A Semantics-Based Approach to Malware Detection [*]

Mila Dalla Preda
Dipartimento di Informatica,
University of Verona,
Strada le Grazie 15, 37134 Verona, Italy.
dallapre@sci.univr.it

Mihai Christodorescu and Somesh Jha
Department of Computer Science,
University of Wisconsin, Madison, WI
53706, USA.
{mihai,jha}@cs.wisc.edu

Saumya Debray
Department of Computer Science,
University of Arizona, Tucson, AZ
85721, USA.
debray@cs.arizona.edu

Abstract

Malware detection is a crucial aspect of software security. Current malware detectors work by checking for "signatures," which attempt to capture (syntactic) characteristics of the machine-level byte sequence of the malware. This reliance on a syntactic approach makes such detectors vulnerable to code obfuscations, increasingly used by malware writers, that alter syntactic properties of the malware byte sequence without significantly affecting their execution behavior. This paper takes the position that the key to malware identification lies in their semantics. It proposes a semantics-based framework for reasoning about malware detectors and proving properties such as soundness and completeness of such detectors. Our approach uses a trace semantics to characterize the behaviors of malware as well as the program being checked for infection, and uses abstract interpretation to "hide" irrelevant aspects of these behaviors. As a concrete application of our approach, we show that the semantics-aware malware detector proposed by Christodorescu *et al.* is complete with respect to a number of common obfuscations used by malware writers.

1. Introduction

Malware is a program with malicious intent that has the potential to harm the machine on which it executes or the network over which it communicates. A *malware detector* identifies malware. A *misuse malware detector* (or, alternately, a *signature-based malware detector*) uses a list of signatures (traditionally known as a *signature database* [19]). For example, if part of a program matches a signature in the database, the program is labeled as malware [22]. Their low false-positive rate and ease of use have led to widespread deployment of such systems.

Malware writers continuously test the limits of malware detectors in an attempt to discover ways to evade detection. This leads to an ongoing game of one-upmanship [20], where malware writers find new ways to create undetected malware, and where researchers design new signature-based techniques for detecting such evasive malware. This co-evolution is a result of the theoretical undecidability of malware detection [2, 5]. This means that, in the currently accepted model of computation, no ideal malware detector exists. The only achievable goal in this scenario is to design better detection techniques that jump ahead of evasion techniques and make the malware writer's task harder.

Attackers have resorted to two main approaches for evading malware detectors: *program obfuscation* and *program evolution*. Program obfuscation transforms a program by inserting new code or modifying existing code to make understanding and detection harder, at the same time preserving the malicious behavior. Obfuscation transformations can easily defeat signature-based detection mechanisms. If a signature describes a certain sequence of instructions [22], then those instructions can be reordered or replaced with equivalent instructions [25, 26]. Such obfuscations are especially applicable on CISC architectures, such as the Intel IA-32 [15], where the instruction set is rich and many instructions have overlapping semantics. If a signature describes a certain distribution of instructions in the program, insertion of junk code [16, 23, 26] that acts as a nop so as not to modify the program behavior can defeat frequency-based signatures. If a signature identifies some of the read-only data of a program, packing or encryption with varying keys [13, 21] can effectively hide the relevant data. *Therefore, an important requirement of a robust malware detection technique is to handle obfuscation transformations*.

Program semantics provides a formal model of program behavior. Therefore addressing the malware-detection problem from a semantic point of view could lead to a more robust detection system. Preliminary work by Christodorescu *et al.* [4] and Kinder *et al.* [17] on a formal approach to malware detection confirms the potential benefits of a semantics-based approach to malware detection. The goal of this paper is to provide a formal semantics-based framework that can be used by security researchers to reason about and evaluate the resilience of malware detectors to various kinds of obfuscation transformations. This paper makes the following specific contributions:

- We present a formal definition of what it means for a detector to be sound and complete with respect to a class of obfuscations. We also provide a framework which can be used by malware-detection researchers to prove that their detector is complete with-respect-to a class of obfuscations. As an integral part of

[*] The work of S. Debray was supported in part by the National Science Foundation under grants EIA-0080123, CCR-0113633, and CNS-0410918.

The work of M. Christodorescu and S. Jha was supported in part by the Office of Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

the formal framework, we provide a trace semantics to characterize the program and malware behaviors, using abstract interpretation to "hide" irrelevant aspects of these behaviors.

- We show our formal framework in action by proving that the semantic-aware malware detector $\mathcal{A}_{MD}$ proposed by Christodorescu *et al.* [4] is complete with respect to the some common obfuscations used by malware writers. The soundness of $\mathcal{A}_{MD}$ was proved in [4].

2. Preliminaries

Let P be the set of programs. An *obfuscation* is a program transformer, $\mathcal{O} : \mathsf{P} \rightarrow \mathsf{P}$. Code reordering and variable renaming are two common obfuscations. The set of all obfuscations is denoted by $\mathbf{O}$.

A *malware detector* is $D : \mathsf{P} \times \mathsf{P} \rightarrow \{0, 1\}$: $D(P, M) = 1$ means that P is infected with M or with an obfuscated variant of M. Our treatment of malware detectors is focused on detecting variants of existing malware. When a program P is infected with a malware M, we write $M \hookrightarrow P$. Intuitively, a malware detector is *sound* if it never erroneously claims that a program is infected, i.e., there are no false positives; and it is *complete* if it always detects programs that are infected, i.e., there are no false negatives. More formally, these properties can be defined as follows:

DEFINITION 1 (Soundness and Completeness). *A malware detector D is complete for an obfuscation $\mathcal{O} \in \mathbf{O}$ if and only if $\forall M \in \mathsf{P}$, $\mathcal{O}(M) \hookrightarrow P \Rightarrow D(P, M) = 1$. A malware detector D is sound for an obfuscation $\mathcal{O} \in \mathbf{O}$ if and only if $\forall M \in \mathsf{P}$, $D(P, M) = 1 \Rightarrow \mathcal{O}(M) \hookrightarrow P$.*

Note that this definition of soundness and completeness can be applied to a deobfuscator as well. In other words, our definitions are not tied to the concept of malware detection. Most malware detectors are built on top of other static-analysis techniques for problems that hard or undecidable. For example, most malware detectors [4, 17] that are based on static analysis assume that the control-flow graph for an executable can be extracted. As shown by researchers [18], simply disassembling an executable can be quite tricky. Therefore, we want introduce the notion of *relative soundness and completeness* with respect to algorithms that a detector uses. In other words, we want to prove that a malware detector is sound or complete with respect to a class of obfuscations if the static-analysis algorithms that the detector uses are perfect.

DEFINITION 2 (Oracle). *An oracle is an algorithm over programs. For example, a CFG oracle is an algorithm that takes a program as an input and produces its control-flow graph.*

$D^{\mathcal{OR}}$ denotes a detector that uses a set of oracles $\mathcal{OR}$.[1] For example, let OR_{CFG} be a static-analysis oracle that given an executable provides a perfect control-flow graph for it. A detector that uses the oracle OR_{CFG} is denoted as $D^{OR_{CFG}}$. In the definitions and proofs in the rest of the paper we assume that oracles that a detector uses are perfect.

DEFINITION 3 (Soundness and completeness relative to oracles). *A malware detector $D^{\mathcal{OR}}$ is complete with respect to an obfuscation $\mathcal{O}$, if D is complete for that obfuscation $\mathcal{O}$ given that all oracles in the set $\mathcal{OR}$ are perfect. Soundness of a detector $D^{\mathcal{OR}}$ can be defined in a similar manner.*

[1] We assume that detector D can query an oracle from the set $\mathcal{OR}$, and the query is answered perfectly and in $O(1)$ time. This type of relative completeness and soundness results are common in cryptography.

2.1 A Framework for Proving Soundness and Completeness of Malware Detectors

When a new malware detection algorithm is proposed, one of the criteria of evaluation is its resilience to obfuscations. Unfortunately, identifying the classes of obfuscations for which a detector is resilient can be a complex and error-prone task. A large number of obfuscation schemes exist, both from the malware world and from the intellectual-property protection industry. Furthermore, obfuscations and detectors are defined using different languages (e.g., program transformation vs program analysis), complicating the task of comparing one against the other.

We present a framework for proving soundness and completeness of malware detectors in the presence of obfuscations. This framework operates on programs described through their execution traces—thus, program trace semantics is the building block of our framework. Both obfuscations and detectors can be elegantly expressed as operations on traces (as we describe in Section 3 and Section 4).

In this framework, we propose the following two step *proof strategy* for showing that a detector is sound or complete with respect to an obfuscation or a class of obfuscations.

1. [Step 1] Relating the two worlds.
   Let $D^{\mathcal{OR}}$ be a detector that uses a set of oracles $\mathcal{OR}$. Assume that we are given a program P and malware M. Let $\mathfrak{S}\llbracket P\rrbracket$ and $\mathfrak{S}\llbracket M\rrbracket$ be the set of traces corresponding to P and M, respectively. In Section 3 we describe a detector D_{Tr} which works in the semantic world of traces. We then prove that if the oracles in $\mathcal{OR}$ are perfect, the two detectors are equivalent, i.e, for all P and M in P, $D^{\mathcal{OR}}(P, M) = 1$ iff $D_{Tr}(\mathfrak{S}\llbracket P\rrbracket, \mathfrak{S}\llbracket M\rrbracket) = 1$. In other words, this step shows the equivalence of the two worlds: the concrete world of programs and the semantic world of traces.
2. [Step 2] Proving soundness and completeness in the semantic world.
   After step 1, we prove the desired property (e.g., completeness) about the trace-based detector D_{Tr}, with respect to the chosen class of obfuscations. In this step, the detector effects on the trace semantics are compared to the effects of obfuscation on the trace semantics. This also allows us to evaluate the detector against whole classes of obfuscations, as long as the obfuscations have similar effects on the trace semantics.

The requirement for equivalence in step 1 above might be too strong if only one of completeness or soundness is desired. For example, if the goal is to prove only completeness of a malware detector $D^{\mathcal{OR}}$, then it is sufficient to find a trace-based detector that classifies only malware and malware variants in the same way as $D^{\mathcal{OR}}$. Then, if the trace-based detector is complete, so is $D^{\mathcal{OR}}$.

2.2 Abstract Interpretation

The basic idea of abstract interpretation is that program behaviour at different levels of abstraction is an approximation of its formal semantics [8, 9]. The (concrete) semantics of a program is computed on the (concrete) domain $\langle C, \leq_C\rangle$, i.e., a complete lattice which models the values computed by programs. The partial ordering $\leq_C$ models relative precision between concrete values. An abstract domain $\langle A, \leq_A\rangle$ is a complete lattice which encodes an approximation of concrete program values. As usual abstract domains are specified by Galois connections [8, 9]. Two complete lattices C and A form a Galois connection (C, α, γ, A), when $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ form an adjunction, namely $\forall a \in A, \forall c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ where $\alpha(\gamma)$ is the left(right) adjoint of $\gamma(\alpha)$. α and γ are called, respectively, abstraction and concretization maps. A tuple (C, α, γ, A) is a Ga-

Syntactic Categories:

$n \in \mathbf{N}$	(integers)
$X \in \mathbf{X}$	(variable names)
$L \in \mathbf{L}$	(labels)
$E \in \mathbf{E}$	(integer expressions)
$B \in \mathbf{B}$	(Boolean expressions)
$A \in \mathbf{A}$	(actions)
$D \in \mathbf{E} \cup (\mathbf{A} \times \wp(\mathbf{L}))$	(assignment r-values)
$C \in \mathbf{C}$	(commands)
$P \in \mathbf{P}$	(programs)

Value Domains:

$\mathbb{B} = \{true, false\}$	(truth values)
$n \in \mathbb{Z}$	(integers)
$\rho \in \mathcal{E} = \mathbf{X} \to \mathbf{L}_\perp$	(environments)
$m \in \mathcal{M} = \mathbf{L} \to \mathbb{Z} \cup (\mathbf{A} \times \wp(\mathbf{L}))$	(memory)
$\xi \in \mathcal{X} = \mathcal{E} \times \mathcal{M}$	(execution contexts)
$\Sigma = \mathbf{C} \times \mathcal{X}$	(program states)

Syntax:

$$
\begin{array}{ll}
E ::= n \mid X \mid E_1 \;\texttt{op}\; E_2 & (\texttt{op} \in \{+, -, *, /, \ldots\}) \\
B ::= true \mid false \mid E_1 < E_2 \mid \neg B_1 \mid B_1 \;\&\&\; B_2 & \\
A ::= X := D \mid \texttt{skip} \mid \texttt{assign}(L, X) & \\
C ::= L : A \to L' & \text{(unconditional actions)} \\
\quad \mid \; L : B \to \{L_T, L_F\} & \text{(conditional jumps)} \\
P ::= \wp(C) &
\end{array}
$$

Semantics:

ARITHMETIC EXPRESSIONS

$$\mathscr{E} : \mathbf{A} \times \mathcal{X} \to \mathbb{Z}_\perp \cup (\mathbf{A} \times \wp(\mathbf{L}))$$
$$\mathscr{E}[\![n]\!]\,\xi \;=\; n$$
$$\mathscr{E}[\![X]\!]\,\xi \;=\; m(\rho(X)), \text{ where } \xi = (\rho, m)$$
$$\mathscr{E}[\![E_1 \;\texttt{op}\; E_2]\!]\,\xi \;=\; \text{if } (\mathscr{E}[\![E_1]\!]\,\xi \in \mathbb{Z} \text{ and } \mathscr{E}[\![E_2]\!]\,\xi \in \mathbb{Z}) \text{ then } \mathscr{E}[\![E_1]\!]\,\xi \;\texttt{op}\; \mathscr{E}[\![E_2]\!]\,\xi;\ \text{else } \perp$$

BOOLEAN EXPRESSIONS

$$\mathscr{B} : \mathbf{B} \times \mathcal{X} \to \mathbb{B}_\perp$$
$$\mathscr{B}[\![true]\!]\,\xi \;=\; true$$
$$\mathscr{B}[\![false]\!]\,\xi \;=\; false$$
$$\mathscr{B}[\![E_1 < E_2]\!]\,\xi \;=\; \text{if } (\mathscr{E}[\![E_1]\!]\,\xi \in \mathbb{Z} \text{ and } \mathscr{E}[\![E_2]\!]\,\xi \in \mathbb{Z}) \text{ then } \mathscr{E}[\![E_1]\!]\,\xi < \mathscr{E}[\![E_2]\!]\,\xi;\ \text{else } \perp$$
$$\mathscr{B}[\![\neg B]\!]\,\xi = \text{ if } (\mathscr{B}[\![B]\!]\,\xi \in \mathbb{B}) \text{ then } \neg\mathscr{B}[\![B]\!]\,\xi;\ \text{else } \perp$$
$$\mathscr{B}[\![B_1 \;\&\&\; B_2]\!]\,\xi \;=\; \text{if } (\mathscr{B}[\![B_1]\!]\,\xi \in \mathbb{B} \text{ and } \mathscr{B}[\![B_2]\!]\,\xi \in \mathbb{B}) \text{ then } \mathscr{B}[\![B_1]\!]\,\xi \wedge \mathscr{B}[\![B_2]\!]\,\xi;\ \text{else } \perp$$

ACTIONS

$$\mathscr{A} : \mathbf{A} \times \mathcal{X} \to \mathcal{X}$$
$$\mathscr{A}[\![\texttt{skip}]\!]\,\xi \;=\; \xi$$
$$\mathscr{A}[\![X := D]\!]\,\xi \;=\; (\rho, m') \text{ where } \xi = (\rho, m), m' = m[\rho(X) \leftarrow \delta], \text{ and } \delta = \begin{cases} D & \text{if } D \in \mathbf{A} \times \wp(\mathbf{L}) \\ \mathscr{E}[\![D]\!]\,(\rho, m) & \text{if } D \in \mathbf{E} \end{cases}$$
$$\mathscr{A}[\![\texttt{assign}(L', X)]\!]\,\xi \;=\; (\rho', m) \text{ where } \xi = (\rho, m) \text{ and } \rho' = \rho[X \rightsquigarrow L']$$

COMMANDS

The semantic function $\mathscr{C}$ effectively specifies the transition relation between states. Here, $lab[\![C]\!]$ denotes the label for the command C, i.e., $lab[\![L : A \to L']\!] \;=\; L$ and $lab[\![L : B \to \{L_T, L_F\}]\!] \;=\; L$.

$$\mathscr{C} : \Sigma \to \wp(\Sigma)$$
$$\mathscr{C}[\![L : A \to L']\!]\,\xi \;=\; \{(C, \xi') \mid lab[\![C]\!] = L', \xi' = \mathscr{A}[\![A]\!]\,\xi\}$$
$$\mathscr{C}[\![L : B \to \{L_T, L_F\}]\!]\,\xi \;=\; \{(C, \xi) \mid lab[\![C]\!] = \begin{cases} L_T & \text{if } \mathscr{B}[\![B]\!]\,\xi = true \\ L_F & \text{if } \mathscr{B}[\![B]\!]\,\xi = false \end{cases} \}$$

Figure 1. A simple programming language.

lois connection iff α is additive iff γ is co-additive. This means that whenever we have an additive(co-additive) function f between two domains we can always build a Galois connection by considering the right(left) adjoint map induced by f. Given two Galois connections $(C, \alpha_1, \gamma_1, A_1)$ 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 a Galois connection. (C, α, γ, A) specifies a Galois insertion if each element of A is abstraction of concrete element in C, namely (C, α, γ, A) is a Galois insertion iff α is surjective iff γ is injective. Abstract domains can be related to each other w.r.t. their relative degree of precision. We say that an abstraction $\alpha_1 : C \to A_1$ is more concrete then $\alpha_2 : C \to A_2$, i.e., A_2 is more abstract than A_1, namely if $\forall c \in C : \gamma_1(\alpha_1(c)) \leq_C \gamma_2(\alpha_2(c))$.

2.3 Programming Language

The language we consider is a simple extension of the one introduced by Cousot and Cousot [10], the main difference being the ability of programs to generate code dynamically (this facility is added to accommodate certain kinds of malware obfuscations where the payload is unpacked and decrypted at runtime). The syntax and semantics of our language are given in Figure 1. Given a set S, we use $S_\perp$ to denote the set $S \cup \{\perp\}$, where $\perp$ denotes an undefined value.[2] Program variables are integer-valued, i.e., range over $\mathbb{Z}_\perp$. Commands can be either conditional or unconditional. A conditional command at a label L has the form '$L : B \to \{L_T, L_F\}$,'

[2] We abuse notation and use $\perp$ to denote undefined values of different types, since the type of an undefined value is usually clear from the context.

where B is a Boolean expression and L_T (respectively, L_F) is the label of the command to execute when B evaluates to *true* (respectively, *false*); an unconditional command at a label L is of the form '$L : A \to L_1$,' where A is an action and L_1 the command to be executed next. A variable can store either an integer or a (appropriately encoded) pair $(A, S) \in \mathbf{A} \times \wp(\mathbf{L})$. A program consists of a set of commands (since each command explicitly mentions its successors, the program need not maintain an explicit sequence of commands).

An *environment* $\rho \in \mathcal{E}$ maps variables in $dom(\rho) \subseteq \mathbf{X}$ to memory locations $\mathbf{L}_\perp$. Given a program P we denote with $\mathcal{E}(P)$ its environments, i.e. if $\rho \in \mathcal{E}(P)$ then $dom(\rho) = var[\![P]\!]$. Let $\rho[X \rightsquigarrow L]$ denote environment ρ where label L is assigned to variable X. The *memory* is represented as a function $m : \mathbf{L} \to \mathbb{Z}_\perp \cup (\mathbf{A} \times \wp(\mathbf{L}))$. Let $m[L \leftarrow D]$ denote memory m where element D is stored at location L. When considering a program P, we denote with $\mathcal{M}(P)$ the set of program memories, namely if $m \in \mathcal{M}(P)$ then $dom(m) = Luse[\![P]\!]$. This means that $m \in \mathcal{M}(P)$ is defined on the set of memory locations that are affected by the execution of program P (excluding the memory locations storing the commands of P).

Labels:

$lab[\![L : A \to L']\!] = L$
$lab[\![L : B \to \{L_T, L_F\}]\!] = L$
$lab[\![P]\!] = \{lab[\![C]\!] | C \in P\}$

Successors of a comamnd:

$suc[\![L : A \to L']\!] = L'$
$suc[\![L : B \to \{L_T, L_F\}]\!] = \{L_T, L_F\}$

Action of a command:

$act[\![L : A \to L_2]\!] = A$

Variables:

$var[\![L_1 : A \to L_2]\!] = var[\![A]\!]$
$var[\![P]\!] = \bigcup_{C \in P} var[\![C]\!]$
$var[\![A]\!] = \{$ variables occuring in $A\}$

Memory locations used by a program:

$Luse[\![L : A \to L']\!] = Luse[\![A]\!]$
$Luse[\![P]\!] = \bigcup_{C \in P} Luse[\![C]\!]$
$Luse[\![A]\!] = \{$locations occuring in $A\} \cup \rho(var[\![A]\!])$

Figure 2. Auxiliary functions for the language of Figure 1.

The behavior of a command when it is executed depends on its *execution context*, i.e., the environment and memory in which it is executed. The set of execution contexts is given by $\mathcal{X} = \mathcal{E} \times \mathcal{M}$. A *program state* is a pair (C, ξ) where C is the next command that has to be executed in the execution context ξ. $\Sigma = \mathbf{C} \times \mathcal{X}$ denotes the set of all possible states. Given a state $s \in \Sigma$, the semantic function $\mathscr{C}(s)$ gives the set of possible successor states of s; in other words, $\mathscr{C} : \Sigma \to \wp(\Sigma)$ defines the transition relation between states. Let $\Sigma(P) = P \times \mathcal{X}(P)$ be the set of states of a program P, then we can specify the transition relation $\mathscr{C}$ on P as follows:

$$\mathscr{C}[\![P]\!](C, \xi) = \big\{(C', \xi') \,\big|\, (C', \xi') \in \mathscr{C}(C, \xi), C' \in P, \text{ and } \xi, \xi' \in \mathcal{X}(P)\big\}.$$

Let A^* denote the Kleene closure of a set A, i.e., the set of finite sequences over A. A *trace* $\sigma \in \Sigma^*$ is a sequence of states $s_1 ... s_n$ of length $|\sigma| \geq 0$ such that for all $i \in [1, n)$: $s_i \in \mathscr{C}(s_{i-1})$. The *finite partial traces semantics* $\mathfrak{S}[\![P]\!] \subseteq \Sigma^*$ of program P is the least fixpoint of the function F:

$$F[\![P]\!](T) = \Sigma(P) \cup \{ss'\sigma | s' \in \mathscr{C}[\![P]\!](s),\ s'\sigma \in T\}$$

where T is a set of traces, namely $\mathfrak{S}[\![P]\!] = lfp^{\subseteq} F[\![P]\!]$. The set of all partial trace semantics, ordered by set inclusion, forms a complete lattice.

Finally, we use the following notation. Given a function $f : A \to B$ and a set $S \subseteq A$, we sue $f_{|S}$ to denote the restriction of function f to variables in $S \cap A$, and $f \smallsetminus S$ to denote the restriction of function f to elements not in S, namely to $A \smallsetminus S$.

3. Semantics-Based Malware Detection

Intuitively, a program P is infected by a malware M if (part of) P's execution behavior is similar to that of M. In order to detect the presence of a malicious behaviour M in a program P, therefore, we need to check whether there is a part (a restriction) of $\mathfrak{S}[\![P]\!]$ that "matches" (in a sense that will be made precise) $\mathfrak{S}[\![M]\!]$. In the following we show how program restriction as well as semantic matching are actually appropriate abstractions of program semantics, in the abstract interpretation sense.

The process of considering only a portion of program semantics can be seen as an abstraction. Given a subset of a program P's labels (i.e., commands) $lab_r[\![P]\!] \subseteq lab[\![P]\!]$ that characterize a *restriction* of P, let $var_r[\![P]\!]$ and $Luse_r[\![P]\!]$ be, respectively, the set of variables occurring in the restriction and the set of memory locations it uses:

$$\begin{aligned} var_r[\![P]\!] &= \{var[\![C]\!] \mid lab[\![C]\!] \in lab_r[\![P]\!]\} \\ Luse_r[\![P]\!] &= \{Luse[\![C]\!] \mid lab[\![C]\!] \in lab_r[\![P]\!]\}. \end{aligned}$$

The set of labels $lab_r[\![P]\!]$ induces a restriction on environment and memory maps. Given $\rho \in \mathcal{E}(P)$ and $m \in \mathcal{M}(P)$, let $\rho^r = \rho_{|var_r[\![P]\!]}$ and $m^r = m_{|Luse_r[\![P]\!]}$ denote the restricted set of environments and memories induced by the restricted set of labels $lab_r[\![P]\!]$. Define $\alpha_r : \Sigma^* \to \Sigma^*$ that propagates restriction $lab_r[\![P]\!]$ on a given a trace σ:

$$\alpha_r(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ (C_1, (\rho_1^r, m_1^r))\alpha_r(\sigma') & \text{if } \sigma = (C_1, (\rho_1, m_1))\sigma' \\ & \text{ and } lab[\![C_1]\!] \in lab_r[\![P]\!] \\ \alpha_r(\sigma') & \text{otherwise} \end{cases}$$

Given a function $f : A \to B$ we denote, by a slight abuse of notation, its pointwise extension on powerset as $f : \wp(A) \to \wp(B)$, where $f(X) = \{f(x) | x \in X\}$. Note that the pointwise extension is additive. Therefore, the function $\alpha_r : \wp(\Sigma^*) \to \wp(\Sigma_r^*)$ is an abstraction that discards information outside the restriction $lab_r[\![P]\!]$. Moreover α_r is surjective and defines a Galois insertion: $\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_r}{\overset{\gamma_r}{\leftrightarrows}} \langle \wp(\Sigma_r^*), \subseteq \rangle$. Let $\alpha_r(\mathfrak{S}[\![P]\!])$ be the *restricted semantics* of program P. Observe that program behaviour is expressed by the effects that program execution has on environment and memory. Consider a transformation $\alpha_e : \Sigma^* \to \mathcal{X}^*$ that, given a trace σ, discards from σ all information about the commands that are executed, retaining only information about changes to the environment and effects on memory during execution:

$$\alpha_e(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \xi_1 \alpha_e(\sigma') & \text{if } \sigma = (C_1, \xi_1)\sigma' \end{cases}$$

Two traces are considered to be "similar" if they are the same under α_e, i.e., if they have the same sequence of effects on the restrictions of the environment and memory defined by $lab_r[\![P]\!]$. This semantic matching relation between program traces is the basis of our approach to malware detection. The additive function $\alpha_e : \wp(\Sigma^*) \to \wp(\mathcal{X}^*)$ abstracts from the trace semantics of a program and defines a Galois insertion: $\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} \langle \wp(\mathcal{X}^*), \subseteq \rangle$.

Let us say that a malware is a *vanilla malware* if no obfuscating transformations have been applied to it. The following definition provides a semantic characterization of the presence of a vanilla malware M in a program P in terms of the semantic abstractions α_r and α_e.

DEFINITION 4. *A program P is infected by a vanilla malware M, i.e., $M \hookrightarrow P$, if:*

$$\exists lab_r [\![P]\!] \in \wp(lab [\![P]\!]) : \alpha_e(\mathfrak{S} [\![M]\!]) \subseteq \alpha_e(\alpha_r(\mathfrak{S} [\![P]\!])).$$

A *semantic malware detector* is a system that verifies the presence of a malware in a program by checking the truth of the inclusion relation of the above definition.

4. Obfuscated Malware

To prevent detection malware writers usually obfuscate the malicious code. Thus, a robust malware detector needs to handle possibly obfuscated versions of a malware. While obfuscation may modify the original code, the obfuscated code has to be equivalent (up to some notion of equivalence) to the original one. Given an obfuscating transformation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ on programs and a suitable abstract domain A, we define an abstraction $\alpha_{\mathcal{O}} : \wp(\mathcal{X}^*) \to A$ that discards the details changed by the obfuscation while preserving the maliciousness of the program. Thus, different obfuscated versions of a program are equivalent up to $\alpha_{\mathcal{O}} \circ \alpha_e$. Hence, in order to verify program infection, we check whether there exists a semantic program restriction that matches the malware behaviour up to $\alpha_{\mathcal{O}}$, formally if:

$$\exists\ lab_r [\![P]\!] \in \wp(lab [\![P]\!]) : \\ \alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S} [\![M]\!])) \subseteq \alpha_{\mathcal{O}}(\alpha_e(\alpha_r(\mathfrak{S} [\![P]\!]))).$$

Here $\alpha_r(\mathfrak{S} [\![P]\!])$ is the restricted semantics for P; $\alpha_e(\alpha_r(\mathfrak{S} [\![P]\!]))$ retains only the environment-memory traces from the restricted semantics; and $\alpha_{\mathcal{O}}$ further discards any effects due to the obfuscation $\mathcal{O}$. We then check that the resulting set of environment-memory traces contains all of the environment-memory traces from the malware semantics, with obfuscation effects abstracted away via $\alpha_{\mathcal{O}}$.

4.1 Soundness vs Completeness

The extent to which a semantic malware detector is able to discriminate between infected and uninfected code, and therefore the balance between any false positives and any false negatives it may incur, depends on the abstraction function $\alpha_{\mathcal{O}}$. We can provide semantic characterizations of the notions of soundness and completeness, introduced in Definition 1, as follows:

DEFINITION 5. *A semantic malware detector on $\alpha_{\mathcal{O}}$ is complete for $\mathcal{O}$ if and only if*

$$\mathcal{O}(M) \hookrightarrow P \Rightarrow \left\{ \begin{array}{l} \exists lab_r [\![P]\!] \in \wp(lab [\![P]\!]) : \\ \alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S} [\![M]\!])) \subseteq \alpha_{\mathcal{O}}(\alpha_e(\alpha_r(\mathfrak{S} [\![P]\!]))) \end{array} \right. .$$

A semantic malware detector on $\alpha_{\mathcal{O}}$ is sound for $\mathcal{O}$ if and only if

$$\left. \begin{array}{l} \exists lab_r [\![P]\!] \in \wp(lab [\![P]\!]) : \\ \alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S} [\![M]\!])) \subseteq \alpha_{\mathcal{O}}(\alpha_e(\alpha_r(\mathfrak{S} [\![P]\!]))) \end{array} \right\} \Rightarrow \mathcal{O}(M) \hookrightarrow P.$$

It is interesting to observe that completeness is guaranteed when abstraction $\alpha_{\mathcal{O}}$ is preserved by obfuscation $\mathcal{O}$, namely when $\forall P \in \mathbf{P} : \alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S} [\![P]\!])) = \alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S} [\![\mathcal{O}(P)]\!]))$.

THEOREM 1. *If $\alpha_{\mathcal{O}}$ is preserved by the transformation $\mathcal{O}$ then the semantic malware detector on $\alpha_{\mathcal{O}}$ is complete for $\mathcal{O}$.*

However, the preservation condition of Theorem 1 is too weak to imply soundness of the semantic malware detector. As an example let us consider the abstraction $\alpha_{\top} = \lambda X.\top$ that loses all information. It is clear that $\alpha_{\top}$ is preserved by every obfuscating transformation, and the semantic malware detector on $\alpha_{\top}$ classifies every program as infected by every malware. Unfortunately we do not have a result analogous to Theorem 1 that provides a property of $\alpha_{\mathcal{O}}$ that characterizes soundness of the semantic malware detector.

4.2 A Semantic Classification of Obfuscations

Obfuscating transformations can be classified according to their effects on program semantics. Given $s, t \in A^*$ for some set A, let $s \preceq t$ denote that s is a subsequence of t, i.e., if $s = s_1 s_2 \ldots s_n$ then t is of the form $\ldots s_1 \ldots s_2 \ldots s_n \ldots$.

4.2.1 Conservative Obfuscations

An obfuscation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ is a *conservative obfuscation* if $\forall \sigma \in \mathfrak{S} [\![P]\!], \exists \delta \in \mathfrak{S} [\![\mathcal{O}(P)]\!]$ such that: $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Let $\mathbb{O}_c$ denote the set of conservative obfuscating transformations.

When dealing with conservative obfuscations we have that a trace δ of a program P presents a malicious behaviour M, if there is a malware trace $\sigma \in \mathfrak{S} [\![M]\!]$ whose environment-memory evolution is contained in the the environment-memory evolution of δ, namely if $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Let us define the abstraction $\alpha_c : \wp(\mathcal{X}^*) \to (\mathcal{X}^* \to \wp(\mathcal{X}^*))$ that given a sequence $s \in \mathcal{X}^*$ and a set $S \in \wp(\mathcal{X}^*)$, returns the elements $t \in S$ that are subtraces of s.

$$\alpha_c[S](s) \ = \ S \cap \mathit{SubSeq}(s)$$

where $\mathit{SubSeq}(s) = \{t | t \preceq s\}$ denotes the set of all subsequences of s. For any $S \in \wp(\mathcal{X}^*)$, the additive function α_c defines a Galois connection: $\langle \wp(\mathcal{X}^*), \subseteq \rangle \underset{\alpha_c[S]}{\overset{\gamma_c[S]}{\leftrightarrows}} \langle \wp(\mathcal{X}^*), \subseteq \rangle$. The abstraction α_c turns out to be a suitable approximation when dealing with conservative obfuscations. In fact the semantic malware detector on $\alpha_c[\alpha_e(\mathfrak{S} [\![M]\!])] \circ \alpha_e$ is complete and sound for the class of conservative obfuscations.

THEOREM 2. *If M is a vanilla malware and $\mathcal{O}_c \in \mathbb{O}_c$, then $\mathcal{O}_c(M) \hookrightarrow P$ iff $\exists\ lab_r [\![P]\!] \in \wp(lab [\![P]\!])$ such that:*

$$\alpha_c[\alpha_e(\mathfrak{S} [\![M]\!])](\alpha_e(\mathfrak{S} [\![M]\!])) \subseteq \\ \alpha_c[\alpha_e(\mathfrak{S} [\![M]\!])](\alpha_e(\alpha_r(\mathfrak{S} [\![P]\!]))).$$

Many obfuscating transformations commonly used by malware writers are conservative; a partial list of such conservative obfuscations is given below. It follows that Theorem 2 is applicable to a significant class of malware-obfuscation transformations.

– *Code reordering*. This transformation, commonly used to avoid signature matching detection, changes the order in which commands are written, while maintaining the execution order through the insertion of unconditional jumps.

– *Opaque predicate insertion*. This program transformation confuses the original control flow of the program by inserting opaque predicates, i.e., a predicate whose value is known a priori to a program transformation but is difficult to determine by examining the transformed program [7].

– *Semantic* NOP *insertion*. This transformation inserts commands that are irrelevant with respect to the program semantics.

– *Substitution of Equivalent Commands*. This program transformation replaces a single command with an equivalent one, with the goal of thwarting signature matching.

The following result shows that the composition of conservative obfuscations is a conservative obfuscation. Thus when more than one conservative obfuscation is applied, it can be handled as a single conservative obfuscation.

LEMMA 1. *Given $\mathcal{O}_1, \mathcal{O}_2 \in \mathbb{O}_c$ then $\mathcal{O}_1 \circ \mathcal{O}_2 \in \mathbb{O}_c$.*

EXAMPLE 1. *Let us consider a fragment of malware M presenting the decryption loop used by polymorphic viruses. Such a fragment writes, starting from memory location B, the decryption of memory locations starting at location A. Let $\mathcal{O}_c(M)$ be a conservative obfuscation of M:*

M	
L_1	$: \texttt{assign}(L_B, B) \to L_2$
L_2	$: \texttt{assign}(L_A, A) \to L_c$
L_c	$: cond(A) \to \{L_T, L_F\}$
L_T	$: B := Dec(A) \to L_{T_1}$
L_{T_1}	$: \texttt{assign}(\pi_2(B), B) \to L_{T_2}$
L_{T_2}	$: \texttt{assign}(\pi_2(A), A) \to L_C$
L_F	$: ...$

$\mathcal{O}_c(M)$	
L_1	$: \texttt{assign}(L_B, B) \to L_2$
L_2	$: \texttt{skip} \to L_4$
L_c	$: cond(A) \to \{L_O, L_F\}$
L_4	$: \texttt{assign}(L_A, A) \to L_5$
L_5	$: \texttt{skip} \to L_c$
L_O	$: P^T \to \{L_N, L_k\}$
L_N	$: X := X - 3 \to L_{N_1}$
L_{N_1}	$: X := X + 3 \to L_T$
L_T	$: B := Dec(A) \to L_{T_1}$
L_{T_1}	$: \texttt{assign}(\pi_2(B), B) \to L_{T_2}$
L_{T_2}	$: \texttt{assign}(\pi_2(A), A) \to L_c$
L_k	$: \ldots$
L_F	$: \ldots$

Given a variable X, the semantics of $\pi_2(X)$ is the label expressed by $\pi_2(m(\rho(X)))$, in particular $\pi_2(n) = \bot$, while $\pi_2(A, S) = S$. Given a variable X, let $Dec(X)$ denote the execution of a set of commands that decrypts the value stored in the memory location $\rho(X)$. The obfuscations are as follows: $L_2 : \texttt{skip} \to L_4$ and $L_5 : \texttt{skip} \to L_c$ are inserted by code reordering; $L_N : X := X + 3 \to L_{N_1}$ and $L_{N_1} : X := X - 3 \to L_T$ represent semantic nop insertion, and $L_O : P^T \to \{L_N, L_k\}$ true opaque predicate insertion. It can be shown that $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\mathfrak{S}[\![\mathcal{O}_c(M)]\!])) = \alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\mathfrak{S}[\![M]\!]))$, i.e., our semantics-based approach is able to see through the obfuscations and identify $\mathcal{O}(M)$ as matching the malware M.

4.2.2 Non-Conservative Obfuscations

A non-conservative transformation modifies the program semantics in such a way that the original environment-memory traces are not present any more. A possible way to face these transformations is to identify the set of all possible modifications induced by a non-conservative obfuscation, and fix, when possible, a *canonical* one. In this way the abstraction would reduce the original semantics to the canonical version before checking malware infection.

Another possible approach comes from Theorem 1 that states that if $\alpha_{\mathcal{O}}$ is preserved by $\mathcal{O}$ then the semantic malware detector on $\alpha_{\mathcal{O}}$ is complete w.r.t. $\mathcal{O}$. Recall that, given a program transformation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$, it is possible to systematically derive the most concrete abstraction $\alpha_{\mathcal{O}}$ preserved by $\mathcal{O}$ [12]. This systematic methodology can be used in presence of non-conservative obfuscations in order to derive a complete semantic malware detector when it is not easy to identify a canonical abstraction.

Moreover in Section 5 we show how it is possible to handle a class of non-conservative obfuscations through a further abstraction of the malware semantics.

In the following we consider a non-conservative transformation, known as *variable renaming*, and propose a canonical abstraction that leads to a sound and complete semantic malware detector.

Variable Renaming Variable renaming is a simple obfuscating transformation, often used to prevent signature matching, that replaces the names of variables with some different new names. Let $\mathcal{O}_v : \mathbf{P} \times \Pi \to \mathbf{P}$ denote the obfuscating transformation that, given a program P, renames its variables according to a mapping $\pi \in \Pi$, where $\pi : var[\![P]\!] \to Names$ is a bijective function that relates name of each program variable to its new name.

$$\mathcal{O}_v(P, \pi) = \left\{ C \,\middle|\, \exists C' \in P : act[\![C]\!] = act[\![C']\!]\,[X/\pi(X)] \right\}$$

where $A[X/\pi(X)]$ represents action A where each variable name X is replaced by $\pi(X)$. Recall that the matching relation between program traces considers the abstraction α_e of traces, thus it is interesting to observe that:

$$\alpha_e(\mathfrak{S}[\![\mathcal{O}_v(P, \pi)]\!]) = \left\{ \alpha_v[\pi](s) \,\middle|\, s \in \alpha_e(\mathfrak{S}[\![P]\!]) \right\}$$

where $\alpha_v : \Pi \to (\mathcal{X}^* \to \mathcal{X}^*)$ is defined as:

$$\alpha_v[\pi]((\rho_1, m_1)\ldots(\rho_n, m_n)) = (\rho_1 \circ \pi^{-1}, m_1)\ldots(\rho_n \circ \pi^{-1}, m_n).$$

In order to deal with variable renaming obfuscation we define the notion of *canonical variable renamings* $\widehat{\pi}$. Let $\{V_i\}_{i\in\mathbb{N}}$ be a set of canonical variable names. Given an environment-memory sequence $s \in \mathcal{X}^*$, the canonical renaming $\widehat{\pi}_s : var[\![s]\!] \to \{V_i\}_{i\in\mathbb{N}}$ renames the variables of s in such a way that the canonical name of the first variable in s is V_1, the canonical name of the second one is V_2, and so on. These canonical mappings $\widehat{\pi}$ have to be such that an environment-memory trace t is a renaming of an environment-memory trace s if and only if s and t have the same canonical form, namely $\alpha_v[\pi](s) = t \Leftrightarrow \alpha_v[\widehat{\pi}_s](s) = \alpha_v[\widehat{\pi}_s](t)$.

Note that program execution starts from the uninitialized environment $\rho_{uninit} = \lambda X.\bot$, and that each command assigns at most one variable X. Let $def(\rho)$ denote the set of variables that have defined (i.e., non-$\bot$) values in an environment ρ. This means that considering $s \in \alpha_e(\mathfrak{S}[\![P]\!])$, we have that $var[\![s]\!] = def(\rho_n)$, and $def(\rho_{i-1}) \subseteq def(\rho_i)$. Let us define $List(s)$ as the list of variables in s ordered according to their assignment time. Let $X : List(s)$ denote the insertion of a new variable X on the beginning of the list, and let $List(s)[i]$ denote the i-th element of the list. Formally, let $s = (\rho_1, m_1)(\rho_2, m_2)...(\rho_n, m_n) = (\rho_1, m_1)s'$:

$$List(s) = \begin{cases} \epsilon & \text{if } s = \epsilon \\ X : List(s') & \text{if } def(s_2) \smallsetminus def(s_1) = \{X\} \\ List(s') & \text{if } def(s_2) \smallsetminus def(s_1) = \varnothing \end{cases}$$

Thus the canonical renaming $\widehat{\pi}_s : var[\![s]\!] \to \{V_1 ... V_{|var[\![s]\!]|}\}$ of the environment sequence s is defined in function of $List(s)$ as:

$$\widehat{\pi}_s(X) = V_i \Leftrightarrow List(s)[i] = X \tag{1}$$

Thus, $\alpha_c[\widehat{\pi}_s] : \mathcal{X}^* \to \mathcal{X}_c^*$, where $\mathcal{X}_c$ denotes execution contexts where environments are defined on canonical variables, reduces $s \in \mathcal{X}^*$ to its canonical form. The following result shows that $\widehat{\pi}_s$ defined by Equation (1) is a canonical renaming.

LEMMA 2. *Given $s, t \in \mathcal{X}^\star$:*

$$\exists \pi : var[\![s]\!] \to var[\![t]\!] : \alpha_v[\pi](s) = t \Leftrightarrow \alpha_v[\widehat{\pi}_s](s) = \alpha_v[\widehat{\pi}_t](t).$$

Let $\widehat{\Pi}$ denote a set of canonical variable renaming and the additive function $\alpha_v : \widehat{\Pi} \to (\wp(\mathcal{X}^*) \to \wp(\mathcal{X}_c^*))$ is an approximation that abstracts from the names of variables. Thus, we have the following Galois connection: $\langle \wp(\mathcal{X}^*), \subseteq \rangle \underset{\alpha_v[\widehat{\Pi}]}{\overset{\gamma_v[\widehat{\Pi}]}{\leftrightarrows}} \langle \wp(\mathcal{X}_c^*), \subseteq \rangle$. In the following we show that the presence of a renamed malware $\mathcal{O}_v(M, \pi)$ in a program P can be semantically characterized in terms of the abstractions α_v and α_e. Define a variable renaming to be *stable* if, for each test action in the program, the i-th local variable of the *true* branch has the same name as the i-th local variable of the *false* branch. The following result states that the semantic malware detector on $\alpha_v[\widehat{\Pi}]$ is complete and sound for variable renaming.

THEOREM 3. *Given a stable renaming, $\mathcal{O}_v(M, \pi) \hookrightarrow P$ iff*

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_v[\widehat{\Pi}](\alpha_e(\mathfrak{S}[\![M]\!])) \subseteq \alpha_v[\widehat{\Pi}](\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))).$$

4.3 Composition

In general a malware uses multiple obfuscating transformations concurrently to prevent detection, therefore we have to consider the composition of conservative and non-conservative obfuscations, which is clearly not conservative. Let $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ be a non-conservative obfuscation, $\alpha_{\mathcal{O}}$ an abstraction such that the semantic malware detector on $\alpha_{\mathcal{O}}$ is sound and complete for $\mathcal{O}$. It is interesting to observe that if the abstraction $\alpha_{\mathcal{O}}$ preserves $\preceq$, namely if $(\alpha_e(\sigma) \preceq \alpha_e(\delta)) \Rightarrow \alpha_{\mathcal{O}}(\alpha_e(\sigma)) \preceq \alpha_{\mathcal{O}}(\alpha_e(\delta))$, then the semantic malware detector on $\alpha_c \circ \alpha_{\mathcal{O}}$ is complete for $\mathcal{O} \circ \mathcal{O}_c$ and $\mathcal{O}_c \circ \mathcal{O}$.

THEOREM 4. *If M is a vanilla malware, obfuscation $\mathcal{O}$ preserves $\preceq$, $\mathcal{O}_c \in \mathbb{O}_c$, and $\mathcal{O}(\mathcal{O}_c(M)) \hookrightarrow P$ or $\mathcal{O}_c(\mathcal{O}(M)) \hookrightarrow P$, then $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that*

$$\alpha_c[\alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S}[\![M]\!]))) \subseteq \alpha_c[\alpha_{\mathcal{O}}(\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_{\mathcal{O}}(\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])))).$$

EXAMPLE 2. *Let us consider $\mathcal{O}(\mathcal{O}_c(M),\pi)$ obtained by obfuscating the portion of malware M in Example 1 through variable renaming and some conservative obfuscations:*

$\mathcal{O}(\mathcal{O}_c(M),\pi)$
L_1 : $\mathtt{assign}(D, L_B) \rightarrow L_2$
L_2 : $\mathtt{skip} \rightarrow L_4$
L_c : $cond(E) \rightarrow \{L_O, L_F\}$
L_4 : $\mathtt{assign}(E, L_A) \rightarrow L_5$
L_5 : $\mathtt{skip} \rightarrow L_c$
L_O : $P^T \rightarrow \{L_T, L_k\}$
L_T : $D := Dec(E) \rightarrow L_{T_1}$
L_{T_1} : $\mathtt{assign}(\pi_2(D), D) \rightarrow L_{T_2}$
L_{T_2} : $\mathtt{assign}(\pi_2(E), E) \rightarrow L_c$
L_k : ...
L_F : ...

where $\pi(B) = D, \pi(A) = E$. It is possible to show that:

$$\alpha_c[\alpha_v[\widehat{\Pi}](\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_v[\widehat{\Pi}](\alpha_e(\mathfrak{S}[\![M]\!]))) \subseteq$$
$$\alpha_c[\alpha_v[\widehat{\Pi}](\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_v[\widehat{\Pi}](\alpha_e(\alpha_r(\mathfrak{S}[\![\mathcal{O}(\mathcal{O}_c(M),\pi)]\!])))).$$

5. Further Malware Abstractions

Definition 4 characterizes the presence of malware M in a program P as the existence of a restriction $lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that $\alpha_e(\mathfrak{S}[\![M]\!]) \subseteq \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))$. This means that program P is infected by malware M if P matches all malware behaviours. This notion of malware infection can be weakened in two different ways. First we can abstract the malware traces eliminating the states that are not relevant to determine maliciousness, and then check if program P matches this simplified behaviour. Second we can require program P to match a proper subset of malicious behaviours. Clearly that it is possible to combine the generalizations above. A deeper understanding of the malware behaviour is necessary in order to identify both the set of interesting states and the set of interesting behaviours.

Interesting States. Assume that we have an oracle that, given a malware M, returns the set of its interesting states. These states could be selected based on a security policy, for example, the states could represent network operations. This means that, in order to verify if P is infected by M, we have to check whether the malicious sequences of interesting states are present in P. Let us define the trace transformation $\alpha_{Int(M)} : \Sigma^* \rightarrow \Sigma^*$ that considers only the interesting states in a given trace $\sigma = \sigma_1\sigma'$:

$$\alpha_{Int(M)}(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \sigma_1\alpha_{Int(M)}(\sigma') & \text{if } \sigma_1 \in Int(M) \\ \alpha_{Int(M)}(\sigma') & \text{otherwise} \end{cases}$$

The following definition characterizes the presence of malware M in terms of its interesting states, i.e., through abstraction $\alpha_{Int(M)}$.

DEFINITION 6. *A program P is infected by a vanilla malware M with interesting states $Int(M)$, i.e., $M \hookrightarrow P$, if $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:*

$$\alpha_{Int(M)}(\mathfrak{S}[\![M]\!]) \subseteq \alpha_{Int(M)}(\alpha_r(\mathfrak{S}[\![P]\!])).$$

Thus we can weaken the standard notion of conservative transformation by saying that $\mathcal{O} : \mathbf{P} \rightarrow \mathbf{P}$ is *conservative w.r.t. $Int(M)$* if $\forall \sigma \in \mathfrak{S}[\![P]\!], \exists \delta \in \mathfrak{S}[\![\mathcal{O}(P)]\!]$ such that $\alpha_{Int(M)}(\sigma) = \alpha_{Int(M)}(\delta)$.

When program infection is characterized by Definition 6, the semantic malware detector on $\alpha_{Int(M)}$ is complete and sound for the obfuscating transformations that are conservative w.r.t. $Int(M)$.

THEOREM 5. *Let $Int(M)$ be the set of interesting states of a vanilla malware M, and let $\mathcal{O}$ be conservative w.r.t. $Int(M)$. Then $\mathcal{O}(M) \hookrightarrow P$ iff $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:*

$$\alpha_{Int(M)}(\mathfrak{S}[\![M]\!]) \subseteq \alpha_{Int(M)}(\alpha_r(\mathfrak{S}[\![P]\!])).$$

It is clear that transformations that are non-conservative may be conservative w.r.t. $Int(M)$, meaning that knowing the set of interesting states of a malware allows us to handle also some non-conservative obfuscations. For example the abstraction $\alpha_{Int(M)}$ allows the semantic malware detector to deal with reordering of independent instructions, as the following example shows.

EXAMPLE 3. *Let us consider the malware M and its obfuscation $\mathcal{O}(M)$ obtained by reordering independent instructions.*

M	$\mathcal{O}(M)$
$L_1 : A_1 \rightarrow L_2$	$L_1 : A_1 \rightarrow L_2$
$L_2 : A_2 \rightarrow L_3$	$L_2 : A_3 \rightarrow L_3$
$L_3 : A_3 \rightarrow L_4$	$L_3 : A_2 \rightarrow L_4$
$L_4 : A_4 \rightarrow L_5$	$L_4 : A_4 \rightarrow L_5$
$L_5 : A_5 \rightarrow L_6$	$L_5 : A_5 \rightarrow L_6$

In the above example A_2 and A_3 are independent, meaning that $\mathscr{A}[\![A_2]\!](\mathscr{A}[\![A_3]\!](\rho,m)) = \mathscr{A}[\![A_3]\!](\mathscr{A}[\![A_2]\!](\rho,m))$. Considering malware M, we have the trace $\sigma = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$ where:
- $\sigma_1 = \langle L_1 : A_1 \rightarrow L_2, (\rho, m)\rangle$,
- $\sigma_5 = \langle L_5 : A_5 \rightarrow L_6,$
$(\mathscr{A}[\![A_4]\!](\mathscr{A}[\![A_3]\!](\mathscr{A}[\![A_2]\!](\mathscr{A}[\![A_1]\!](\rho,m)))))\rangle$,
while considering the obfuscated version, we have the trace $\delta = \delta_1\delta_2\delta_3\delta_4\delta_5$, where:
- $\delta_1 = \langle L_1 : A_1 \rightarrow L_2, (\rho, m)\rangle$,
- $\delta_5 = \langle L_5 : A_5 \rightarrow L_6,$
$(\mathscr{A}[\![A_4]\!](\mathscr{A}[\![A_2]\!](\mathscr{A}[\![A_3]\!](\mathscr{A}[\![A_1]\!](\rho,m)))))\rangle$.
Let $Int(M) = \{\sigma_1, \sigma_5\}$. Then $\alpha_{Int(M)}(\sigma) = \sigma_1\sigma_5$ as well as $\alpha_{Int(M)}(\delta) = \delta_1\delta_5$, which concludes the example. It is obvious that $\delta_1 = \sigma_1$, moreover $\delta_5 = \sigma_5$ follows from the independence of A_2 and A_3.

Interesting Behaviours. Assume we have an oracle that given a malware M returns the set $T \subseteq \mathfrak{S}[\![M]\!]$ of its behaviours that characterize the maliciousness of M. Thus, in order to verify if P is infected by M, we check whether program P matches the malicious behaviours T. The following definition characterizes the presence of malware M in terms of its interesting behaviours T.

DEFINITION 7. *A program P is infected by a vanilla M with interesting behaviours $T \subseteq \mathfrak{S}[\![M]\!]$, i.e., $M \hookrightarrow P$ if:*

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha_e(T) \subseteq \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])).$$

It is interesting to observe that, when program infection is characterized by Definition 7, all the results obtained in Section 4 still hold if we replace $\mathfrak{S}[\![M]\!]$ with T.

6. Case Study: Completeness of Semantics-Aware Malware Detector $\mathcal{A}_{MD}$

An algorithm called *semantics-aware malware detection* was proposed by Christodorescu, Jha, Seshia, Song, and Bryant [4]. This approach to malware detection uses instruction semantics to identify malicious behavior in a program, even when obfuscated.

The obfuscations considered in [4] are from the set of conservative obfuscations, together with variable renaming. The paper proved the algorithm to be oracle-sound, so we focus in this section on proving its oracle-completeness using our abstraction-based framework. The list of obfuscations we consider (shown in Table 1) is based on the list described in the semantics-aware malware detection paper.

Obfuscation	*Completeness of* $\mathcal{A}_{MD}$
Code reordering	Yes
Semantic-nop insertion	Yes
Substitution of equivalent commands	No
Variable renaming	Yes

Table 1. List of obfuscations considered by the semantics-aware malware detection algorithm, and the results of our completeness analysis.

Description of the Algorithm The semantics-aware malware detection algorithm $\mathcal{A}_{MD}$ matches a program against a template describing the malicious behavior. If a match is successful, the program exhibits the malicious behavior of the template. Both the template and the program are represented as control-flow graphs during the operation of $\mathcal{A}_{MD}$.

The algorithm $\mathcal{A}_{MD}$ seeks to find a subset of the program P that matches the commands in the malware M, possibly after renaming of variables and locations used in the subset of P. Furthermore, $\mathcal{A}_{MD}$ checks that any def-use relationship that holds in the malware also holds in the program, across program paths that connect consecutive commands in the subset.

A control-flow graph $G = (V, E)$ is a graph with the vertex set V representing program commands, and edge set E representing control-flow transitions from one command to its successor(s). For our language the control-flow graph (CFG) can be easily constructed as follows:

- For each command $C \in \mathbf{C}$, create a CFG node annotated with that command, $v_{lab[\![C]\!]}$. Correspondingly, we write $C[\![v]\!]$ to denote the command at CFG node v.
- For each command $C = L_1 : C \to S$, where $S \in \wp(\mathbf{L})$, and for each label $L_2 \in S$, create a CFG edge (v_{L_1}, v_{L_2}).

Consider a path θ through the CFG from node v_1 to node v_k, $\theta = v_1 \to \ldots \to v_k$. There is a corresponding sequence of commands in the program P, written $P|_\theta = \{C_1, \ldots, C_k\}$. Then we can express the set of states possible after executing the sequence of commands $P|_\theta$ as $\mathscr{C}^k[\![P|_\theta]\!](\langle C_1, \rho, m\rangle)$, by extending the transition relation $\mathscr{C}$ to a set of states, such that $\mathscr{C} : \wp(\Sigma) \to \wp(\Sigma)$. Let us define the following basic functions:

$$mem[\![\langle C, \rho, m\rangle]\!] = m$$
$$env[\![\langle C, \rho, m\rangle]\!] = \rho$$

The algorithm takes as inputs the CFG for the template, $G^T = (V^T, E^T)$, and the binary file for the program, $File[\![P]\!]$. For each path θ in G^T, the algorithm proceeds in two steps:

1. Identify a one-to-one map from template nodes in the path θ to program nodes, $\mu_\theta : V^T \to V^P$.

   A template node n^T can match a program node n^P if the top-level operators in their actions are identical. This map induces a map $\nu_\theta : \mathbf{X}^T \times V^T \to \mathbf{X}^P$ from variables at a template node to variables at the corresponding program node, such that when renaming the variables in the template command $C[\![n^T]\!]$ according to the map ν_θ, we obtain the program command $C[\![n^P]\!] = C[\![n^T]\!][X/\nu_\theta(X, n^T)]$.

   This step makes use of the CFG oracle OR_{CFG} that returns the control-flow graph of a program P, given P's binary-file representation $File[\![P]\!]$.

2. Check whether the program preserves the def-use dependencies that are true on the template path θ.

   For each pair of template nodes m^T, n^T on the path θ, and for each template variable x^T defined in $act[\![C_m^T]\!]$ and used in $act[\![C_n^T]\!]$, let λ be a program path $\mu(v_1^T) \to \ldots \to \mu(v_k^T)$, where $m^T \to v_1^T \to \ldots \to v_k^T \to n^T$ is part of the path θ in the template CFG. λ is therefore a program path connecting the program CFG node corresponding to m^T with the program CFG node corresponding to n^T. We denote by $T|_\theta = \{C[\![m^T]\!], C_1^T, \ldots, C_k^T, C[\![n^T]\!]\}$ the sequence of commands corresponding to the template path θ.

   The def-use preservation check can be expressed formally as follows:

$$\forall \rho \in \mathcal{E}, \forall m \in \mathcal{M}, \forall s \in \mathscr{C}^k[\![P|_\lambda]\!]\left(\left\langle \mu_\theta\left(v_{C_1^T}\right), \rho, m\right\rangle\right) :$$
$$\mathscr{A}\left[\!\!\left[\nu_\theta\left(x^T, v_{C_1^T}\right)\right]\!\!\right](\rho, m) =$$
$$\mathscr{A}\left[\!\!\left[\nu_\theta\left(x^T, v_{C_n^T}\right)\right]\!\!\right](env[\![s]\!], mem[\![s]\!]).$$

   This check is implemented in $\mathcal{A}_{MD}$ as a query to a *semantic-nop oracle* OR_{SNop}. The semantic-nop oracle determines whether the value of a variable X before the execution of a code sequence $\psi \subseteq P$ is equal to the value of a variable Y after the execution of ψ.

The semantics-aware malware detector $\mathcal{A}_{MD}$ makes use of two oracles, OR_{CFG} and OR_{SNop}, described in Table 2. Thus $\mathcal{A}_{MD} = D^{\mathcal{OR}}$, for the set of oracles $\mathcal{OR} = \{OR_{CFG}, OR_{SNop}\}$. Our goal is then to show that $\mathcal{A}_{MD}$ is $\mathcal{OR}$-complete with respect to the obfuscations from Table 1. Since three of those obfuscations (code reordering, semantic-nop insertion, and substitution of equivalent commands) are conservative, we only need to prove $\mathcal{OR}$-completeness of $\mathcal{A}_{MD}$ for each individual obfuscation. We would then know (from Lemma 1) that $\mathcal{A}_{MD}$ is also $\mathcal{OR}$-complete with respect to any combination of these obfuscations.

Oracle	*Notation*
CFG oracle	$OR_{CFG}(File[\![P]\!])$
	Returns the control-flow graph of the program P, given its binary-file representation $File[\![P]\!]$.
Semantic-nop oracle	$OR_{SNop}(\psi, X, Y)$
	Determines whether the value of variable X before the execution of code sequence $\psi \subseteq P$ is equal to the value of variable Y after the execution of ψ.

Table 2. Oracles used by the semantics-aware malware detection algorithm $\mathcal{A}_{MD}$. Notation: $P \in \mathbf{P}, X, Y \in var[\![P]\!], \psi \subseteq P$.

We follow the proof strategy proposed in Section 2.1. First, in step 1 below, we develop a trace-based detector D_{Tr} based on an abstraction α, and show that $D^{\mathcal{OR}} = \mathcal{A}_{MD}$ and D_{Tr} are equivalent. This equivalence of detectors holds only if the oracles in $\mathcal{OR}$ are perfect. Then, in step 2, we show that D_{Tr} is complete w.r.t. the obfuscations of interest.

Step 1: Design an Equivalent Trace-Based Detector We can model the algorithm for semantics-aware malware detection using two abstractions, α_{SAMD} and α_{Act}. The abstraction α that characterizes the trace-based detector D_{Tr} is the composition of these two abstractions, $\alpha = \alpha_{Act} \circ \alpha_{SAMD}$. We will show that D_{Tr} is equivalent $\mathcal{A}_{MD} = D^{\mathcal{OR}}$, when the oracles in $\mathcal{OR}$ are perfect.

The abstraction α_{SAMD}, when applied to a trace $\sigma \in \mathfrak{S}[\![P]\!]$, $\sigma = \langle C'_1, \rho'_1, m'_1 \rangle \ldots \langle C'_n, \rho'_n, m'_n \rangle$, to a set of variable maps $\{\pi_i\}$, and a set of location maps $\{\gamma_i\}$, returns an abstract trace:

$$\begin{aligned} \alpha_{SAMD}(\sigma, \{\pi_i\}, \{\gamma_i\}) = \langle C_1, \rho_1, m_1 \rangle \ldots \langle C_n, \rho_n, m_n \rangle \\ \text{if } \forall i,\ 1 \le i \le n\ :\ & act[\![C_i]\!] = act[\![C'_i]\!][X/\pi_i(X)] \\ & \wedge\ lab[\![C_i]\!] = \gamma_i(lab[\![C'_i]\!]) \\ & \wedge\ suc[\![C_i]\!] = \gamma_i(suc[\![C'_i]\!]) \\ & \wedge\ \rho_i = \rho'_{j_i} \circ \pi_i \\ & \wedge\ m_i = m'_{j_i} \circ \gamma_i \end{aligned}$$

Otherwise, if the condition does not hold, $\alpha_{SAMD}(\sigma, \{\pi_i\}, \{\gamma_i\}) = \epsilon$. A map $\pi_i : var[\![P]\!] \to \mathbf{X}$ renames program variables such that they match malware variables, while a map $\gamma_i : lab[\![P]\!] \to \mathbf{L}$ reassigns program memory locations to match malware memory locations.

The abstraction α_{Act} simply strips all labels from the commands in a trace $\sigma = \langle C_1, \rho_1, m_1 \rangle \sigma'$, as follows:

$$\alpha_{Act}(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ \langle A_1, \rho_1, m_1 \rangle\, \alpha_{Act}(\sigma') & \text{otherwise} \end{cases}$$

Definition 8 (α-Semantic Malware Detector). *An α-semantic malware detector is a malware detector on the abstraction α, i.e., it evaluates the validity of the following for a program P and a malware M:*

$$\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]) : \alpha(\mathfrak{S}[\![M]\!]) \subseteq \alpha(\alpha_r(\mathfrak{S}[\![P]\!])).$$

By this definition, a semantic malware detector (from Definition 4) is a special instance of the α-semantic malware detector, for $\alpha = \alpha_e$. Then let D_{Tr} be a $(\alpha_{Act} \circ \alpha_{SAMD})$-semantic malware detector.

Proposition 1. *The semantics-aware malware detector algorithm $\mathcal{A}_{MD}$ is equivalent to the $(\alpha_{Act} \circ \alpha_{SAMD})$-semantic malware detector D_{Tr}. In other words, $\forall P, M \in \mathbf{P}$, we have that $\mathcal{A}_{MD}(P, M) = D_{Tr}(\mathfrak{S}[\![P]\!], \mathfrak{S}[\![M]\!])$.*

The proof has two parts, to show that $\mathcal{A}_{MD}(P, M) = 1 \Rightarrow D_{Tr}(\mathfrak{S}[\![P]\!], \mathfrak{S}[\![M]\!]) = 1$, and then to show the reverse. For the first implication, it is sufficient to show that for any path θ in the CFG of M and the path χ in the CFG of P, such that θ and χ are found as related by the algorithm $\mathcal{A}_{MD}$, the corresponding traces are equal when abstracted by $\alpha_{Act} \circ \alpha_{SAMD}$. The proof for the second implication proceeds by showing that any two traces $\sigma \in \mathfrak{S}[\![M]\!]$ and $\delta \in \mathfrak{S}[\![P]\!]$, that are equal when abstracted by $\alpha_{Act} \circ \alpha_{SAMD}$, have corresponding paths through the CFGs of M and P, respectively, such that these paths satisfy the conditions in the algorithm $\mathcal{A}_{MD}$. Both parts of the proof depend on the oracles used by $\mathcal{A}_{MD}$ to be perfect.

Now we can define the operation of the semantics-aware malware detector in terms of its effect on the trace semantics of a program P.

Definition 9 (Semantics-Aware Malware Detection). *A program P is a infected by a vanilla malware M, i.e. $M \hookrightarrow P$, if:*

$$\begin{aligned} \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!]),\ \{\pi_i\}_{i \ge 1},\ \{\gamma_i\}_{i \ge 1}\ : \\ \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \\ \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\})) \end{aligned}$$

Step 2: Prove Completeness of the Trace-Based Detector We are interested in finding out which classes of obfuscations are handled by a semantics-aware malware detector. We check the validity of the completeness condition expressed in Definition 5. In other words, if the program is infected with an obfuscated variant of the malware, then the semantics-aware detector should return 1.

Proposition 2. *A semantics-aware malware detector is complete on α_{SAMD} w.r.t. the code-reordering obfuscation $\mathcal{O}_J$:*

$$\mathcal{O}_J(M) \hookrightarrow P \Rightarrow \begin{cases} \exists lab_r[\![P]\!] \in \wp(lab[\![P]\!],\ \{\pi_i\}_{i \ge 1},\ \{\gamma_i\}_{i \ge 1} : \\ \quad \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \\ \qquad \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}[\![P]\!]), \{\pi_i\}, \{\gamma_i\})) \end{cases}$$

The code-reordering obfuscation inserts `skip` commands into the program and changes the labels of existing commands. The restriction α_r "eliminates" the inserted `skip` commands, while the α_{Act} abstraction allows for trace comparison while ignoring command labels. Thus, the detector D_{Tr} is $\mathcal{OR}$-complete w.r.t. the code-reordering obfuscation. Similar proofs confirm that D_{Tr} is $\mathcal{OR}$-complete w.r.t. variable renaming and semantic-nop insertion.

Proposition 3. *A semantics-aware malware detector is complete on α_{SAMD} w.r.t. the variable-renaming obfuscation $\mathcal{O}_v$.*

Proposition 4. *A semantics-aware malware detector is complete on α_{SAMD} w.r.t. the semantic-nop insertion obfuscation $\mathcal{O}_N$.*

Additionally, D_{Tr} is $\mathcal{OR}$-complete on α_{SAMD} w.r.t. a limited version of substitution of equivalent commands, when the commands in the original malware M are not substituted with equivalent commands.

Unfortunately, D_{Tr} is not $\mathcal{OR}$-complete w.r.t. all conservative obfuscation, as the following result illustrates.

Proposition 5. *A semantics-aware malware detector is not complete on α_{SAMD} w.r.t. all conservative obfuscations $\mathcal{O}_c \in \mathbb{O}_c$.*

The cause for this incompleteness is the fact that the abstraction applied by D_{Tr} still preserves some of the actions from the program. Thus, an obfuscation that affects at least one action on every path through the program, e.g., a well chosen instance of substitution of equivalent commands (see proof of Proposition 5 in the Appendix for an example), will defeat the detector.

7. Related Work

There is a considerable body of literature on existing techniques for malware detection: Szor gives an excellent summary [22].

Code obfuscation has been extensively studied in the context of protecting intellectual property. The goal of these techniques is to make reverse engineering of code harder [3, 6, 7, 11, 12, 18]. Cryptographers are also pursuing research on the question of possibility of obfuscation [1, 14, 24]. To our knowledge, we are not aware of existing research on formal approaches to obfuscation in the context of malware detection.

8. Conclusions and Future Work

Malware detectors have traditionally relied upon syntactic approaches, typically based on signature-matching. While such approaches are simple, they are easily defeated by obfuscations. To address this problem, this paper presents a semantics-based framework within which one can specify what it means for a malware detector to be sound and/or complete, and reason about the completeness of malware detectors with respect to various classes of obfuscations. As a concrete application, we have shown that a semantics-aware malware detector proposed by Christodorescu *et al.* is complete with respect to some commonly used malware obfuscations.

Given an obfuscating transformation $\mathcal{O}$, we assumed that the abstraction $\alpha_{\mathcal{O}}$ is provided by the malware detector designer. We are currently investigating how to design a systematic (ideally automatic) methodology for deriving an abstraction $\alpha_{\mathcal{O}}$ that leads to a sound and complete semantic malware detector. We observed that if the abstraction $\alpha_{\mathcal{O}}$ is preserved by the obfuscation $\mathcal{O}$ then the malware detection is complete, i.e. no false negatives. However, preservation is not enough to eliminate false positives. Hence, an interesting research task consists in characterizing the set of semantic abstractions that prevents false positives.

One further step would be to investigate whether and how model checking techniques can be applied to detect malware. Some works along this line already exist [17]. Observe that the abstraction $\alpha_{\mathcal{O}}$, actually defines a set of program traces that are equivalent up to $\mathcal{O}$. In model checking, sets of program traces are represented by formulae of some linear/branching temporal logic. Hence, we aim at defining a temporal logic whose formulae are able to express normal forms of obfuscations together with operators for composing them. This would allow to use standard model checking algorithms to detect malwares in programs. This could be a possible direction to follow in order to develop a practical tool for malware detection based on our semantic model. We expect this semantics-based tool to be significantly more precise than existing virus scanners.

References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology (CRYPTO'01)*, volume 2139 of *Lecture Notes in Computer Science*, pages 1 – 18. Springer-Verlag, Aug. 2001.

[2] D. Chess and S. White. An undetectable computer virus. In *Proc. of the 2000 Virus Bulletin Conference (VB2000)*, 2000.

[3] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proc. 4th. Information Security Conference (ISC 2001)*, Springer LNCS vol. 2000, pages 144–155, 2001.

[4] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proc. Usenix Security '05*, pages 32–46, Aug. 2005.

[5] F. B. Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6:22–35, 1987.

[6] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. 25th. ACM Symposium on Principles of Programming Languages (POPL 1998)*, pages 184–196, Jan. 1998.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[9] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, 1979.

[10] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 178–190, Jan. 2002.

[11] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Proc. of the 3rd IEEE International Conference on Software Engineeering and Formal Methods* (*SEFM '05*), pages 301–310, 2005.

[12] M. Dalla Preda and R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*, volume 3580 of *Springer LNCS*, pages 1325–1336, 2005.

[13] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. von Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61):published online at `http://www.phrack.org`. Last accessed: 16 Jan. 2004, Aug. 2003.

[14] S. Goldwasser and Y. T. Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 553–562, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*.

[16] M. Jordan. Dealing with metamorphism. *Virus Bulletin*, pages 4–6, Oct. 2002.

[17] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In K. Julisch and C. Krügel, editors, *Proc. of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187, Vienna, Austria, July 2005. Springer-Verlag.

[18] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM Press, Oct. 2003.

[19] P. Morley. Processing virus collections. In *Proc. of the 2001 Virus Bulletin Conference (VB2001)*, pages 129–134, Sept. 2001.

[20] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, Jan. 1997.

[21] Rajaat. Polymorphism. *29A Magazine*, 1(3), 1999.

[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. of the 2001 Virus Bulletin Conference (VB2001)*, pages 123 – 144, Prague, Czech Republic, Sept. 2001. Virus Bulletin.

[24] H. Wee. On obfuscating point functions. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532, New York, NY, USA, 2005. ACM Press.

[25] z0mbie. Automated reverse engineering: Mistfall engine. Published online at `http://z0mbie.host.sk/autorev.txt`. Last accessed: 16 Jan. 2004.

[26] z0mbie. RPME mutation engine. Published online at `http://z0mbie.host.sk/rpme.zip`. Last accessed: 16 Jan. 2004.

9. Appendix: Selected Proofs

Theorem 2

($\Rightarrow$) Completeness: If $\mathcal{O}_c(M) \hookrightarrow P$ it means that $\exists\ lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ such that $P_r = \mathcal{O}_c(M)$. Such restriction is the one that satisfies the condition on the right. In fact if $P_r = \mathcal{O}_c(M)$ it means that $\alpha_r(\mathfrak{S} \llbracket P \rrbracket) = \mathfrak{S} \llbracket \mathcal{O}_c(M) \rrbracket$. Thus we have to prove that $\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\mathfrak{S} \llbracket M \rrbracket)) \subseteq \alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\mathfrak{S} \llbracket \mathcal{O}_c(M) \rrbracket))$. By definition of conservative obfuscation for each trace $\sigma \in \mathfrak{S} \llbracket M \rrbracket$ there exists $\delta \in \mathfrak{S} \llbracket \mathcal{O}_c(M) \rrbracket$ such that: $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Considering such σ and δ we show that $\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\delta))$, in fact:

$$\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\delta)) = \alpha_e(\mathfrak{S} \llbracket M \rrbracket) \cap SubSeq(\alpha_e(\delta))$$
$$\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\sigma)) = \alpha_e(\mathfrak{S} \llbracket M \rrbracket) \cap SubSeq(\alpha_e(\sigma)).$$

Since $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ then $SubSeq(\alpha_e(\sigma)) \subseteq SubSeq(\alpha_e(\delta))$. Therefore $\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\delta))$, which concludes the proof.
($\Leftarrow$) Soundness: By hypothesis there exists $lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ $\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\mathfrak{S} \llbracket M \rrbracket)) \subseteq \alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))$. This means that $\forall \sigma \in \mathfrak{S} \llbracket M \rrbracket$ we have that: $\alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))$. By hypothesis $\sigma \in \mathfrak{S} \llbracket M \rrbracket$, therefore $\alpha_e(\sigma) \in \alpha_c[\alpha_e(\mathfrak{S} \llbracket M \rrbracket)](\alpha_e(\sigma))$, which means that $\alpha_e(\sigma) \in \left\{\alpha_c(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))(\alpha_e(\delta)) \mid \delta \in \alpha_r(\mathfrak{S} \llbracket P \rrbracket)\right\}$. Thus

there exists $\delta \in \alpha_r(\mathfrak{S}\llbracket P \rrbracket)$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and this means that that P_r is a conservative obfuscation of malware M.

Lemma 2

($\Rightarrow$) Let $s = (\rho_1^s, m_1)...(\rho_n^s, m_n)$ and $t = (\rho_1^t, m_1)...(\rho_n^t, m_n)$ such that $t = \alpha_v[\pi](s)$, with $\pi : var\llbracket s \rrbracket \rightarrow var\llbracket t \rrbracket$, then $\forall i \in [1, n] : def(\rho_i^t) = \{\pi(X) | X \in def(\rho_i^s)\}$, thus $|def(\rho_i^t)| = |def(\rho_i^s)|$. By definition of $List$ we have that: $List(s)[i] = X \Leftrightarrow List(t)[i] = \pi(X)$, which means that $\pi_s^c(X) = V_i \Leftrightarrow \pi_t^c(\pi(X)) = V_i$. This implies that $\alpha_v[\pi_s^c](s) = \alpha_v[\pi_t^c](t)$.
($\Leftarrow$) Let $\alpha_v[\pi_s^c](s) = \alpha_v[\pi_t^c](t) = (\rho_1^c, m_1)...(\rho_n^c, m_n)$, thus $|var\llbracket s \rrbracket| = |var\llbracket t \rrbracket| = k$. By definition $\pi_s^c : var\llbracket s \rrbracket \rightarrow \{V_1...V_k\}$ and $\pi_t^c : var\llbracket t \rrbracket \rightarrow \{V_1...V_k\}$. Let $\pi = \pi_t^{c^{-1}} \circ \pi_s^c : var\llbracket s \rrbracket \rightarrow var\llbracket t \rrbracket$. π is a bijection because is a composition of bijective functions. We show that $\alpha_v[\pi](s) = t$. In fact $\alpha_v[\pi](s) = t$ iff $\forall i$: $\rho_i^s(X) = \rho_i^t(\pi(X))$ which holds since $\rho_i^t(\pi(X)) = \rho_i^t(\pi_t^{c^{-1}} \circ \pi_s^c(X)) = \rho_i^c(\pi_t^c \circ \pi_t^{C^{-1}} \circ \pi_s^c(X)) = \rho_i^c(\pi_s^c(X)) = \rho_i^s(X)$.

Theorem 3

Given a stable renaming, for each test action the i-th local variable of the *true* branch has the same name of the i-th local variable of the *false* branch. In particular considering two paths $path_1 = A \rightarrow B \rightarrow C$ and $path_2 = A \rightarrow D \rightarrow C$ of the CGF of P. Let $Loc\llbracket B \rrbracket = \{X | X \in var\llbracket B \rrbracket, X \notin var\llbracket A \rrbracket \cup var\llbracket C \rrbracket\}$ be the set of local variables of B enumerated following their definition order. Let $n = min\{|Loc\llbracket B \rrbracket|, |Loc\llbracket D \rrbracket|\}$ then: $\forall i \in [1, n]$ the i-th variable of $Loc\llbracket B \rrbracket$ has the same name of the i-th variable of $Loc\llbracket D \rrbracket$.
($\Rightarrow$) Completeness: If $\mathcal{O}_v(M, \pi) \hookrightarrow P$, then exists $lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket)$ such that $P_r = \mathcal{O}_v(M, \pi)$, therefore $\alpha_r(\mathfrak{S}\llbracket P \rrbracket) = \mathfrak{S}\llbracket \mathcal{O}_v(M, \pi) \rrbracket$. This restriction is the one that satisfies the condition on the right, in fact we show: $\alpha_v[\Pi^c](\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \subseteq \alpha_v[\Pi^c](\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_v(M, \pi) \rrbracket))$. From definition of $\mathcal{O}_v$ we have that:

$$\begin{aligned}
& \alpha_e(\mathfrak{S}\llbracket \mathcal{O}_v(M, \pi) \rrbracket) = \{\alpha_v[\pi](s) | s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket)\} \\
\Leftrightarrow \quad & \forall s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket), \exists t \in \alpha_e(\mathfrak{S}\llbracket \mathcal{O}_v(M, \pi) \rrbracket) : t = \alpha_v[\pi](s) \\
\Leftrightarrow \quad & \text{(Lem2) } \forall s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket), \exists t \in \alpha_e(\mathfrak{S}\llbracket \mathcal{O}_v(M, \pi) \rrbracket) : \\
& \alpha_v[\pi_s^c](s) = \alpha_v[\pi_t^c](t) \\
\Leftrightarrow \quad & \alpha_v[\Pi^c](\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) = \alpha_v[\Pi^c](\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_v(M, \pi) \rrbracket)).
\end{aligned}$$

($\Leftarrow$) Soundness: Observe that:

$$\begin{aligned}
& \alpha_v[\Pi^c](\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \subseteq \alpha_v[\Pi^c](\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))) \\
\Leftrightarrow \quad & \{\alpha_v[\pi_s^c](s) \mid s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket)\} \subseteq \\
& \{\alpha_v[\pi_t^c](t) \mid t \in \alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))\} \\
\Leftrightarrow \quad & \forall s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket), \exists t \in \alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)) : \\
& \alpha_v[\pi_s^c](s) = \alpha_v[\pi_t^c](t) \\
\Leftrightarrow \quad & \text{(Lem2) } \forall s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket), \exists t \in \alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)) : \\
& \exists \pi_{s,t} : var\llbracket s \rrbracket \rightarrow var\llbracket t \rrbracket : \alpha_v[\pi_{s,t}](s) = t.
\end{aligned}$$

It is clear that $var\llbracket M \rrbracket = \bigcup_{s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket)} var\llbracket s \rrbracket$ and $var\llbracket P_r \rrbracket = \bigcup_{t \in \alpha_e(\alpha_R(\mathfrak{S}\llbracket P \rrbracket))} var\llbracket t \rrbracket$. Let us define $\pi : var\llbracket M \rrbracket \rightarrow var\llbracket P \rrbracket$ as follows: $\pi(X) = \{Y | \exists s \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket), \exists t \in \alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)) : \pi_{s,t}(X) = Y\}$. Let us show that π is a function. In fact if we consider $s, q \in \alpha_e(\mathfrak{S}\llbracket M \rrbracket)$ and $t, r \in \alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))$ such that $\alpha_v[\pi_s^c](s) = \alpha_v[\pi_t^c](t)$ and $\alpha_v[\pi_q^c](q) = \alpha_v[\pi_r^c](r)$, then $\forall X \in var\llbracket s \rrbracket \cap var\llbracket q \rrbracket : \pi_{s,t}(X) = \pi_{q,r}(X)$. Let s, q be environment instances of respectively $path_1 = A \rightarrow B \rightarrow C$ and $path_2 = A \rightarrow D \rightarrow C$ of the CFG of M, and t, r of respectively $\hat{path}_1 = \hat{A} \rightarrow \hat{B} \rightarrow \hat{C}$ and $\hat{path}_2 = \hat{A} \rightarrow \hat{D} \rightarrow \hat{C}$ of the CFG of P_r. Then we have two possible cases: (1) if $X \in var\llbracket s \rrbracket \cap var\llbracket q \rrbracket$ and $X \in var\llbracket A \rrbracket \cup var\llbracket C \rrbracket$ then $\pi(X) \in var\left\llbracket \hat{A} \right\rrbracket \cup var\left\llbracket \hat{C} \right\rrbracket$ and $\pi_{s,t}(X) = \pi_{q,r}(X)$ is guaranteed by the fact that X is present in a common part of the two paths. (2) $X \in var\llbracket s \rrbracket \cap var\llbracket q \rrbracket$ and if $X \in Loc\llbracket B \rrbracket \cap Loc\llbracket D \rrbracket$ then $\pi(X) \in Loc\left\llbracket \hat{B} \right\rrbracket \cap Loc\left\llbracket \hat{D} \right\rrbracket$ and the assumption of a stable renaming guarantees that $\pi_{s,t}(X) = \pi_{q,r}(X)$.

Theorem 4

(1) When $\mathcal{O}(\mathcal{O}_c(M)) \hookrightarrow P$ then $\exists lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket)$ such that $P_r = \mathcal{O}(\mathcal{O}_c(M))$, therefore $\alpha_r(\mathfrak{S}\llbracket P \rrbracket) = \mathfrak{S}\llbracket \mathcal{O}(\mathcal{O}_c(M)) \rrbracket$. Thus, we have to show that:

$$\begin{aligned}
& \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))) \subseteq \\
& \qquad \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}(\mathcal{O}_c(M)) \rrbracket))).
\end{aligned}$$

$\alpha_\mathcal{O}$ is sound by hypothesis, therefore:

$$\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_c(M) \rrbracket)) \subseteq \alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}(\mathcal{O}_c(M)) \rrbracket)).$$

α_c is monotone therefore:

$$\begin{aligned}
& \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_c(M) \rrbracket))) \subseteq \\
& \qquad \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}(\mathcal{O}_c(M)) \rrbracket))).
\end{aligned}$$

Thus, we only have to prove that:

$$\begin{aligned}
& \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))) \subseteq \\
& \qquad \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_c(M) \rrbracket))).
\end{aligned}$$

By definition of conservative transformation we have that $\forall \sigma \in \mathfrak{S}\llbracket M \rrbracket, \exists \delta \in \mathfrak{S}\llbracket \mathcal{O}_c(M) \rrbracket$ such that if $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. For such σ and δ, we show that:

$$\begin{aligned}
& \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\sigma))) \subseteq \\
& \qquad \alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\delta))).
\end{aligned}$$

By definition we have that $\alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\sigma))) = \alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \cap SubSeq(\alpha_\mathcal{O}(\alpha_e(\sigma))$ and $\alpha_c[\alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))](\alpha_\mathcal{O}(\alpha_e(\delta))) = \alpha_\mathcal{O}(\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \cap SubSeq(\alpha_\mathcal{O}(\alpha_e(\delta)))$.
Since $\alpha_\mathcal{O}(\alpha_e(\sigma)) \preceq \alpha_\mathcal{O}(\alpha_e(\delta))$, then $SubSeq(\alpha_\mathcal{O}(\alpha_e(\sigma))) \subseteq SubSeq(\alpha_\mathcal{O}(\alpha_e(\delta)))$, and this concludes the proof.
Proof for (2) is similar to point (1).

Proposition 1

To show that $\mathcal{A}_{MD} = D$, we can equivalently show that $\forall P, M \in \mathbf{P} : \mathcal{A}_{MD}(P, M) = 1 \iff \exists lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket), \exists\{\pi_i\}_{i \geq 1}$, and $\exists\{\gamma_i\}_{i \geq 1}$ such that $\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket M \rrbracket, \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}\llbracket P \rrbracket), \{\pi_i\}, \{\gamma_i\}))$. Since π_i renames variables only from P (i.e., $\forall V \in \mathbf{V} \setminus var\llbracket P \rrbracket$, π_i is the identity function, $\pi_i(X) = X$), and similarly γ_i remaps locations only from P, then we have that $\alpha_{SAMD}(\mathfrak{S}\llbracket M \rrbracket, \{\pi_i\}, \{\gamma_i\}) = \mathfrak{S}\llbracket M \rrbracket$.

($\Rightarrow$) We know that $\mathcal{A}_{MD}(P, M) = 1$. We can construct the restriction $lab_r\llbracket P \rrbracket$ from the path-sensitive map μ_θ as follows:

$$lab_r\llbracket P \rrbracket = \bigcup_{\theta \in Paths\left(G^M\right)} \left\{ lab\left\llbracket C\left\llbracket \mu\left(v^M\right)\right\rrbracket\right\rrbracket : v^M \in \theta \right\}$$

The variable maps $\{\pi_i\}$ can be defined based on ν_θ. For a path $\theta = v_1^M \rightarrow \ldots \rightarrow v_k^M$, $\pi_i(X) = \nu_\theta\left(X, v_i^M\right)$. Similarly, $\gamma_i(L) = L'$ if $lab\llbracket C\llbracket v_i^M \rrbracket\rrbracket = L'$ and $lab\llbracket C\llbracket \mu_\theta\left(v_i^M\right)\rrbracket\rrbracket = L$.
Let $\sigma \in \mathfrak{S}\llbracket M \rrbracket$ and denote by $\theta = v_1^M \rightarrow \ldots \rightarrow v_k^M$ the CFG path corresponding to this trace. By algorithm $\mathcal{A}_{MD}$, there exists a path χ in the CFG of P of the form:

$$\ldots \rightarrow C\left\llbracket \mu\left(v_1^M\right)\right\rrbracket \rightarrow \ldots \rightarrow C\left\llbracket \mu\left(v_k^M\right)\right\rrbracket \rightarrow \ldots$$

Let $\delta \in \mathfrak{S}\llbracket P \rrbracket$ be the trace corresponding to the path χ in G^P, $\delta = \ldots\langle C\llbracket \mu\left(v_1^M\right)\rrbracket, \rho_1^P, m_1^P\rangle \ldots \langle C\llbracket \mu\left(v_k^M\right)\rrbracket, \rho_k^P, m_k^P\rangle \ldots$. For two states i and $j > i$ of the trace σ, denote the intermediate states in the trace δ by $\langle C_1'^P, \rho_1'^P, m_1'^P\rangle \ldots \langle C_l'^P, \rho_l'^P, m_l'^P\rangle$, i.e.: $\delta = \ldots \langle C\llbracket\mu(v_i^M)\rrbracket, \rho_i^P, m_i^P\rangle \langle C_1'^P, \rho_1'^P, m_1'^P\rangle \ldots \langle C_l'^P, \rho_l'^P, m_l'^P\rangle \langle C\llbracket\mu(v_j^M)\rrbracket, \rho_j^P, m_j^P\rangle \ldots$. From step 1 of algorithm $\mathcal{A}_{MD}$, we have that the following hold:

$$act\left\llbracket C\left\llbracket\mu\left(v_i^M\right)\right\rrbracket\right\rrbracket[X/\pi_i(X)] = act\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket$$
$$\gamma_i\left(lab\left\llbracket C\left\llbracket\mu\left(v_i^M\right)\right\rrbracket\right\rrbracket\right) = lab\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket$$
$$\gamma_i\left(suc\left\llbracket C\left\llbracket\mu\left(v_i^M\right)\right\rrbracket\right\rrbracket\right) = suc\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket$$

From step 2 of algorithm $\mathcal{A}_{MD}$, we know that for any template variable x^M that is defined in $C\llbracket v_i^M\rrbracket$ and used in $C\llbracket v_j^M\rrbracket$ (for $1 \leq i < j \leq k$), we have that $\mathscr{A}\llbracket\nu(x^M, v_i^M)\rrbracket(\rho, m) = \mathscr{A}\llbracket\nu(x^M, v_j^M)\rrbracket(env\llbracket s\rrbracket, mem\llbracket s\rrbracket)$, where $s \in \mathscr{C}^l(\langle\mu(v_i^M)\rangle, \rho, m)$. As $act\llbracket C\llbracket\mu(v_i^M)\rrbracket\rrbracket[X/\pi_i(X)] = act\llbracket C\llbracket v_i^M\rrbracket\rrbracket$, it follows that $\rho_i^P(\nu(x^M, v_i^M)) = \rho_j^P(\nu(x^M, v_j^M))$. Since $\rho_i^M(x^M) = \rho_j^M(x^M)$, then we can write $\rho_i^M = \rho_i^P \circ \pi_i$. Similarly, $m_i^M = m_i^P \circ \pi_i$. Then it follows that:

$$\begin{aligned}\alpha_{Act}(\alpha_{SAMD}(\sigma, \{\pi_i\}, \{\gamma_i\})) &= \alpha_{Act}(\sigma)\\ &= \alpha_{Act}(\alpha_{SAMD}(\delta, \{\pi_i\}, \{\gamma_i\})).\end{aligned}$$

Therefore, $\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket M\rrbracket, \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket P\rrbracket, \{\pi_i\}, \{\gamma_i\}))$.

($\Leftarrow$) We know that $lab_r\llbracket P\rrbracket$, $\{\pi_i\}_{i\geq 1}$, and $\{\gamma_i\}_{i\geq 1}$ exist such as to satisfy the RHS of the logical equivalence. We will show that $\mathcal{A}_{MD}$ returns 1 in such a case, that is, the two steps of the algorithm complete successfully.

Let $\sigma \in \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket M\rrbracket, \{\pi_i\}, \{\gamma_i\}))$, with

$$\sigma = \left\langle A_1, \rho_1^M, m_1^M\right\rangle \ldots \left\langle A_k, \rho_k^M, m_k^M\right\rangle.$$

Then there exists $\sigma' \in \mathfrak{S}\llbracket M\rrbracket$

$$\sigma' = \left\langle C_1^M, \rho_1^M, m_1^M\right\rangle \ldots \left\langle C_k^M, \rho_k^M, m_k^M\right\rangle,$$

such that $\forall i,\ act\llbracket C_i^M\rrbracket[X/\pi_i(X)] = A_i$. Similarly, there exists $\delta \in \alpha_r(\mathfrak{S}\llbracket P\rrbracket)$, $\delta = \langle C_1^P, \rho_1^P, m_1^P\rangle \ldots \langle C_k^P, \rho_k^P, m_k^P\rangle$, such that $\forall i,\ act\llbracket C_i^P\rrbracket[X/\pi_i(X)] = A_i$, $\rho_i^P = \rho_i^M \circ \pi_i^{-1}$, and $m_i^P = m_i^M \circ \gamma_i^{-1}$. In other words, $\sigma = \alpha_{Act}(\alpha_{SAMD}(\sigma', \{\pi_i\}, \{\gamma_i\})) = \alpha_{Act}(\alpha_{SAMD}(\delta', \{\pi_i\}, \{\gamma_i\}))$, where σ' is a malware trace and δ' is a trace of the restricted program P_r induced by $lab_r\llbracket P\rrbracket$. For each pair of traces (σ, δ) chosen as above, we can define a map μ from nodes in the CFG of M to nodes in the CFG of P by setting $\mu\left(v_{lab\llbracket C_i^M\rrbracket}\right) = v_{lab\llbracket C_i^P\rrbracket}$. Without loss of generality, we assume that $lab\llbracket M\rrbracket \cap lab\llbracket P\rrbracket = \emptyset$. Then μ is a one-to-one, onto map, and step 1 of algorithm $\mathcal{A}_{MD}$ is complete.

Consider a variable $x^M \in var\llbracket M\rrbracket$ that is defined by action A_i and later used by action A_j in the trace σ', for $j > i$, such that $\rho_{i+1}^M(x^M) = \rho_j^M(x^M)$. Let x_i^P be the program variable corresponding to x^M at program command C_i^P, and x_j^P the program variable corresponding to x^M at program command C_j^P:

$$x_i^P = \nu\left(x^M, v_{lab\llbracket C_i^M\rrbracket}\right) \qquad\qquad x_j^P = \nu\left(x^M, v_{lab\llbracket C_j^M\rrbracket}\right)$$

If $\delta \in \alpha_r(\mathfrak{S}\llbracket P\rrbracket)$, then there exists a $\delta' \in \mathfrak{S}\llbracket P\rrbracket$ of the form:

$$\delta' = \ldots\left\langle C_i^P, \rho_i^P, m_i^P\right\rangle \ldots \left\langle C_j^P, \rho_j^P, m_j^P\right\rangle \ldots$$

where $1 \leq i < j \leq k$. Let θ be a path in the CFG of P, $\theta = v_1^P \to \ldots \to v_k^P$, such that $v_{lab\llbracket C_i^P\rrbracket}^P \to v_1^P \to \ldots \to v_k^P \to v_{lab\llbracket C_j^P\rrbracket}^P$ is also a path in the CFG of P. Since $\rho_{i+1}^M(x^M) = \rho_j^M(x^M)$, then $\rho_{suc\llbracket C_i^P\rrbracket}^P(x_i^P) = \rho_{i+1}^M(\pi_i(x_i^P)) = \rho_{i+1}^M(x^M) = \rho_j^M(x^M) = \rho_j^P(\pi_j(x_j^P)) = \rho_j^P(x_j^P)$. But $suc\llbracket C_i^P\rrbracket = lab\llbracket C^P\llbracket v_1\rrbracket\rrbracket$ in the trace δ'. As $\mathscr{A}\llbracket x_i^P\rrbracket(\rho, m) = \rho(x_i^P)$, it follows that

$$\begin{aligned}&\mathscr{A}\left\llbracket\nu\left(x^M, v_{lab\llbracket C_i^M\rrbracket}\right)\right\rrbracket(\rho, m) =\\ &\qquad\qquad \mathscr{A}\left\llbracket\nu\left(x^M, v_{lab\llbracket C_j^M\rrbracket}\right)\right\rrbracket(env\llbracket s\rrbracket, mem\llbracket s\rrbracket)\end{aligned}$$

for any $\rho \in \mathcal{E}$, any $min\mathcal{M}$, and any state s of P at the end of executing the path θ, i.e., $s \in \mathscr{C}^k\llbracket P|_\theta\rrbracket\left(\left\langle\mu\left(v_{lab\llbracket C_i^P\rrbracket}^P\right), \rho, m\right\rangle\right)$. If the semantic-nop oracle queried by $\mathcal{A}_{MD}$ is complete, then the second step of the algorithm is successful. Thus $\mathcal{A}_{MD}(P, M) = 1$.

Proposition 2

If $\mathcal{O}_J(M) \hookrightarrow P$, and given that $\mathcal{O}_J$ inserts only `skip` commands into a program, then $\exists lab_r\llbracket P\rrbracket \in \wp(lab\llbracket P\rrbracket)$ such that $P_r = \mathcal{O}_J(M) \setminus Skip$, where $Skip$ is a set of `skip` commands inserted by $\mathcal{O}_J$. Let $M' = \mathcal{O}_J(M) \setminus Skip$. Then $\alpha_r(\mathfrak{S}\llbracket P\rrbracket) = \mathfrak{S}\llbracket M'\rrbracket$. Thus we have to prove that

$$\begin{aligned}&\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket M\rrbracket, \{\pi_i\}, \{\gamma_i\})) \subseteq\\ &\qquad\qquad \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket M'\rrbracket, \{\pi_i\}, \{\gamma_i\})),\end{aligned}$$

for some $\{\pi_i\}$ and $\{\gamma_i\}$. As $\mathcal{O}_J(M)$ does not rename variables or change memory locations, we can set π_i and γ_j, for all i and j, to be the respective identity maps, $\pi_i = Id_{var\llbracket P\rrbracket}$ and $\gamma_j = Id_{lab\llbracket P\rrbracket}$. It follows that $\alpha_{SAMD}(\mathfrak{S}\llbracket M'\rrbracket, \{Id_{var\llbracket P\rrbracket}\}, \{Id_{lab\llbracket P\rrbracket}\}) = \mathfrak{S}\llbracket M'\rrbracket$ and $\alpha_{SAMD}(\mathfrak{S}\llbracket M\rrbracket, \{Id_{var\llbracket P\rrbracket}\}, \{Id_{lab\llbracket P\rrbracket}\}) = \mathfrak{S}\llbracket M\rrbracket$. It remains to show that $\alpha_{Act}(\mathfrak{S}\llbracket M\rrbracket) \subseteq \alpha_{Act}(\mathfrak{S}\llbracket M'\rrbracket)$. By the definition of $\mathcal{O}_J$, we have that $M' = \mathcal{O}_J(M) \setminus Skip = (M \setminus S) \cup \eta(S)$, for some $S \subset M$. But $\eta(S)$ only updates the labels of the commands in S, and thus we have:

$$\begin{aligned}\alpha_{Act}(\mathfrak{S}\llbracket M'\rrbracket) &= \alpha_{Act}(\mathfrak{S}\llbracket(M \setminus S) \cup \eta(S)\rrbracket)\\ &= \alpha_{Act}(\mathfrak{S}\llbracket M\rrbracket).\end{aligned}$$

It follows that $\alpha_{Act}(\mathfrak{S}\llbracket M\rrbracket) \subseteq \alpha_{Act}(\mathfrak{S}\llbracket\mathcal{O}_J(M) \setminus Skip\rrbracket)$.

Proposition 5

To prove that semantics-aware malware detection is not complete on α_{SAMD} w.r.t. all conservative obfuscations, it is sufficient to find one conservative obfuscation such that

$$\begin{aligned}&\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\llbracket M\rrbracket, \{\pi_i\}, \{\gamma_i\})) \subseteq\\ &\qquad \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}\llbracket\mathcal{O}_c(M)\rrbracket), \{\pi_i\}, \{\gamma_i\})) \qquad (2)\end{aligned}$$

cannot hold for any restriction $lab_r\llbracket\mathcal{O}_c(M)\rrbracket \in \wp(lab\llbracket\mathcal{O}_c(M)\rrbracket)$ and any maps $\{\pi_i\}_{i\geq 1}$ and $\{\gamma_i\}_{i\geq 1}$.

Consider an instance of the substitution of equivalent commands obfuscating transformation $\mathcal{O}_I$ that substitutes the action of at least one command for each path through the program (i.e., $\mathfrak{S}\llbracket P\rrbracket \cap \mathfrak{S}\llbracket\mathcal{O}_I(P)\rrbracket = \emptyset$)—for example, the transformation could modify the command at the start label of the program. Assume that $\exists\{\pi_i\}_{i\geq 1}$ and $\exists\{\gamma_i\}_{i\geq 1}$ such that Equation 2 holds, where $\mathcal{O}_c = \mathcal{O}_I$. Then $\exists\sigma \in \mathfrak{S}\llbracket M\rrbracket$ and $\exists\delta \in \mathfrak{S}\llbracket\mathcal{O}_I(M)\rrbracket$ such that $\alpha_{Act}(\sigma) = \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\delta), \{\pi_i\}, \{\gamma_i\}))$. As $|\sigma| = |\delta|$, we have that $\alpha_r(\delta) = \delta$. If $\sigma = \ldots\langle C_i, \rho_i, m_i\rangle\ldots$ and $\delta = \ldots\langle C_i', \rho_i', m_i'\rangle\ldots$, then we have that $\forall i, act\llbracket C_i\rrbracket = act\llbracket C_i'\rrbracket[X/\pi_i(X)]$. But from the definition of the obfuscating transformation $\mathcal{O}_I$ above, we know that $\forall\sigma \in \mathfrak{S}\llbracket M\rrbracket$, $\forall\delta \in \mathfrak{S}\llbracket\mathcal{O}_I(M)\rrbracket$, $\exists i \geq 1$ such that $C_i \in \sigma$, $C_i' \in \delta$, and $\forall\pi : \mathbf{X} \to \mathbf{X}$, $act\llbracket C_i\rrbracket \neq act\llbracket C_i'\rrbracket[X/\pi(X)]$. Hence we have a contradiction.