (m_l, a_l), \sigma = \sigma_0 \ldots \sigma_{l-1}\sigma_l \in \mathbf{S}[\![P_0]\!], \sigma_l = \langle a_l, \mathtt{m}_l, \theta_l, \mathfrak{I}_l \rangle, \\ (\sigma_{l-1}, \sigma_l) \in \mathit{MT}(P_0), \forall i \in [0, l-1[: (\sigma_i, \sigma_{i+1}) \notin \mathit{MT}(P_0) \end{array} \right\}$$

$\mathcal{T}^{Ph}$ can be extended to traces $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!] : \wp(\mathbb{P}^*) \to \wp(\mathbb{P}^*)$ as: $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!](Z) = P_0 \cup \{zP_iP_j \mid P_j \in \mathcal{T}^{Ph}(P_i), zP_i \in Z\}$.

Theorem 1 $\mathit{lfp}^{\subseteq}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!] = \mathbf{S}^{Ph}[\![P_0]\!]$.

A program Q is a metamorphic variant of a program P_0, denoted $P_0 \leadsto_{Ph} Q$, if Q is an element of at least one sequence in $\mathbf{S}^{Ph}[\![P_0]\!]$.

Correctness and completeness of phase semantics. We prove the correctness of phase semantics by showing that it is a sound approximation of trace semantics, namely by providing a pair of adjoint maps $\alpha_{Ph} : \wp(\Sigma^*) \to \wp(\mathbb{P}^*)$ and $\gamma_{Ph} : \wp(\mathbb{P}^*) \to \wp(\Sigma^*)$, for which the fixpoint computation of $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$ approximates the fixpoint computation of $\mathcal{F}_{\mathcal{T}}[\![P_0]\!]$. Given $\sigma = \langle a_0, \mathtt{m}_0, \theta_0, \mathfrak{I}_0 \rangle \ldots \sigma_{i-1}\sigma_i \ldots \sigma_n$ we define α_{Ph} as:

$\alpha_{Ph}(\sigma) = (\mathtt{m}_0, a_0)\alpha_{Ph}(\sigma_i \ldots \sigma_n)$ s.t. $\sigma_i \in \mathit{bound}(\sigma), \forall l \in [0, i-1] : \sigma_l \notin \mathit{bound}(\sigma)$

Abstraction α_{Ph} observes only the states of a trace that are phase boundaries and it can be lifted point-wise to $\wp(\Sigma^*)$ giving rise to the GC $(\wp(\Sigma^*), \alpha_{Ph}, \gamma_{Ph}, \wp(\mathbb{P}^*))$. The following result shows the correctness of the phase semantics.

Theorem 2 $\forall X \in \wp(\Sigma^*) : \alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}}[\![P_0]\!](X)) \subseteq \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!](\alpha_{Ph}(X))$.

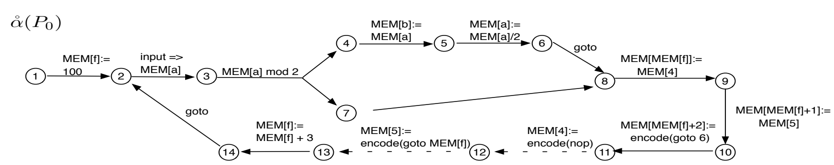
The converse may not hold: $\alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}}[\![P_0]\!](X)) \subset \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!](\alpha_{Ph}(X))$. In fact, given $X \in \wp(\Sigma^*)$, the concrete function $\mathcal{F}_{\mathcal{T}}[\![P_0]\!]$ makes only one transition in $\mathcal{T}$ and this may not be a mutating transition, while the abstract function $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$ *jumps* directly to the next mutating transition. Even if the fixpoint of $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$ is not step-wise complete, it is complete at the fixpoint, as shown by the following theorem.

Theorem 3 $\alpha_{Ph}(\mathit{lfp}^{\subseteq}\mathcal{F}_{\mathcal{T}}[\![P_0]\!]) = \mathit{lfp}^{\subseteq}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$.

4 Abstracting metamorphism

Our model of metamorphic code behaviour 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 analyze the different code snapshots encountered during a program's execution. Our idea is to design 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. We have to: (1) Define an abstract domain $\langle A, \sqsubseteq_A \rangle$ of *code properties* such that $(\wp(\mathbb{P}^*), \alpha_A, \gamma_A, A)$; (2) Define the abstract transition $\mathcal{T}^A : \wp(A) \to \wp(A)$ and $\mathcal{F}_{\mathcal{T}^A}[\![P_0]\!] : A \to A$ such that $lfp^{\sqsubseteq_A} \mathcal{F}_{\mathcal{T}^A}[\![P_0]\!] = \mathbf{S}^A[\![P_0]\!]$; (3) Prove that $\mathbf{S}^A[\![P_0]\!]$ is a correct approximation of phase semantics $\mathbf{S}^{Ph}[\![P_0]\!]$, i.e., $\alpha_A(lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \sqsubseteq_A \mathbf{S}^A[\![P_0]\!]$. This proves that $\mathbf{S}^A[\![P_0]\!]$ is such that *a program Q is a metamorphic variant of program P_0 with respect to A, denoted $P_0 \rightsquigarrow_A Q$, if $\mathbf{S}^A[\![P_0]\!]$ approximates Q in the abstract domain A*: $P_0 \rightsquigarrow_A Q \quad \Leftrightarrow \quad \alpha_A(Q) \sqsubseteq_A \mathbf{S}^A[\![P_0]\!]$. In this sense, $\mathbf{S}^A[\![P_0]\!]$ is an *abstract metamorphic signature* for P_0. Abstract domains for code properties need to approximate properties of sequences of instructions. This can be achieved naturally by grammar-based, constraint-based and finite state automata abstractions. In the following we propose to abstract programs by a FSA describing the sequence of (possibly abstract) instructions that may be disassembled from the given memory.

Phases as FSA. The most commonly used program representation 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. Let M_P denote the FSA-representation of a given program P and let $\mathscr{L}(M_P)$ be the language it recognizes. The idea is that for each sequence in $\mathscr{L}(M_P)$ the order of the 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. In the rest of the paper, for the sake of simplicity, we consider function $\iota : \mathbb{I} \to \mathring{\mathbb{I}}$ defined in Fig. 3. Let $\rho : \mathbb{I} \times \mathbb{N} \to \wp(\mathbb{N})$ denote any sound control flow analysis that



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

Edges$(P = (\mathtt{m}, a), Q_P, \rho)$
$E_P = \emptyset$
while $Q_P \neq \emptyset$
  select $b \in Q_P$ and $Q_P = Q_P \smallsetminus \{b\}$
  $I = \mathtt{decode}(\mathtt{m}(b))$
  for each $c \in \rho(I, b) \cap Q_P$
    $E_P = E_P \cup \{(b, \iota(I), c)\}$
return E_P

Fig. 3. FSA $\mathring{\alpha}(P_0)$ corresponding to program P_0 of Fig. 2, instruction abstraction $\iota : \mathbb{I} \to \mathring{\mathbb{I}}$ and the algorithm that computes E_P

determines the possible successors of a given instruction at a given location, namely $\rho(I, b)$ associates with instruction I stored at memory location b the set of locations of its possible successors. 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 $M = (Q, E, S)$. We define function $\mathring{\alpha} : \mathbb{P} \to \mathfrak{F}$ that associates with each program $P = (\mathtt{m}, a)$ its corresponding FSA-representation as follows: $\mathring{\alpha}(P) = (Q_P, E_P, \{a\})$ where $Q_P = \{b \mid \mathtt{decode}(\mathtt{m}(b)) \in \mathbb{I}\}$ is the set of locations that store an instruction of P, and the set of edges $E_P \subseteq Q_P \times \mathring{\mathbb{I}} \times Q_P$ is computed by the algorithm **Edges** in Fig. 3. This algorithm, given $P = (m, a)$, starts by initializing E_P to the empty set and then for every memory location b that stores an instruction I it adds an edge labeled with $\iota(I)$, whose source is the location b and whose destinations are the locations in $\rho(I, b)$. As an example, at the top of Fig. 3 we show the automaton $\mathring{\alpha}(P_0)$ corresponding to program P_0 of Fig. 2. We say that $\pi = a_0[\mathring{I}_0] \ldots [\mathring{I}_{n-1}]a_n[\mathring{I}_n]a_{n+1}$ is a *path* of automaton $M = (Q, E, S)$, denoted $\pi \in \Pi(M)$, if $a_0 \in S$ and $\forall i \in [0, n[: (a_i, \mathring{I}_i, a_{i+1}) \in E$. Observe that even if the alphabet $\mathring{\mathbb{I}}$ is unbounded (due to the unlimited number of possible expressions), the FSA-representation of every program uses only a finite subset of alphabet $\mathring{\mathbb{I}}$. By point-wise extension of function $\mathring{\alpha}$ we obtain the GC $(\wp(\mathbb{P}), \mathring{\alpha}, \mathring{\gamma}, \wp(\mathfrak{F}))$. Note that abstraction ι defined above makes the FSA-representation of programs independent (up to renaming) from program position.

Theorem 4 *If P_1 and P_2 differ only in their memory position then $\mathring{\alpha}(P_1)$ and $\mathring{\alpha}(P_2)$ are equivalent up to address renaming.*

Abstract phase semantics as traces of FSA. Let $\alpha_{\mathfrak{F}} : \mathbb{P}^* \to \mathfrak{F}^*$ be the extension of $\mathring{\alpha} : \mathbb{P} \to \mathfrak{F}$ to sequences: $\alpha_{\mathfrak{F}}(\epsilon) = \epsilon$ and $\alpha_{\mathfrak{F}}(P_0 P_1 \ldots P_n) = \mathring{\alpha}(P_0)\alpha_{\mathfrak{F}}(P_1 \ldots P_n)$. $\alpha_{\mathfrak{F}}$ can be lifted point-wise to $\wp(\mathbb{P}^*)$ and it gives rise to the GC $(\wp(\mathbb{P}^*), \alpha_{\mathfrak{F}}, \gamma_{\mathfrak{F}}, \wp(\mathfrak{F}^*))$. 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 $\mathcal{T}^{\mathfrak{F}} : \wp(\mathfrak{F}) \to \wp(\mathfrak{F})$ on FSA that correctly approximates $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$. One possibility is to define $\mathcal{T}^{\mathfrak{F}}$ as the best correct approximation of $\mathcal{T}^{Ph}$ on $\wp(\mathfrak{F})$, namely $\mathcal{T}^{\mathfrak{F}} = \mathring{\alpha} \circ \mathcal{T}^{Ph} \circ \mathring{\gamma}$, and function $\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!] : \wp(\mathfrak{F}^*) \to \wp(\mathfrak{F}^*)$ as follows: $\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!](K) = \mathring{\alpha}(P_0) \cup \{kM_iM_j \mid kM_i \in K, M_j \in \mathcal{T}^{\mathfrak{F}}(M_i)\}$. From $\mathcal{T}^{\mathfrak{F}}$ correctness we have $\mathbf{S}^{\mathfrak{F}}[\![P_0]\!] = \mathit{lfp}\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!]$ correctness.

Theorem 5 $\alpha_{\mathfrak{F}}(\mathit{lfp}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \subseteq \mathit{lfp}\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!] = \mathbf{S}^{\mathfrak{F}}[\![P_0]\!]$.

$\mathbf{S}^{\mathfrak{F}}[\![P_0]\!]$ approximates phase semantics by abstracting programs with FSA, while the transitions, i.e., the effect of the metamorphic engine, follow directly from $\mathcal{T}^{Ph}$ and are not approximated. For this reason $\mathbf{S}^{\mathfrak{F}}[\![P_0]\!]$ is not computable 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 for P_0:

– a stack analysis $\mathit{StackVal} : \mathbb{N} \to \wp(\mathbb{N})$ that approximates the set of possible values on the top of the stack when control reaches a given location (e.g. [1, 2]);

– a memory analysis $LocVal : \mathbb{N} \times \mathbb{N} \to \wp(\mathbb{N})$ that approximates the set of possible values that can be stored in a memory location when the control reaches a given location (e.g. [1, 2]).

These analyses allow us to define $EVal : \mathbb{N} \times \mathbb{E} \to \wp(\mathbb{N})$, that approximates the evaluation of an expression in a given point:
$EVal(b, n) = \{n\}$
$EVal(b, \texttt{MEM}[e]) = \{LocVal(b, l) \mid l \in EVal(b, e)\}$
$EVal(b, \texttt{MEM}[e_1] \ \texttt{op} \ \texttt{MEM}[e_2]) = \{n_1 \ \texttt{op} \ n_2 \mid i \in \{1, 2\} : n_i \in EVal(b, \texttt{MEM}[e_i])\}$
$EVal(\texttt{MEM}[e] \ \texttt{op} \ n) = \{n_1 \ \texttt{op} \ n \mid n_1 \in EVal(b, \texttt{MEM}[e])\}$
and a sound control flow analysis $\rho : \mathbb{I} \times \mathbb{N} \to \wp(\mathbb{N})$:
$\rho(\texttt{call} \ e, b) = \rho(\texttt{goto} \ e) = EVal(b, e)$
$\rho(\texttt{ret}, b) = StackVal(b)$
$\rho(\texttt{if} \ e_1 \ \texttt{goto} \ e_2, b) = \{b + 1\} \cup EVal(b, e_2)$
$\rho(\texttt{halt}, b) = \emptyset$
$\rho(I, b) = \{b + 1\}$ in all other cases
Moreover, we define $write : \mathring{\mathbb{I}} \times \mathbb{N} \to \wp(\mathbb{N})$ approximating the set of locations that may be modified by the execution of an abstract instruction memorized at a given location:

$$write(\mathring{I}, b) = \begin{cases} EVal(b, e_1) & \text{if } \mathring{I} = \texttt{MEM}[e_1] := e_2 \\ EVal(b, e) & \text{if } \mathring{I} \in \{\texttt{input} \Rightarrow \texttt{MEM}[e], \texttt{pop} \ e\} \\ \emptyset & \text{otherwise} \end{cases}$$

We define the *limits* of a 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 4 *A pair of program locations (b, c) is a transition edge of $M = (Q, E, S)$, denoted $(b, c) \in \texttt{TE}(M)$, if there exists $a \in S$: $\pi = a[\mathring{I}_a] \dots [\mathring{I}_{b-1}]b[\mathring{I}_b]c \in \Pi(M)$ and $c \in limit(\pi)$.*

In the FSA of Fig. 3 the transition edges are the dashed ones since the instructions labeling these edges 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 evolution of a given automaton $M = (Q, E, S)$ we need to statically execute the abstract instructions that may modify an FSA. Algorithm $\mathbf{EXE}(M, \mathring{I}, b)$ in Fig. 4 returns the set *Exe* of all possible FSA that can be obtained by executing instruction $\mathring{I}$ stored at location b of automaton M. 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 instruction $\mathring{I}$ stored at b in M. This ensures correctness when the execution of instruction $\mathring{I}$ does not correspond to a real code mutation. Then if $\mathring{I}$ writes in memory we consider the set X of locations that it can modify and the set Y of possible instructions that it can write, and we add to *Exe* the set of all possible automata that can be obtained by writing an instruction of Y in a memory location in X, i.e., $\mathbf{NEXT}(X, Y, M, b)$.

```
EXE(M, I̊, b)  // M = (Q, E, S) is a FSA
Exe = {M' = (Q, E, S') | S' = {d | (b, I̊, d) ∈ E}}
if I̊ = MEM[e_1] := e_2
  then X = write(I̊, b)
  Y = {n | n ∈ EVal(b, e_2), decode(n) ∈ 𝕀}
  Exe = Exe ∪ NEXT(X, Y, M, b)
if I̊ = input ⇒ MEM[e]
  then X = write(I̊, b)
  Y = {n | n is an input , decode(n) ∈ 𝕀}
  Exe = Exe ∪ NEXT(X, Y, M, b)
if I̊ = pop  e
  then X = write(I̊, b)
  Y = {n | n ∈ StackVal(b), decode(n) ∈ 𝕀}
  Exe = Exe ∪ NEXT(X, Y, M, b)
return Exe

NEXT(X, Y, M, b)
Next = ∅
while X ≠ ∅
  select a_j from X and X = X ∖ {a_j}
  Ê = E ∖ {(a_j, I̊_j, c) | (a_j, I̊_j, c) ∈ E}
  Next = Next ∪ ⋃_{n∈Y} { M̂ = (Q̂, Ê, Ŝ) |
      Q̂ = Q ∪ {a_j} ∪ ρ(decode(n), a_j)
      Ê = Ê ∪ {(a_j, ι(decode(n)), d) |
                d ∈ ρ(decode(n), a_j)}
      Ŝ = {d | (b, I̊, d) ∈ E} }
return Next
```

Fig. 4. Algorithm for statically executing instruction $\mathring{I}$

Let $Succ(M)$ 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:

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

We can now define the static transition $\mathcal{T}^\sharp : \wp(\mathfrak{F}) \to \wp(\mathfrak{F})$. The idea is that the possible static successors of an automaton M are all the automata in $Succ(M)$ 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 $\mathcal{T}^\sharp$, i.e., $M_l \in \mathcal{T}^{\mathfrak{F}}(M_0) \Rightarrow M_l \in \mathcal{T}^\sharp(M_0)$, even if between M_0 and M_l there are transition edges that do not correspond to any mutating transition.

Definition 5 *Let $M = (Q, E, S)$. $\mathcal{T}^\sharp : \wp(\mathfrak{F}) \to \wp(\mathfrak{F})$ is given by the point-wise extension of:*

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

This allows us to define function $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!] : \wp(\mathfrak{F}^*) \to \wp(\mathfrak{F}^*)$ that statically approximates the iterative computation of phase semantics on the abstract domain $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$ as follows: $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!](K) = \mathring{\alpha}(P_0) \cup \{kM_iM_j \mid (M_i, M_j) \in \mathcal{T}^\sharp, kM_i \in K\}$. The following result shows the correctness of $\mathbf{S}^\sharp[\![P_0]\!] = lfp\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$.

Theorem 6 $\alpha_{\mathfrak{F}}(lfp\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \subseteq lfp\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$.

In Fig. 5 we report a possible sequence of FSA that can be generated during the execution of program P_0 of Fig. 2. In this case, thanks to the simplicity of the example, it is possible to use the transition relation over FSA defined by $\mathcal{T}^{\mathfrak{F}}$.

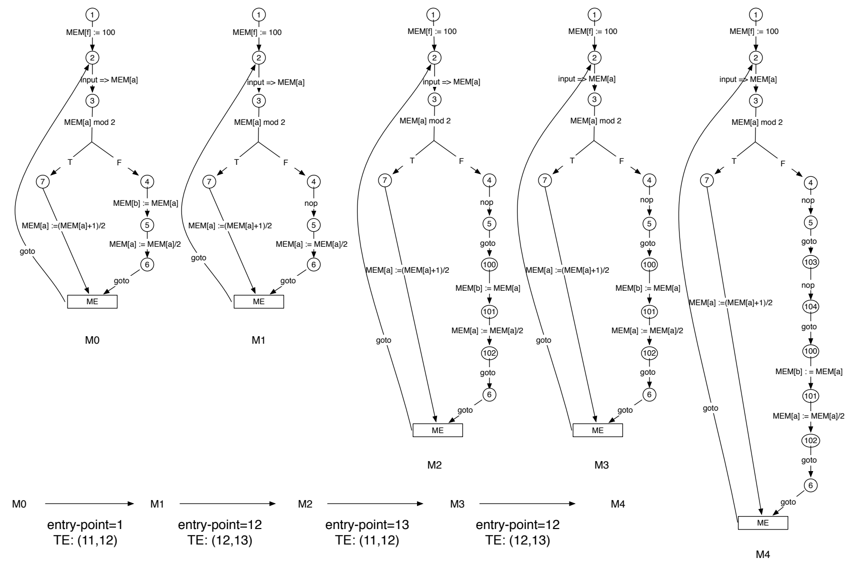


Fig. 5. Some metamorphic variants of program P_0 of Fig. 2, where the metamorphic engine, namely the instructions stored at locations from 8 to 14, is briefly represented by the box marked ME. In the graphic representation of automata we omit to show the nodes that are not reachable.

5 Widening phases for regular metamorphism

Regular metamorphism models the metamorphic behaviour 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 behaviour.

It is known that FSA can be ordered according to the language they recognize: $M_1 \sqsubseteq_{\mathfrak{F}} M_2$ if $\mathscr{L}(M_1) \subseteq \mathscr{L}(M_2)$. Observe that $\sqsubseteq_{\mathfrak{F}}$ is reflexive and transitive but not antisymmetric and it is therefore a pre-order. Moreover, according to this ordering, an 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 $\mathscr{L}(M_1) \cup \mathscr{L}(M_2)$. Given two automata $M_1 = (Q_1, \delta_1, S_1, F_1, A_1)$ and $M_2 = (Q_2, \delta_2, S_2, F_2, A_2)$, we approximate their least upper bound as follows:

$$M_1 \Cup M_2 = (Q_1 \cup Q_2, \hat{\delta}, S_1 \cup S_2, F_1 \cup F_2, A_1 \cup A_2)$$

where $\hat{\delta} : (Q_1 \cup Q_2) \times (A_1 \cup A_2) \to \wp(Q_1 \cup Q_2)$ is defined as $\hat{\delta}(q, s) = \delta_1(q, s) \cup \delta_2(q, s)$. FSA are $\Cup$-closed for finite sets, and the following result shows that $\Cup$ approximates any upper bound with respect to the ordering $\sqsubseteq_{\mathfrak{F}}$.

Lemma 1 *Given two FSA M_1 and M_2 we have: $\mathscr{L}(M_1) \cup \mathscr{L}(M_2) \subseteq \mathscr{L}(M_1 \Cup M_2)$.*

We can now define $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!] : \mathfrak{F} \to \mathfrak{F}$ as follows: $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!](M) = \mathring{\alpha}(P_0) \Cup M \Cup (\Cup\{M' \mid M' \in \mathcal{T}^\sharp(M)\})$. Observe that the set of possible successors of a given automaton M, i.e., $\mathcal{T}^\sharp(M)$, 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 $\Cup$-closed for finite sets, then $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ is well defined. Let $\wp_F(\mathfrak{F}^*)$ denote the domain of finite sets of strings of FSA and let us define function $\alpha_S : \wp_F(\mathfrak{F}^*) \to \mathfrak{F}$ as $\alpha_S(M_0 \dots M_k) = \Cup\{M_i \mid 0 \leq i \leq k\}$ and $\alpha_S(K) = \Cup\{\alpha_S(M_0 \dots M_k) \mid M_0 \dots M_k \in K\}$, with $K \in \wp_F(\mathfrak{F}^*)$. Function α_S is additive and it defines a GC $(\wp_F(\mathfrak{F}^*), \alpha_S, \gamma_S, \mathfrak{F})$. The following result shows that, when considering finite sets of sequences of FSA, $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ correctly approximates $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$ on $\mathfrak{F}$.

Theorem 7 *For any $K \in \wp_F(\mathfrak{F}^*)$ we have $\alpha_S(\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!](K)) \sqsubseteq_{\mathfrak{F}} \mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!](\alpha_S(K))$.*

The domain $\langle \mathfrak{F}, \sqsubseteq_{\mathfrak{F}} \rangle$ has infinite ascending chains, which means that, in general, the fixpoint computation of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ 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 fixpoint iteration, i.e., an element in $\mathfrak{F}$ which upper approximates the iterates of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$. We refer to the widening operation over FSA described by D'Silva [14]. Here the widening operator between two FSA $M_1 = (Q_1, E_1, S_1)$ and $M_2 = (Q_2, E_2, S_2)$ over a finite alphabet A is formalized in terms of an equivalence relation $R \subseteq Q_1 \times Q_2$ between states. R, also called *widening seed*, is used to define another equivalence relation $\equiv_R \subseteq Q_2 \times Q_2$ over the states of M_2, such that $\equiv_R = R \circ R^{-1}$. The widening between M_1 and M_2 is then given by the quotient of M_2 with respect to the partition induced by $\equiv_R$: $M_1 \triangledown M_2 = M_2/\equiv_R$. By changing the widening seed we obtain different widening operators. It has been proved that convergence is guaranteed when the widening seed is the relation $R_n \subseteq Q_1 \times Q_2$ such that $(q_1, q_2) \in R_n$ if q_1 and q_2 recognize the same language of strings of length at most n [14]. When considering the widening seed R_n we have that two states q and q' of M_2 are $\equiv_{R_n}$-equivalent if they recognize the same language of length at most n that is recognized by a state r of M_1, i.e., if $\exists r \in Q_1 : (r, q) \in R_n$ and $(r, q') \in R_n$. $\triangledown_n$ denotes the widening operator that uses R_n as widening seed. $\triangledown_n$ is well defined if $\mathring{\mathbb{I}}$ is finite. This can be achieved by considering expressions as terms and by applying some of the standard methods for approximating them. The most straightforward one is the depth-k string abstraction [24], while more refined expression abstractions can be designed by considering graph-based or grammar-based term abstractions [3, 15]. For simplicity we consider here the depth-k term abstraction where expressions are represented as trees with leafs that are natural numbers denoting either a memory location or a constant, and internal nodes are the operators constructing expressions, namely the unary operator `MEM` or the binary operators `op`. We annotate each node with its depth, namely with the length of the path from the root to the node. The depth-k abstraction, given a tree representation of an expression, considers only the nodes with depth less or equal to k and "cuts" the remaining nodes by approximating them with $\top$. For example, the depth-3 abstraction of expression $\mathtt{MEM}[(\mathtt{MEM}[a]\ \mathtt{op}\ \mathtt{MEM}[b\ \mathtt{op}\ \mathtt{MEM}[c]])\ \mathtt{op}\ d]$ is $\mathtt{MEM}[(\mathtt{MEM}[\top]\ \mathtt{op}\ \mathtt{MEM}[\top])\ \mathtt{op}\ d]$. Given $k \in \mathbb{N}$, let $\iota_k : \mathring{\mathbb{I}} \to \mathring{\mathbb{I}}_k$ be the instruction abstraction that applies the depth-k abstraction to the expressions occurring in an abstract instruction, and let $\alpha_k : \mathfrak{F} \to \mathfrak{F}_k$ be the function that abstracts the edge labels of a FSA in $\mathfrak{F}$ according to ι_k. It is possible to show that $(\mathfrak{F}, \alpha_k, \gamma_k, \mathfrak{F}_k)$ is a GC, where $\gamma_k(M^k) = \Cup\{M' \mid \alpha_k(M') \sqsubseteq_{\mathfrak{F}} M^k\}$. This allows us to approximate the least fixpoint of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ on $\langle \mathfrak{F}_k, \sqsubseteq_{\mathfrak{F}}\rangle$ with the limit $\mathbf{W}[\![P_0]\!]$ of the following widening sequence: $W_0 = \alpha_k(\mathring{\alpha}(P_0))$ and $W_{i+1} = W_i \bigtriangledown_n \alpha_k(\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!](\gamma_k(W_i)))$. Let us refer to $\mathbf{W}[\![P_0]\!]$ as the *widened fixpoint* of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ and to $W_0 W_1, \ldots$ as the *widening sequence* of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$. From the correctness of $\bigtriangledown_n$ and by Theorem 7, it follows that the widening sequence $W_0 W_1 \ldots$ converges to an upper-approximation of the least fixpoint of $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$, namely any automata modelling a possible static variant of P_0 is approximated by $\mathbf{W}[\![P_0]\!]$ i.e., $\ldots M_i \ldots \in \mathit{lfp}^{\subseteq}\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!] \;\Rightarrow\; M_i \sqsubseteq_{\mathfrak{F}} \mathbf{W}[\![P_0]\!]$. Therefore $\mathscr{L}(\mathbf{W}[\![P_0]\!])$ contains all the possible sequences of abstract instructions that can be executed by a metamorphic variant of 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 up to address renaming: $P_0 \rightsquigarrow_{\mathfrak{F}_k} Q$ iff there exists an address renaming ϑ such that $\vartheta(\mathscr{L}(\alpha_k(\mathring{\alpha}(Q)))) \subseteq \mathscr{L}(\mathbf{W}[\![P_0]\!])$. The language $\mathscr{L}(\mathbf{W}[\![P_0]\!])$ represents the regular metamorphic signature of P_0 and 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. Fig. 6 (a) shows the widened fixpoint $\mathbf{W}[\![P_0]\!]$ of program P_0 in Fig. 2, where the widening seed is R_2 and $k \geq 3$, This automaton recognizes any possible program that can be obtained during the execution of P_0. Note that, we may have false positives, as for example the sequences of instructions along the bold path $\mathtt{MEM}[f] := 100; \mathtt{input} \Rightarrow \mathtt{MEM}[a]; \mathtt{MEM}[a] \bmod 2 = 0; \mathtt{MEM}[b] := \mathtt{MEM}[a]; \mathtt{goto}; \mathtt{MEM}[b] := \mathtt{MEM}[a]; \mathtt{goto}; \ldots$ which is not a run of any of the variants of P_0. Regular metamorphism can easily cope with metamorphic transformations commonly used by malware (e.g., `Win95/Regswap`, `Win32/Ghost`, `Win95/Zperm`, `Win95/Zmorph`, `Win32/Evol`) such as: *register swap* that changes the registers used by the program; *code permutation* that changes the order in which instructions appear in memory while preserving their execution order through the insertion of direct jumps; *junk/nop insertion* that inserts junk instructions and semantic-nops, namely instructions that are not executed or that do not alter program functionality. Observe that all these transformations can be seen as special cases of *code substitution*. Let P_0 be a metamorphic malware: whenever a sequence s_1 of instructions is substituted with an equivalent one s_2, we have that during the widened fixpoint computation a new path containing sequence s_2 is added to the widened fixpoint $\mathbf{W}[\![P_0]\!]$. Therefore, by correctness, $\mathbf{W}[\![P_0]\!]$ recognizes all the possible metamorphic variants of P_0 obtained through code substitution. Of course it is possible to further abstract $\mathbf{W}[\![P_0]\!]$ in order to address semantic-nop/junk insertion, permutation and register swap in a more efficient way, namely in such a way that the resulting widened fixpoint is an automaton of a reduced size. In semantic-nop insertion, the more precise is the static analysis used for identifying (sequences of) instructions that are equivalent to `nop`, the smaller is the widened fixpoint $\mathbf{W}[\![P_0]\!]$ that we obtain. In code permutation, a smaller FSA can be obtained by performing `goto`-reduction, i.e., by folding nodes reachable by `goto`-instructions. In register swapping it is sufficient to replace registers names (i.e., memory locations) with uninterpreted symbols and then use unification to bind these uninterpreted symbols to the actual register names (i.e., memory locations) as done in [5]. Let us consider program P_0^+ obtained by enriching the metamorphic engine of program P_0 of Fig. 2 with a code permutation and a transformation that substitutes instruction $\texttt{MEM}[e_1] := e_2$ with the equivalent sequence $\texttt{push}\ e_1, \texttt{pop}\ e_2$. A possible evolution is shown below, where ME denotes the metamorphic engine.

```
P0+ :
  1 : goto 8
  2 : if (MEM[a] mod 2) goto 11
  3 : nop
  4 : goto 100
  5 : push MEM[a]/2
  6 : pop a
  7 : goto 12
  8 : MEM[f] := 100
  9 : input => MEM[a]
 10 : goto 2
 11 : MEM[a] := (MEM[a] + 1)/2
 12 : ME
 13 : goto 9
100 : push MEM[a]
101 : pop b
102 : goto 5
```

Fig. 6 (b) shows the FSA that represents an approximation of all the possible evolutions of program P_0^+ when $k \geq 3$. This FSA is obtained through widening with widening seed R_2 and by applying the `goto`-reduction to handle permutation. We can observe that every time that in the automaton in Fig. 6 (b) we have an edge labeled with $\texttt{MEM}[e_1] := e_2$ between two states q and p, then we also have a path labeled with $\texttt{push}\ e_2, \texttt{pop}\ e_1$ that connects q and p, and this precisely captures the fact that the metamorphic engine implements this substitution. The `goto`-reduction allows here to have a reduced FSA, and the self-loop labeled with `nop` makes clear that the metamorphism could insert an unbounded number of `nop` instructions.

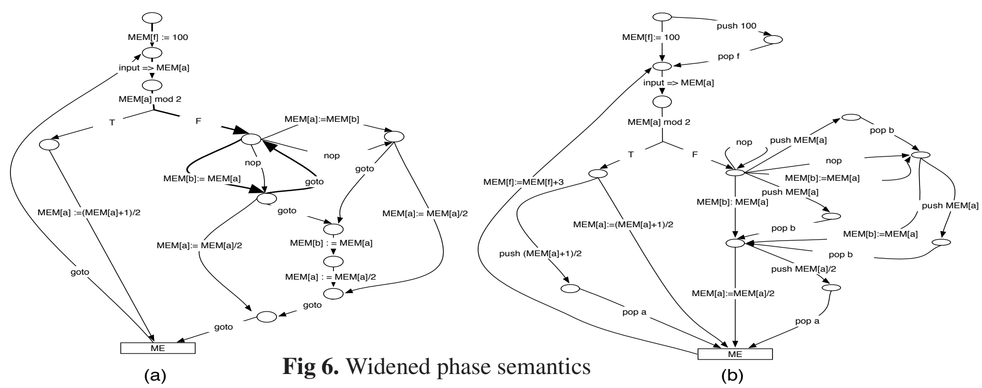


Fig 6. Widened phase semantics

6 Related Works and Discussion

In [13] the authors use trace semantics to characterize the behaviours of both the malware and the potentially infected program, and use abstract interpretation to "hide" their irrelevant behaviours. A program is infected by a malware if their behaviours are indistinguishable up to a certain abstraction, which corresponds to some obfuscations. A significant limitation of this work is that the knowledge of the obfuscation is essential in order to derive abstractions. In [19] the authors model the malware M as a formula in the new logic CTPL, which is an extension of CTL able to handle register renaming. A program P is infected by 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. In [7] the idea is to model the malware as a template that expresses the malicious intent. Also in this case the definition of the template is driven by the knowledge of the obfuscations commonly used by malware. Some researchers have tried to detect metamorphic malware by modelling the metamorphic engine as formal grammars and automata [16, 20, 25]. 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. To the best of our knowledge, we are not aware of any work modelling metamorphism without any a priori knowledge of the transformations used by the metamorphic engine. The only other work we are aware of that formally addresses the analysis of self-modifying code is the one of Cai et al. [4]. However, their goals and results are very different from ours: Cai et al. 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. In this sense, our key contribution relies upon the idea that abstract interpretation of phase semantics may provide useful information about the way code changes, i.e., about the metamorphic engine itself. 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., the metamorphic engine. With our approach it is therefore possible to extract properties of the implementation of the metamorphic engine by abstract interpretation of the phase semantics. It is clear that the depth-k abstraction considered here for approximating the language of instructions towards a finite alphabet for widening traces of FSA is for sake of simplicity. In general, widening phases for taming the sequence of modified programs (FSA) generated by metamorphism into a single FSA modeling regular metamorphism may require a notion of *higher-order widening* on FSA, acting both at the level of the graph-structure of the FSA, for approximating the language of instructions, and at the level of the instruction set, for approximating the way a single instruction may be composed. The abstraction of code layout may induce the abstraction of instructions, which itself can be solved by means of FSA. 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. Of course FSA provide just regular language-based abstractions of the metamorphic engine. More sophisticated approximations, using for instance *a la Cousot*'s context free grammars and set-constraint-based abstractions of sequences of binary instructions [10], may provide alternative and effective solutions for non-regular metamorphism.

References

1. G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-a platform for analyzing x86 executables. In *Proc. Internat. Conf. on Compiler Construction (CC'05)*, pp. 250–254, 2005.
2. G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Proc. Internat. Conf. on Compiler Construction (CC'04)*, pp. 5–23, 2004.

3. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc. Symposium on Logic Programming*, pp. 192–204, 1987.
4. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. ACM conf. on Programming Language Design and Implementation (PLDI'07)*, pp. 66–77, 2007.
5. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proc. USENIX Security Symp.*, pp. 169–186, 2003.
6. M. Christodorescu and S. Jha. Testing malware detectors. In *Proc. ACM SIGSOFT Internat. Symp. on Software Testing and Analysis* (*ISSTA '04*), pp. 34–44, 2004.
7. M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. IEEE Security and Privacy*, pp. 32–46, 2005.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages* (*POPL '77*), pp. 238–252, 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages* (*POPL '79*), pp. 269–282, 1979.
10. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 170–181, 1995.
11. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.* **277**(1-2): 47-103, 2002.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. ACM Symp. on Principles of Programming Languages* (*POPL '78*), 1978.
13. 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.
14. V. D'Silva. Widening for automata. Diploma Thesis, Institut Fur Informatick, Universitat Zurich, 2006.
15. M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. ACM Conf. Programming language design and implementation*, pp. 242–256, 1994.
16. E. Filiol. Metamorphism, formal grammars and undecidable code mutation. In *Proc. World Academy of Science, Engineering and Technology (PWASET)*, vol. 20, 2007.
17. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM.*, 47(2):361–416, 2000.
18. A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *Proc. Internat. Conf. on Computer Aided System Theory*, vol. 4739 of *LNCS*, pp. 497–504, 2007.
19. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proc. Internat. Conf. on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, vol. 3548 of *LNCS*, pp. 174–187, 2005.
20. Qozah. Polymorphism and grammars. *29A E-zine*, 2009.
21. P. Singh and A. Lakhotia. Static verification of worm and virus behaviour in binary executables using model checking. In *Proc. IEEE Information Assurance Workshop*, 2003.
22. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.
23. P. Ször and P. Ferrie. Hunting for metamorphic. In *Proc. Virus Bulleting Conference*, pp. 123–144. Virus Bulletin Ltd, 2001.
24. H.Tamaki and T. Sato. Program Transformation Through Meta-shifting. *New Generation Computing*, 1(1):93–98, 1983.
25. P. Zbitskiy. Code mutation techniques by means of formal grammars and automatons. *Journal in Computer Virology*, 2009.