A Semantics-Based Approach to Malware Detection

MILA DALLA PREDA
University of Verona, dallapre@sci.univr.it
and
MIHAI CHRISTODORESCU and SOMESH JHA
University of Wisconsin, Madison, {mihai,jha}@cs.wisc.edu
and
SAUMYA DEBRAY
University of Arizona, Tucson, debray@cs.arizona.edu

Malware detection is a crucial aspect of software security. Current malware detectors work by checking for *signatures*, which attempt to capture the syntactic characteristics of the machine-level byte sequence of the malware. This reliance on a syntactic approach makes current detectors vulnerable to code obfuscations, increasingly used by malware writers, that alter the 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 these detectors. Our approach uses a trace semantics to characterize the behavior of malware as well as that of 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 (1) standard signature matching detection schemes are generally sound but not complete, (2) the semantics-aware malware detector proposed by Christodorescu *et al.* is complete with respect to a number of common obfuscations used by malware writers and (3) the malware detection scheme proposed by Kinder *et al.* and based on standard model-checking techniques is sound in general and complete on some, but not all, obfuscations handled by the semantics-aware malware detector.

Categories and Subject Descriptors: F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs. —*Malware Detection*

General Terms: Security, Languages, Theory, Verification

The work of M. Dalla Preda was partially supported by the MUR grant "InterAbstract" and by the FIRB grant "Abstract Interpretation and Model Checking for the verification of embedded systems".

The work of M. Christodorescu and S. Jha was supported in part by the National Science Foundation under grants CNS-0448476 and CNS-0627501.

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

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.

Additional Key Words and Phrases: malware detection, obfuscation, trace semantics, abstract interpretation

1. INTRODUCTION

The term *malware* (*mal*icious soft*ware*) refers to a program with malicious intent designed to damage the machine on which it executes or the network over which it communicates. The growth in the complexity of modern computing systems makes it difficult, if not impossible, to avoid bugs. This increases the possibility of malware attacks that usually exploit such vulnerabilities in order to damage the system. Furthermore, as the size and complexity of a system grows, it becomes more difficult to analyze it and prove that it is not infected. Thus, the threat of malware attacks is an unavoidable problem in computer security, and therefore *it is crucial to detect the presence of malicious code in software systems*.

A considerable body of literature on malware detection techniques exists – Ször provides an excellent summary [Ször 2005]. *Misuse detection*, also called *signature-based detection*, represents one of the most popular approaches to malware detection. This detection scheme is based on the assumption that malware can be described through patterns (also called signatures). In fact, a misuse detection scheme classifies a program P as infected by a malware when the malware signature – namely the sequence of instructions characterizing a malware – occurs in P [Morley 2001; Ször 2005]. In general, signature-based algorithms detect known malware, but are ineffective against unknown malicious programs, since no signature is available for them. To tackle this limitation, anti-virus companies strive to update the signature lists as often as possible. Thanks to their low false positive rate and ease of use, misuse detectors are widely used.

Malware writers resort to sophisticated hiding techniques, often based on code obfuscation, in order to avoid misuse detection [Nachenberg 1997]. In particular, recent developments in malware technology have led to the so-called *metamorphic malware*. The basic idea of metamorphism is that each successive generation of a malware changes the syntax while leaving the semantics almost unchanged in order to foil misuse detection systems. Thus, it is not surprising that hackers often use code obfuscation in order to automatically generate metamorphic malware. In fact, it is possible to design obfuscations that transform a malicious program, either manually or automatically, by inserting new code or modifying existing code in order to make detection harder while preserving the malicious behavior. If a signature describes a certain sequence of instructions [Ször 2005], then those instructions can be reordered or replaced with equivalent instructions [z0mbie 2001b; 2001a]. Such obfuscations are especially applicable on CISC architectures, such as the Intel IA-32 [Intel Corporation 2001], 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 [Ször and Ferrie 2001; Jordan 2002; z0mbie 2001b] 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 [Rajaat 1999; Detristan et al. 2003] can effectively hide the relevant data. Of course, attackers have the choice of creating new malware from scratch, but that does not appear to be a favored tactic [Symantec Corporation 2006]. *Therefore, an important requirement of a robust malware detection technique is to handle obfuscating* *transformations.*

The reason why obfuscation can easily foil signature matching lies in the syntactic nature of this approach that ignores program functionality. In fact, code obfuscation changes the malware syntax but not its intended behavior, which has to be preserved. Formal methods for program analysis, such as semantics-based static analysis and model checking, could be useful in designing more sophisticated malware detection algorithms that are able to deal with obfuscated versions of the same malware. For example, in [Christodorescu et al. 2005] the authors put forward a semantics-aware malware detector that is able to handle some of the obfuscations commonly used by hackers, while in [Kinder et al. 2005] the authors introduce an extension of the CTL temporal logic, which is able to express some malicious properties that can be used to detect malware through standard model checking algorithms. These preliminary works confirm the potential benefits of a formal approach to malware detection.

We believe that addressing the malware-detection problem from a semantic point of view could lead to a more robust detection system. In fact, different obfuscated versions of the same malware have to share (at least) the malicious intent, namely the maliciousness of their semantics, even if they might express it through different syntactic forms. The goal of this work 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. In particular, this work makes the following contributions:

- A formal definition of what it means for a malware detector to be sound and complete with respect to a class of obfuscations.
- A framework for proving that a detector is complete and/or sound with-respect-to a class of obfuscations.
- A trace semantics to characterize the program and malware behaviors, using abstract interpretation to "hide" irrelevant aspects of these behaviors.
- A series of case studies to evaluate the power of our formal framework by proving soundness and completeness of some well known detection schemes.
  - Signature-based detection is proven to be generally sound but not complete.
  - Semantics-aware malware detection proposed in [Christodorescu et al. 2005] is proven to be complete with respect to some common obfuscations used by malware writers (soundness is proved in [Christodorescu et al. 2005]).
  - The model checking-based detection scheme proposed in [Kinder et al. 2005] is proved to be sound in general and complete on some, but not all, obfuscations handled by the semantics-aware malware detector.

The results presented in this work are an extended and reviewed version of [Dalla Preda et al. 2007].

2. OVERVIEW

The precision of a malware detector is usually expressed in terms of soundness and completeness properties. In the following we formally define what it means for a malware detector to be sound and complete with respect to a class of obfuscations. Moreover, we provide an informal description of a proof strategy that can be used to certify soundness and completeness of existing detection schemes.

Following a standard definition, an *obfuscating transformation* $\mathcal{O} : \mathbf{P} \rightarrow \mathbf{P}$ is a potent program transformer that preserves the functionality of programs [Collberg et al. 1998], where potent means that the transformed program is more complex, i.e., more difficult to understand, than the original one. Let $\mathbf{O}$ denote the set of all obfuscating transformations. A *malware detector* can be seen as a function $D : \mathbf{P} \times \mathbf{P} \rightarrow \{0, 1\}$ that, given a program P and a malware M, decides if program P is infected with malware M. For example, $D(P, M) = 1$ means that P is infected with M or with an obfuscated variant of M. Our treatment of malware detectors focuses 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.

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

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.

Many malware detectors are built on top of other static-analysis techniques for problems that are hard or undecidable. For example, malware detectors that are based on static analysis [Kinder et al. 2005; Christodorescu et al. 2005] assume that the control-flow graph for an executable can be extracted. As shown by researchers [Linn and Debray 2003], simply disassembling an executable can be quite tricky. Therefore, we want to 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. 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. Soundness (resp. completeness) with respect to perfect oracles is also called *oracle-soundness* (resp. *oracle-completeness*).

DEFINITION 3. A malware detector $D^{\mathcal{OR}}$ is *oracle-complete* with respect to an obfuscation $\mathcal{O}$, if $D^{\mathcal{OR}}$ is complete for that obfuscation $\mathcal{O}$ when all oracles in the set $\mathcal{OR}$ are perfect. *Oracle-soundness* of a detector $D^{\mathcal{OR}}$ can be defined in a similar manner.

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, both current and future. A malware writer who has access

[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. These types of relative completeness and soundness results are common in cryptography.

to the detection algorithm and to its inner workings can use such knowledge in order to design ad-hoc obfuscation tools to bypass detection. As the malware detection problem is in general undecidable, for any given malware detector it is always possible to design an obfuscation transformation that defeats that detector. 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 formal framework for proving soundness and completeness of malware detectors in the presence of obfuscations. The basic idea is to describe programs through their execution traces—thus, program trace semantics is the building block of our framework. In Section 4 and Section 5 we describe how both obfuscations and detectors can be elegantly expressed as operations on traces. In this setting it is interesting to consider soundness and completeness of malware detectors with respect to classes of transformations that share similar effects on program trace semantics. For this reason we introduce a classification of obfuscation techniques based on the effects these transformations have on program trace semantics and we investigate soundness and completeness of malware detectors with respect to these families of obfuscations (see Section 6 and Section 7).

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.
Consider a malware detector $D^{\mathcal{OR}}$ that uses a set of oracles $\mathcal{OR}$. Let $\mathfrak{S}[\![P]\!]$ and $\mathfrak{S}[\![M]\!]$ denote the trace semantics of program P and malware M respectively. Describe a detector D_{Tr} that works in the semantic world of traces and classifies a program P as infected by a malware M if $\mathfrak{S}[\![P]\!]$ matches certain properties of $\mathfrak{S}[\![M]\!]$. Prove that if the oracles in $\mathcal{OR}$ are perfect, then the two detectors are equivalent, i.e., for all P and M in $\mathbf{P}$, $D^{\mathcal{OR}}(P, M) = 1$ iff $D_{Tr}(\mathfrak{S}[\![P]\!], \mathfrak{S}[\![M]\!]) = 1$. 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.
We are now ready to 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's effects on trace semantics are compared to the effects of obfuscation on 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 $D^{\mathcal{OR}}$.

Observe that the proof strategy presented above works under the assumption that the set of oracles $\mathcal{OR}$ used by the detector $D^{\mathcal{OR}}$ are perfect. In fact, the equivalence of the semantic malware detector D_{Tr} to the detection algorithm $D^{\mathcal{OR}}$ is stated and proved under the hypothesis of perfect oracles. This means that when the oracles in $\mathcal{OR}$ are perfect then:

—$D^{\mathcal{OR}}$ is sound w.r.t. obfuscation $\mathcal{O} \Leftrightarrow D_{Tr}$ is sound w.r.t. obfuscation $\mathcal{O}$
—$D^{\mathcal{OR}}$ is complete w.r.t. obfuscation $\mathcal{O} \Leftrightarrow D_{Tr}$ is complete w.r.t. obfuscation $\mathcal{O}$

Consequently, the proof of soundness/completeness of D_{Tr} with respect to a given obfuscation $\mathcal{O}$ implies soundness/completeness of $D^{\mathcal{OR}}$ with respect to obfuscation $\mathcal{O}$ and vice versa. However, even when the oracles used by the detection scheme $D^{\mathcal{OR}}$ are not perfect it is possible to deduce some properties of $D^{\mathcal{OR}}$ by analyzing its semantic counterpart D_{Tr}. Let D_{Tr} denote the semantic malware detection algorithm which is equivalent to the detection scheme $D^{\mathcal{OR}}$ working on perfect oracles. In general, by relaxing the hypothesis of perfect oracles, we have that the malware detector $D^{\mathcal{OR}}$ is less precise than its (ideal) semantic counterpart D_{Tr}. This means that:

—$D^{\mathcal{OR}}$ is sound w.r.t. obfuscation $\mathcal{O} \Rightarrow D_{Tr}$ is sound w.r.t. obfuscation $\mathcal{O}$
—$D^{\mathcal{OR}}$ is complete w.r.t. obfuscation $\mathcal{O} \Rightarrow D_{Tr}$ is complete w.r.t. obfuscation $\mathcal{O}$

In this case, by proving that D_{Tr} is not sound/complete with respect to a given obfuscation $\mathcal{O}$ we prove as well that $D^{\mathcal{OR}}$ is not sound/complete with respect to $\mathcal{O}$. On the other hand, even if we are able to prove that D_{Tr} is sound/complete with respect to an obfuscation $\mathcal{O}$ we cannot say anything about the soundness/completeness of $D^{\mathcal{OR}}$ with respect to $\mathcal{O}$.

3. PRELIMINARIES

3.1 Abstract Interpretation

The basic idea of abstract interpretation is that program behavior at different levels of abstraction is an approximation of its formal semantics [Cousot and Cousot 1979; 1977]. The (concrete) semantics of a program is computed on the (concrete) domain $\langle C, \leq_C \rangle$, i.e., a complete lattice modeling the values computed by programs. The partial ordering $\leq_C$ models relative precision: $c_1 \leq_C c_2$ means that c_1 is more precise (concrete) than c_2. Approximation is encoded by an abstract domain $\langle A, \leq_A \rangle$, i.e., a complete lattice, that represents some approximation properties on concrete objects. Also in the abstract domain the ordering relation $\leq_A$ denotes relative precision. As usual, abstract domains are specified by Galois connections [Cousot and Cousot 1979; 1977]. Two complete lattices C and A form a Galois connection (C, α, γ, A), also denoted $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, when the functions $\alpha : C \rightarrow A$ 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 α (γ) is the left (right) adjoint of γ (α). α and γ are called, respectively, abstraction and concretization maps. Given a Galois connection, the abstraction map can be uniquely determined using the concretization map and vice versa [Cousot and Cousot 1992]. In fact $\forall c \in C$ we can define the function $\alpha(c) = \bigwedge\{a \in A | c \leq_C \gamma(a)\}$, while $\forall a \in A$ we can define the function $\gamma(a) = \bigvee\{c \in C | \alpha(c) \leq_A a\}$. This means that α maps each element $c \in C$ in the smallest element in A whose image by γ is greater than c as regards $\leq_C$. On the other side, γ maps each element $a \in A$ in the greatest element in C whose image by α is lower than a as regards $\leq_A$. A tuple (C, α, γ, A) is a Galois 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, denoted $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$, if each element of A is an abstraction of a concrete element in C, namely (C, α, γ, A) is a Galois insertion iff α is

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)

Syntax:

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

Fig. 1. A simple programming language.

surjective iff γ is injective. Abstract domains can be related to each other with respect to their relative degree of precision. In particular, we say that an abstraction $\alpha_1 : C \to A_1$ is more concrete than $\alpha_2 : C \to A_2$, i.e., A_2 is more abstract than A_1, denoted $\alpha_1 \sqsubseteq \alpha_2$ or $A_1 \sqsubseteq A_2$, if $\forall c \in C : \gamma_1(\alpha_1(c)) \leq_C \gamma_2(\alpha_2(c))$.

3.2 Programming Language

The language we consider is a simple extension of the one introduced in [Cousot and Cousot 2002], 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 Fig. 1 and Fig. 2 (Fig. 3 provides some auxiliary functions used in the definitions of the semantics). Given a set S, we use $S_\perp$ to denote the set $S \cup \{\perp\}$, where $\perp$ represents an undefined value.[2] Commands can be either conditional or unconditional. A conditional command at a label L has the form '$L : B \to \{L_T, L_F\}$,' 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 label of the command to be executed next. A variable can be undefined ($\perp$), or it can store either an integer or a (appropriately encoded) pair $(A, S) \in \mathbf{A} \times \wp(\mathbf{L})$. Let $\mathscr{C}^n(C, \xi) = \mathscr{C}(\mathscr{C}^{n-1}(C, \xi))$ denote the fact that the semantic function $\mathscr{C}$ has been applied n times starting from state (C, ξ), where function $\mathscr{C}$ is extended to sets of states, $\mathscr{C}(S) = \bigcup_{\sigma \in S} \mathscr{C}(\sigma)$. Let $\mathscr{C}^*$ denote the closure of $\mathscr{C}$. A program consists of an initial set of commands together with all the commands that are reachable through execution from the initial set. In other words, if P_{init}

[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.

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)

Semantics:

ARITHMETIC EXPRESSIONS

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

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

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

COMMANDS

The semantic function $\mathscr{C} : \Sigma \to \wp(\Sigma)$ 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 $lab[\![L : B \to \{L_T, L_F\}]\!] = L$.

$$\mathscr{C}[\![L : A \to L']\!]\,\xi = \{(C, \xi') \mid \xi' = \mathscr{A}[\![A]\!]\,\xi, lab[\![C]\!] = L', \langle act[\![C]\!] : suc[\![C]\!]\rangle = m'(L')\} \qquad \text{where } \xi' = (\rho', m')$$
$$\mathscr{C}[\![L : B \to \{L_T, L_F\}]\!]\,\xi = \{(C, \xi) \mid lab[\![C]\!] = \begin{cases} L_T & \text{if } \mathscr{B}[\![B]\!]\,\xi = true \\ L_F & \text{if } \mathscr{B}[\![B]\!]\,\xi = false \end{cases}\}$$

Fig. 2. Semantics for our simple programming language of Fig. 1.

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 COMMAND

$suc\,[\![L : A \to L']\!] \ = \ L'$

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

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 occurring 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 occurring in $A\} \cup \rho(var\,[\![A]\!])$

COMMANDS IN SEQUENCES OF PROGRAM STATES

$cmd\,[\![\{(C_1, \xi_1), \ldots, (C_k, \xi_k)\}]\!] = \{C_1, \ldots, C_k\}$

Fig. 3. Auxiliary functions for the language of Fig. 1.

denotes the initial set of commands, then $P = cmd\left[\!\!\left[\bigcup_{C \in P_{init}} \left(\bigcup_{\xi \in \mathcal{X}} \mathscr{C}^*(C, \xi)\right)\right]\!\!\right]$. Since each command explicitly mentions its successors, a program does not need to maintain an explicit sequence of commands. This definition allows us to represent programs that generate code dynamically.

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 initial commands of P).

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 $\mathscr{C}\,[\![P]\!] : \Sigma(P) \to \wp(\Sigma(P))$ on program P as:

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

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 fix point 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 use $f_{|S}$ to denote the restriction of function f to elements in $S \cap A$, and $f \smallsetminus S$ to denote the restriction of function f to elements not in S, namely to $A \smallsetminus S$.

4. 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. Therefore, in order to detect the presence of a malicious behavior from a malware M in a program P, we need to check whether there is a part (a restriction) of $\mathfrak{S}\llbracket P \rrbracket$ that "matches" (in a sense that will be made precise) $\mathfrak{S}\llbracket M \rrbracket$. In the following we show how *program restriction* as well as *semantic matching* can be expressed as appropriate abstractions of program semantics, in the abstract interpretation sense.

Program restriction. Given a program P, the process of considering only a portion of program semantics can be clearly seen as an abstraction of $\mathfrak{S}\llbracket P \rrbracket$. In particular, a subset of a program P's labels (i.e., commands) $lab_r\llbracket P \rrbracket \subseteq lab\llbracket P \rrbracket$ characterizes a *restriction* of program P. In this setting, let $var_r\llbracket P \rrbracket$ and $Luse_r\llbracket P \rrbracket$ denote, respectively, the set of variables occurring and the set of memory locations used in the restriction:

$$\begin{aligned} var_r\llbracket P \rrbracket &= \bigcup \{var\llbracket C \rrbracket \mid lab\llbracket C \rrbracket \in lab_r\llbracket P \rrbracket\} \\ Luse_r\llbracket P \rrbracket &= \bigcup \{Luse\llbracket C \rrbracket \mid lab\llbracket C \rrbracket \in lab_r\llbracket P \rrbracket\}. \end{aligned}$$

This means that the set $lab_r\llbracket P \rrbracket$ of labels induces a restriction on environment and memory maps. In particular, given $\rho \in \mathcal{E}(P)$ and $m \in \mathcal{M}(P)$, let $\rho^r = \rho_{|var_r\llbracket P \rrbracket}$ and $m^r = m_{|Luse_r\llbracket P \rrbracket}$ denote the restricted set of environments and memories induced by the subset $lab_r\llbracket P \rrbracket$ of labels, and let $\Sigma_r = \big\{(C, (\rho^r, m^r)) \,\big|\, lab\llbracket C \rrbracket \in lab_r\llbracket P \rrbracket\big\}$ be the set of restricted program states. Given a trace $\sigma \in \mathfrak{S}\llbracket P \rrbracket$, let us define the abstraction $\alpha_r : \Sigma^* \to \Sigma^*$ that propagates the program restriction specified by $lab_r\llbracket P \rrbracket$ on the trace $\sigma = (C_1, (\rho_1, m_1))\sigma'$:

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

In fact, from the above definition, $\alpha_r(\sigma)$ corresponds exactly to the subsequence of σ given by the states in Σ_r. In the following, 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 $\alpha_r : \wp(\Sigma^*) \to \wp(\Sigma_r^*)$ defines an abstraction of sets of traces (i.e., program semantics) that discards information outside restriction $lab_r\llbracket P \rrbracket$. Moreover, α_r is surjective and defines a Galois insertion:

$$\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_r}{\overset{\gamma_r}{\leftrightarrows}} \langle \wp(\Sigma_r^*), \subseteq \rangle$$

Given a program P and a restriction $lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket)$, let $\alpha_r(\mathfrak{S}\llbracket P \rrbracket)$ be the *restricted semantics* of program P and let $P_r = \{C \in P \mid lab\llbracket C \rrbracket \in lab_r\llbracket P \rrbracket\}$ be the program obtained by considering only the commands of P with labels in $lab_r\llbracket P \rrbracket$. If P_r is a program, namely if it is possible to compute its semantics, then $\mathfrak{S}\llbracket P_r \rrbracket(I) = \alpha_r(\mathfrak{S}\llbracket P \rrbracket)$, where I is the set of possible program states that P can assume when it executes the first command of P_r.

Semantic matching. Let us observe that the effects of program execution on the execution context, i.e., on environments and memories, express program behavior more than the particular sequence of commands that cause such effects (in fact different sequences of commands may produce the same sequence of modifications on environments and memories). Thus, the idea is to define a (semantic) matching relation between traces based on execution contexts rather than commands. Let us consider the 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 the execution context:

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

Two traces σ and δ are considered to be "similar" if they are indistinguishable with respect to α_e, namely if they have the same sequence of effects on environments and memories, i.e., if $\alpha_e(\sigma) = \alpha_e(\delta)$. This *semantic matching* relation between program traces is the basis of our approach to malware detection, since it allows us to abstract from program syntax and concentrate on program behaviors. The additive function $\alpha_e : \wp(\Sigma^*) \to \wp(\mathcal{X}^*)$ defines the following Galois insertion:

$$\langle \wp(\Sigma^*), \subseteq \rangle \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} \langle \wp(\mathcal{X}^*), \subseteq \rangle$$

It follows that abstraction α_r models program restriction, while abstraction α_e models the semantic matching relation between program traces. In this contest, a malware is called 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 abstractions α_r and α_e.

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

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

Following this definition we have that a program P is infected by malware M, when it exhibits behaviors that, under abstractions α_r and α_e, match all of the behaviors of the vanilla malware M. It is clear that this is a strong requirement and that the notion of program infection can be weakened in many ways (see Section 7.2 for some examples).

In general, we say that 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 Definition 4.

5. OBFUSCATED MALWARE

Since malware writers usually obfuscate malicious code in order to prevent detection, 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, our idea is to define an abstraction $\alpha : \wp(\mathcal{X}^*) \to A$ that discards the details changed by the obfuscation while preserving the maliciousness of the program. Thus, the trace semantics of different obfuscated versions of a program are equivalent up to $\alpha \circ \alpha_e$. Hence, in order to verify program infection, we check whether there exists a semantic program restriction that matches the malware semantics up to $\alpha \circ \alpha_e$, formally $M \hookrightarrow P$ if:

$$\exists\ lab_r\, [\![P]\!] \in \wp(lab\, [\![P]\!]) : \alpha(\alpha_e(\mathfrak{S}\, [\![M]\!])) \subseteq \alpha(\alpha_e(\alpha_r(\mathfrak{S}\, [\![P]\!]))). \qquad (1)$$

Here $\alpha_r(\mathfrak{S}\, [\![P]\!])$ is the restricted semantics for program P; α_e denotes the abstraction that retains only the environment-memory traces; and α is the abstraction that further discards any effects due to obfuscation $\mathcal{O}$. Then, the above condition checks whether the abstraction of the restricted program semantics matches the abstract malware semantics, with obfuscation effects abstracted away via α.

In this setting, abstraction α allows us to ignore obfuscation and focus only on the malicious intent. A semantic malware detector on α refers to a semantic detection scheme that verifies infection according to equation (1).

EXAMPLE 1. Let us consider the fragment of program P that computes the factorial of variable X and its obfuscation $\mathcal{O}[\![P]\!]$ obtained inserting commands that do not affect the execution context (at labels L_2 and L_{F+1} in the example).

P	$\mathcal{O}[\![P]\!]$
$L_1 : F := 1 \to L_2$	$L_1 : F := 1 \to L_2$
$L_2 : (X = 1) \to \{L_T, L_F\}$	$L_2 : F := F \times 2 - F \to L_3$
$L_F : X := X - 1 \to L_{F+1}$	$L_3 : (X = 1) \to \{L_T, L_F\}$
$L_{F+1} : F := F \times X \to L_2$	$L_F : X := X - 1 \to L_{F+1}$
$L_T : ...$	$L_{F+1} : X := X \times 1 \to L_{F+2}$
	$L_{F+2} : F := F \times X \to L_3$
	$L_T : ....$

It is clear that $\mathbf{A}[\![F := F \times 2 - F]\!]\xi = \xi$ and $\mathbf{A}[\![X := X \times 1]\!]\xi = \xi$ for all $\xi \in \mathcal{X}$. Thus, an abstraction α able to deal with the insertion of such semantic nop commands, is the one that observes modifications in the execution context, formally let $\xi_i = (\rho_i, m_i)$:

$$\alpha(\xi_1, \xi_2, ..., \xi_n) = \begin{cases} \epsilon & \text{if } \xi_1, \xi_2, ..., \xi_n = \epsilon \\ \alpha(\xi_2, ..., \xi_n) & \text{if } \xi_1 = \xi_2 \\ \xi_1 \alpha(\xi_2, ..., \xi_n) & \text{otherwise} \end{cases}$$

In fact it is possible to show that $\alpha(\alpha_e(\mathfrak{S}[\![P]\!])) = \alpha(\alpha_e(\alpha_r[\![\mathcal{O}[\![P]\!]]\!]))$.

5.1 Soundness vs Completeness

According to the notion of infection of equation (1) we have that the extent to which a semantic malware detector on α 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 α. On the one hand, by augmenting the degree of abstraction of α we increase the ability of the detector to deal with obfuscation but, at the same time, we increase the false positives rate, namely the number of programs erroneously classified as infected. On the other hand, a more concrete α makes the detector more sensitive to obfuscation, while decreasing the presence of programs miss-classified as infected. In the following we provide a semantic characterization of the notions of soundness and completeness with respect to a set $\mathbb{O} \subseteq \mathbf{O}$ of obfuscating transformations.

DEFINITION 5. A semantic malware detector on α is *complete for* $\mathbb{O}$ if $\forall \mathcal{O} \in \mathbb{O}$:

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

A semantic malware detector on α is *sound for* $\mathbb{O}$ if:

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

In particular, completeness for a class $\mathbb{O}$ of obfuscating transformations means that, for every obfuscation $\mathcal{O} \in \mathbb{O}$, when program P is infected by a variant $\mathcal{O}\llbracket M \rrbracket$ of a malware, then the semantic malware detector is able to detect it (i.e., no false negatives). On the other side, soundness with respect to the class $\mathbb{O}$ of obfuscating transformations means that when the semantic malware detector classifies a program P as infected by a malware M, then there exists an obfuscation $\mathcal{O} \in \mathbb{O}$, such that program P is infected by the variant $\mathcal{O}\llbracket M \rrbracket$ of the malware (i.e., no false positives). In the following, when considering a class $\mathbb{O}$ of obfuscating transformations, we will also assume that the identity function belongs to $\mathbb{O}$, in this way we include in the set of variants identified by $\mathbb{O}$ the malware itself. It is interesting to observe that, considering an obfuscating transformation $\mathcal{O}$, completeness is guaranteed when abstraction α is preserved by obfuscation $\mathcal{O}$, namely when $\forall P \in \mathbf{P}$: $\alpha(\alpha_e(\mathfrak{S}\llbracket P \rrbracket)) = \alpha(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}(P) \rrbracket))$.

THEOREM 1. If abstraction $\alpha : \wp(\mathcal{X}^*) \to A$ is preserved by the transformation $\mathcal{O}$, namely if $\forall P \in \mathbf{P} : \alpha(\alpha_e(\mathfrak{S}\llbracket P \rrbracket)) = \alpha(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}(P) \rrbracket))$, then the semantic malware detector on α is complete for $\mathcal{O}$.

PROOF. In order to show that the semantic malware detector on α is complete for $\mathcal{O}$, we have to show that if $\mathcal{O}\llbracket M \rrbracket \hookrightarrow P$ then there exists $lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket)$ such that $\alpha(\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \subseteq \alpha(\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)))$. If $\mathcal{O}\llbracket M \rrbracket \hookrightarrow P$, it means that there exists $lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket)$ such that $P_r = \mathcal{O}\llbracket M \rrbracket$. By definition $\mathcal{O}\llbracket M \rrbracket$ is a program, thus $\mathfrak{S}\llbracket \mathcal{O}\llbracket M \rrbracket \rrbracket = \mathfrak{S}\llbracket P_r \rrbracket = \alpha_r(\mathfrak{S}\llbracket P \rrbracket)$. Moreover, we have that $\alpha(\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))) = \alpha(\alpha_e(\mathfrak{S}\llbracket P_r \rrbracket)) = \alpha(\alpha_e(\mathfrak{S}\llbracket \mathcal{O}\llbracket M \rrbracket \rrbracket)) = \alpha(\alpha_e(\mathfrak{S}\llbracket M \rrbracket))$, where the last equality follows from the hypothesis that α is preserved by $\mathcal{O}$. Thus, $\alpha(\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) = \alpha(\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)))$ which concludes the proof. $\square$

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 α that characterizes soundness of the semantic malware detector. However, given an abstraction α, we can characterize the set of transformations for which α is sound.

THEOREM 2. Given an abstraction α, consider the set $\mathbb{O} \subseteq \mathbf{O}$ such that: $\forall P, T \in \mathbf{P}$:

$$(\alpha(\alpha_e(\mathfrak{S}\llbracket T \rrbracket)) \subseteq \alpha(\alpha_e(\mathfrak{S}\llbracket P \rrbracket))) \Rightarrow (\exists \mathcal{O} \in \mathbb{O} : \alpha_e(\mathfrak{S}\llbracket \mathcal{O}\llbracket T \rrbracket \rrbracket) \subseteq \alpha_e(\mathfrak{S}\llbracket P \rrbracket)).$$

Then, a semantic malware detector on α is sound for $\mathbb{O}$.

PROOF. Suppose that these exists $lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket)$ such that $\alpha(\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \subseteq \alpha(\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)))$, since $M, P, P_r \in \mathbf{P}$ and $\alpha_r(\mathfrak{S}\llbracket P \rrbracket) = \mathfrak{S}\llbracket P_r \rrbracket$, then by definition of set $\mathbb{O}$ we have that: $\exists \mathcal{O} \in \mathbb{O} : \alpha_e(\mathfrak{S}\llbracket \mathcal{O}\llbracket M \rrbracket \rrbracket) \subseteq \alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))$, and therefore $\mathcal{O}\llbracket M \rrbracket \hookrightarrow P$. $\square$

5.2 A semantic classification of obfuscations

Let us introduce a classification of obfuscations based on the effects that these transformations have on program trace semantics. In particular, we distinguish between transformations that add new instructions while maintaining the structure of the original program traces (called *conservative*), and transformations that insert new instructions causing major changes to the original semantic structure (called *non-conservative*). Given two sequences $s, t \in A^*$ for some set A, let $s \preceq t$ denote that s is a *subsequence* of t, i.e., if $s = s_1 s_2 \dots s_n$ then t is of the form $\dots s_1 \dots s_2 \dots s_n \dots$. The idea is that an obfuscating transformation is a *conservative* obfuscation if every trace σ of the semantics of the original program is a subsequence of some trace δ of the semantics of the obfuscated program.

DEFINITION 6. An obfuscating transformation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ is *conservative* if:

$$\forall \sigma \in \mathfrak{S}\llbracket P \rrbracket, \exists \delta \in \mathfrak{S}\llbracket \mathcal{O}(P) \rrbracket : \alpha_e(\sigma) \preceq \alpha_e(\delta)$$

An obfuscation that does not satisfy the conservativeness property defined above is said to be *non-conservative*.

6. CONSERVATIVE OBFUSCATIONS

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 behavior M, if there is a malware trace $\sigma \in \mathfrak{S}\llbracket M \rrbracket$ whose environment-memory evolution is "contained" in 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 an environment-memory sequence $s \in \mathcal{X}^*$ and a set $S \in \wp(\mathcal{X}^*)$, returns the elements $t \in S$ that are sub-traces 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 $\alpha_c[S]$ 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 defined above turns out to be a suitable approximation when dealing with conservative obfuscations. In fact the semantic malware detector on $\alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)]$ is complete and sound for the class of conservative obfuscations $\mathbb{O}_c$.

THEOREM 3. Considering a vanilla malware M we have that a semantic malware detector on $\alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)]$ is complete and sound for $\mathbb{O}_c$, namely:
Completeness:

$$\begin{aligned}&\forall \mathcal{O}_c \in \mathbb{O}_c : \mathcal{O}_c\llbracket M \rrbracket \hookrightarrow P \Rightarrow \exists lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket) : \\ &\qquad\qquad \alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \subseteq \alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)))\end{aligned}$$

Soundness:

$$\begin{aligned}&\exists lab_r\llbracket P \rrbracket \in \wp(lab\llbracket P \rrbracket) : \\ &\qquad \alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) \subseteq \alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))) \Rightarrow \\ &\qquad\qquad\qquad\qquad \exists \mathcal{O}_c \in \mathbb{O}_c : \mathcal{O}_c\llbracket M \rrbracket \hookrightarrow P\end{aligned}$$

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

P

L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

$\mathcal{O}_J [\![P]\!]$

L_1	$: F := 1 \rightarrow L_2$
L_2	$: \texttt{skip} \rightarrow L_3$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: \texttt{skip} \rightarrow L_4$
L_3	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_T	$: ....$
L_4	$: F := F \times 2 - F \rightarrow L_3$

Fig. 4. Code reordering

that: $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Considering such σ and δ we show that $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\delta))$, in fact by definition of α_c we have that $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\delta)) = \alpha_e(\mathfrak{S}[\![M]\!]) \cap \mathit{SubSeq}(\alpha_e(\delta))$ and $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\sigma)) = \alpha_e(\mathfrak{S}[\![M]\!]) \cap \mathit{SubSeq}(\alpha_e(\sigma))$. Since $\alpha_e(\sigma) \preceq \alpha_e(\delta)$, it follows that $\mathit{SubSeq}(\alpha_e(\sigma)) \subseteq \mathit{SubSeq}(\alpha_e(\delta))$. Therefore, $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\delta))$, which concludes the proof.
Soundness: By hypothesis there exists $\mathit{lab}_r[\![P]\!] \in \wp(\mathit{lab}[\![P]\!])$ for which it holds that $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\mathfrak{S}[\![M]\!])) \subseteq \alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])))$. This means that $\forall \sigma \in \mathfrak{S}[\![M]\!]$ we have that: $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])))$, which means that $\alpha_e(\sigma) \in \{\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\delta)) \mid \delta \in \alpha_r(\mathfrak{S}[\![P]\!])\}$. Thus, $\forall \sigma \in \mathfrak{S}[\![M]\!]$, there exists $\delta \in \alpha_r(\mathfrak{S}[\![P]\!])$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and this means that P_r is a conservative obfuscation of malware M, namely $\exists \mathcal{O}_c \in \mathbb{O}_c$ such that $\mathcal{O}_c[\![M]\!] \hookrightarrow P$. $\square$

Thus, in order to deal with conservative obfuscations of a malware M, the semantic malware detector has to compute $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])))$, namely the intersection $\alpha_e(\mathfrak{S}[\![M]\!]) \cap \{\mathit{SubSeq}(\delta) \mid \delta \in \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))\}$. We expect the number of subsequences of the traces in $\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))$ to be quite large. Hence, a practical way for computing $\alpha_c[\alpha_e(\mathfrak{S}[\![M]\!])](\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])))$ is to check for each $\sigma \in \alpha_e(\mathfrak{S}[\![M]\!])$ if there exists a $\delta \in \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))$ such that $\sigma \preceq \delta$. This can be done without generating the set $\{\mathit{SubSeq}(\delta) \mid \delta \in \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))\}$, by incrementally searching in δ for successive environment-memory elements of σ. For example, let us consider a malicious behaviour $\sigma = \sigma_0 ... \sigma_n \in \alpha_e(\mathfrak{S}[\![M]\!])$, and let us denote a trace $\delta = \delta_0 ... \delta_m$ whose j-th element is δ_j as $\delta = \eta \delta_j \mu$ where $\eta = \delta_0 ... \delta_{j-1}$ and $\mu = \delta_{j+1} ... \delta_m$. Let $X_0 = \{\delta_j \mu \mid \exists \delta = \eta \delta_j \mu \in \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!])) : \delta_j = \sigma_0\}$, and let us define $X_i = \{\sigma_0 ... \sigma_{i-1} \delta_j \mu \mid \exists \sigma_0 ... \sigma_{i-1} \eta \delta_j \mu \in X_{i-1} : \delta_j = \sigma_i\}$. It is clear that if $X_i \neq \emptyset$ for all $i \in [0, n]$ then there exists $\delta \in \alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))$ such that $\sigma \preceq \delta$, otherwise the malicious behaviour σ is not present in $\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))$ and the program is not classified as infected.

It turns out that many obfuscating transformations commonly used by malware writers are conservative; a partial list of such conservative obfuscations is given below, together with a proof sketch of their conservativeness. It follows that Theorem 3 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 (see Fig. 4 for an example). Observe that, in the programming language introduced in Section 3.2, an unconditional jump is expressed as a command $L : \texttt{skip} \rightarrow L'$ that directs the flow of control of the program to a command labeled by L'. Let P be a program, $P = \{C_i : 1 \leq i \leq N\}$. The code reordering obfuscating transformation $\mathcal{O}_J : \mathbf{P} \to \mathbf{P}$ inserts $L : \texttt{skip} \to L'$ commands after selected commands from the program P. Let $R \subseteq P$ be a set of $m \leq N$ commands selected by the obfuscating transformation $\mathcal{O}_J$, i.e., $|R| = m$. The `skip` commands are then inserted after each one of the m selected commands in R. Let us define the subset S of commands of P that contains the successors of the commands in R:

$$S = \left\{ C' \in P \,\middle|\, \exists C \in R : lab\,[\![C']\!] \in suc\,[\![C]\!] \right\}$$

Effectively, the code reordering obfuscating transformation adds a `skip` command between a command $C \in R$ and its successor $C' \in S$. Define $\eta : \mathbb{C} \to \mathbb{C}$, a command-relabeling function, as follows:

$$\eta\,(L_1 : A \to L_2) = \mathit{NewLabel}(\mathbb{L} \setminus \{L_1\}) : A \to L_2$$

where $\mathit{NewLabel}(H)$ returns a label from the set $H \subseteq \mathbb{L}$. We extend η to a set of commands $T = \{\dots, L_i : A \to L_j, \dots\}$:

$$\eta(T) = \left\{ \dots, \mathit{NewLabel}(\mathbb{L}') : A \to L_j, \dots \right\}$$

where $\mathbb{L}' = \mathbb{L} \setminus \{\dots, L_i, \dots\}$. We can define the set of `skip` commands inserted by this obfuscating transformation:

$$\mathit{Skip}(S) = \left\{ L : \texttt{skip} \to L' \,\middle|\, \exists C \in S : L = lab\,[\![C]\!]\,, L' = lab\,[\![\eta(C)]\!] \right\}$$

Then, $\mathcal{O}_J\,[\![P]\!] = (P \setminus S) \cup \eta(S) \cup \mathit{Skip}(S)$. Considering the effects that code reordering has on program trace semantics, we have that for each trace in the original program $\sigma = \langle C_1, (\rho_1, m_1)\rangle ... \langle C_n, (\rho_n, m_n)\rangle \in \mathfrak{S}\,[\![P]\!]$, there exists a trace in the obfuscated program $\delta \in \mathfrak{S}\,[\![\mathcal{O}_J\,[\![P]\!]]\!]$ of the form

$$\delta = \langle SK, (\rho_1, m_1)\rangle^* \langle C'_1, (\rho_1, m_1)\rangle \dots \langle SK, (\rho_n, m_n)\rangle^* \langle C'_n, (\rho_n, m_n)\rangle$$

where the original commands ($act\,[\![C_i]\!] = act\,[\![C'_i]\!]$) are interleaved with any number of `skip` commands $SK \in \mathit{Skip}(S)$. Thus, $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and $\mathcal{O}_J \in \mathbb{O}_c$.

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 [Collberg et al. 1998]. In the following, we give an idea of why opaque predicate insertion is a conservative transformation by considering the three major types of opaque predicates: true, false and unknown (see Fig. 5 for an example of true opaque predicate insertion). In the considered programming language a *true opaque predicate* is expressed by a command $L : P^T \to \{L_T, L_F\}$. Since P^T always evaluates to $true$ the next command label is always L_T. When a true opaque predicate is inserted after command C the sequence of commands starting at label L_T is the sequence starting at $suc\,[\![C]\!]$ in the original program, while some buggy code is inserted starting form label L_F. Let $\mathcal{O}_T : \mathbf{P} \to \mathbf{P}$ be the obfuscating transformation that inserts true opaque predicates, and let P, R, S and η be defined as in the code reordering case. In fact, transformation $\mathcal{O}_T$ inserts opaque predicates between a command C in R and its successor C' in S. Let us define the

P	
L_1	$: F := 1 \to L_2$
L_2	$: (X = 1) \to \{L_T, L_F\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: F := F \times X \to L_2$
L_T	$: ...$

$\mathcal{O}_T [\![P]\!]$	
L_1	$: F := 1 \to L_2$
L_2	$: (X = 1) \to \{L_T, L_O\}$
L_O	$: P^T \to \{L_F, L_B\}$
L_F	$: X := X - 1 \to L_{F+1}$
L_{F+1}	$: F := F \times X \to L_2$
L_B	: buggy code
L_T	$: ...$

Fig. 5. True opaque predicate insertion at label L_O

set of commands encoding opaque predicate P^T inserted by $\mathcal{O}_T$ as:

$$\mathit{TrueOp}(S) \;=\; \left\{ L : P^T \to \{L_T, L_F\} \,\middle|\, \begin{array}{l} \exists C \in S : \\ L = \mathit{lab}\,[\![C]\!]\,, L_T = \mathit{lab}\,[\![\eta(C)]\!] \end{array} \right\}$$

$$\mathit{Bug}(\mathit{TrueOp}(S)) \;=\; \left\{ B_1...B_k \,\middle|\, \begin{array}{l} B_1...B_k \in \wp(\mathbb{C}) \\ \exists L : P^T \to \{L_T, L_F\} \in \mathit{TrueOp}(S) : \\ \mathit{lab}\,[\![B_1]\!] = L_F \end{array} \right\}$$

where $B_1...B_k$ is a sequence of commands expressing some buggy code. Then:

$$\mathcal{O}_T [\![P]\!] = (P \setminus S) \cup \eta(S) \cup \mathit{TrueOp}(S) \cup \mathit{Bug}(\mathit{TrueOp}(S))$$

Observing the effects on program semantics we have that for each trace $\sigma \in \mathfrak{S}[\![P]\!]$, such that $\sigma = \langle C_1, (\rho_1, m_1)\rangle ... \langle C_n, (\rho_n, m_n)\rangle$ there exists $\delta \in \mathfrak{S}[\![\mathcal{O}_T[\![P]\!]]\!]$ such that:

$$\delta = \langle OP, (\rho_1, m_1)\rangle^* \langle C'_1, (\rho_1, m_1)\rangle \langle OP, (\rho_2, m_2)\rangle^* ... \langle OP, (\rho_n, m_n)\rangle^* \langle C'_n, (\rho_n, m_n)\rangle$$

where $OP \in \mathit{TrueOp}(S)$, $\mathit{act}\,[\![C_i]\!] = \mathit{act}\,[\![C'_i]\!]$. Thus $\alpha_e(\sigma) \preceq \alpha_e(\delta)$ and $\mathcal{O}_T \in \mathbb{O}_c$. The same holds for the insertion of *false opaque predicates*.

An *unknown opaque predicate* $P^?$ sometimes evaluates to true and sometimes evaluates to false, thus the true and false branches have to exhibit equivalent behaviors. Usually, in order to avoid detection, the two branches present different obfuscated versions of the original command sequence. This can be seen as the composition of two or more distinct obfuscations: the first one $\mathcal{O}_U$ that inserts the unknown opaque predicates and duplicates the commands in such a way that the two branches present the same code sequence, and subsequent ones that obfuscate the code in order to make the two branches look different. Let $\mathcal{O}_U : \mathbf{P} \to \mathbf{P}$ be the program transformation that inserts unknown opaque predicates, and let P, R, S and η be defined as in the code reordering case. In the considered programming language an unknown opaque predicate is expressed as $L : P^? \to \{L_T, L_F\}$. Let us define the set of commands encoding an unknown opaque predicate $P^?$ inserted by the transformation $\mathcal{O}_U$:

$$\mathit{UnOp}(S) \;=\; \left\{ L : P^? \to \{L_T, L_F\} \,\middle|\, \begin{array}{l} \exists C \in S : \\ \mathit{lab}\,[\![C]\!] = L, \mathit{lab}\,[\![\eta(C)]\!] = L_T \end{array} \right\}$$

$$\mathit{Rep}(\mathit{UnOp}(S)) \;=\; \left\{ R_1...R_k \,\middle|\, \begin{array}{l} R_1...R_k \in \wp(\mathbb{C}) \\ \mathit{lab}\,[\![R_1]\!] = L_F \end{array} \right\}$$

where $R_1...R_k$ present the same sequence of actions of the commands starting at label L_T. Then, $\mathcal{O}_U [\![P]\!] = (P \setminus S) \cup \mathit{UnOp}(S) \cup \eta(S) \cup \mathit{Rep}(\mathit{UnOp}(S))$. Observ-

P

L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

$\mathcal{O}_N [\![P]\!]$

L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: X := X \times 2 - X$
L_{F+2}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

Fig. 6. Semantic NOP insertion at label L_{F+1}

P

L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - 1 \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \rightarrow L_2$
L_T	$: ...$

$\mathcal{O}_I [\![P]\!]$

L_1	$: F := 1 \rightarrow L_2$
L_2	$: (X = 1) \rightarrow \{L_T, L_F\}$
L_F	$: X := X - X/X \rightarrow L_{F+1}$
L_{F+1}	$: F := F \times X \times 2 - F \times X \rightarrow L_2$
L_T	$: ...$

Fig. 7. Substitution of equivalent commands at label L_F and L_{F+1}

ing the effects on program semantics we have that, for every trace $\sigma \in \mathfrak{S}[\![P]\!]$, where $\sigma = \langle C_1, (\rho_1, m_1)\rangle ... \langle C_n, (\rho_n, m_n)\rangle$, there exists $\delta \in \mathfrak{S}[\![\mathcal{O}_U[\![P]\!]]\!]$ such that:

$$\delta = \langle U, (\rho_1, m_1)\rangle^* \langle C_1', (\rho_1, m_1)\rangle \langle U, (\rho_2, m_2)\rangle^* ... \langle U, (\rho_n, m_n)\rangle^* \langle C_n', (\rho_n, m_n)\rangle$$

where $U \in UnOp(S)$ and $act[\![C_i]\!] = act[\![C_i']\!]$. Thus $\alpha_e(\sigma) = \alpha_e(\delta)$, and $\mathcal{O}_U \in \mathbb{O}_c$.

Semantic NOP *insertion.* This transformation inserts commands that are irrelevant with respect to program trace semantics (see Fig. 6 for an example). Let us consider commands $SN, C_1, C_2 \in \wp(\mathbb{C})$. We say SN is a semantic NOP with respect to $C_1 \cup C_2$ if for every $\sigma \in \mathfrak{S}[\![C_1 \cup C_2]\!]$, there exists $\delta \in \mathfrak{S}[\![C_1 \cup SN \cup C_2]\!]$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. Let $\mathcal{O}_N : \mathbf{P} \rightarrow \mathbf{P}$ be the program transformation that inserts irrelevant instructions, therefore $\mathcal{O}_N[\![P]\!] = P \cup SN$ where SN represents the set of irrelevant instructions inserted in P. Following the definition of semantic NOP we have that for every $\sigma \in \mathfrak{S}[\![P]\!]$ there exists $\delta \in \mathfrak{S}[\![\mathcal{O}_N[\![P]\!]]\!]$ such that $\alpha_e(\sigma) \preceq \alpha_e(\delta)$, thus $\mathcal{O}_N \in \mathbb{O}_c$.

Substitution of Equivalent Commands. This program transformation replaces a single command with an equivalent one, with the goal of thwarting signature matching (see Fig. 7 for an example). Let $\mathcal{O}_I : \mathbf{P} \rightarrow \mathbf{P}$ be the program transformation that substitutes commands with equivalent ones. Two commands C and C' are equivalent if they always cause the same effects, namely if $\forall \xi \in \mathcal{X} : \mathscr{C}[\![C]\!]\xi = \mathscr{C}[\![C']\!]\xi$. Thus, $\mathcal{O}_I[\![P]\!] = P'$ where $\forall C' \in P', \exists C \in P$ such that C and C' are equivalent. Observing the effects on program semantics we have that: for every $\sigma \in \mathfrak{S}[\![P]\!]$ such that $\sigma = \langle C_1, (\rho_1, m_1)\rangle ... \langle C_n, (\rho_n, m_n)\rangle$, there exists a trace $\delta \in \mathfrak{S}[\![\mathcal{O}_J[\![P]\!]]\!]$ such that $\delta = \langle C_1', (\rho_1, m_1)\rangle ... \langle C_n', (\rho_n, m_n)\rangle$ where $\mathscr{C}\langle C_i, (\rho_i, m_i)\rangle = \mathscr{C}\langle C_i', (\rho_i, m_i)\rangle$. Thus, $\alpha_e(\sigma) = \alpha_e(\delta)$, and $\mathcal{O}_I \in \mathbb{O}_c$.

Of course, malware writers usually combine different obfuscating transformations in order to prevent detection. Thus, it is crucial to understand whether (and how) the ability of a semantic malware detector to deal with single obfuscations can be "extended" in order to handle also their composition. First of all we observe that the composition of conservative obfuscations is a conservative obfuscation, namely that the property of being conservative is preserved by composition.

This means that when more than one conservative obfuscations are applied, they can be handled as a single conservative obfuscation, namely that abstraction α_c is able to deal with any composition of conservative obfuscations.

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

PROOF. By definition of conservative transformations we have that:

$$\begin{aligned} \forall \sigma \in \mathfrak{S}\llbracket P \rrbracket, \exists \delta \in \mathfrak{S}\llbracket \mathcal{O}_1\llbracket P \rrbracket \rrbracket : &\quad \alpha_e(\sigma) \preceq \alpha_e(\delta) \\ \forall \delta \in \mathfrak{S}\llbracket \mathcal{O}_1\llbracket P \rrbracket \rrbracket, \exists \eta \in \mathfrak{S}\llbracket \mathcal{O}_2\llbracket \mathcal{O}_1\llbracket P \rrbracket \rrbracket \rrbracket : &\quad \alpha_e(\delta) \preceq \alpha_e(\eta) \end{aligned}$$

Thus, for transitivity of $\preceq$: $\forall \sigma \in \mathfrak{S}\llbracket P \rrbracket, \exists \eta \in \mathfrak{S}\llbracket \mathcal{O}_2\llbracket \mathcal{O}_1\llbracket P \rrbracket \rrbracket \rrbracket$ such that $\alpha_e(\sigma) \preceq \alpha_e(\eta)$, which proves that $\mathcal{O}_2 \circ \mathcal{O}_1$ is a conservative transformation. $\square$

EXAMPLE 2. 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 and then executes the decrypted instructions. Observe that 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 $\pi_2(A, S) = S$. Moreover, given a variable X, let $Dec(X)$ denote the execution of a set of commands that decrypts the value stored in the memory location $\rho(X)$. Let $\mathcal{O}\llbracket M \rrbracket$ be a conservative obfuscation of M obtained through code reordering, opaque predicate insertion and semantic nop insertion.

M

L_1	$:\texttt{assign}(L_B, B) \to L_2$
L_2	$:\texttt{assign}(L_A, A) \to L_c$
L_c	$:\mathit{cond}(A) \to \{L_T, L_F\}$
L_T	$:B := \mathit{Dec}(A) \to L_{T_1}$
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	$:\texttt{skip} \to L_B$

$\mathcal{O}_c(M)$

L_1	$:\texttt{assign}(L_B, B) \to L_2$
L_2	$:\texttt{skip} \to L_4$
L_c	$:\mathit{cond}(A) \to \{L_O, L_F\}$
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	$:B := \mathit{Dec}(A) \to L_{T_1}$
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	$:\texttt{skip} \to L_B$

It can be shown that $\alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_c\llbracket M \rrbracket \rrbracket)) = \alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\mathfrak{S}\llbracket M \rrbracket))$, i.e., our semantics-based approach is able to see through the obfuscations and identify $\mathcal{O}\llbracket M \rrbracket$ as matching the malware M. In particular, let $\bot$ denote the undefined function.

$$\begin{aligned} \alpha_c[\alpha_e(\mathfrak{S}\llbracket M \rrbracket)](\alpha_e(\mathfrak{S}\llbracket M \rrbracket)) &= \alpha_e(\mathfrak{S}\llbracket M \rrbracket) \\ &= (\bot, \bot), ((B \rightsquigarrow L_B), \bot), ((B \rightsquigarrow L_B, A \rightsquigarrow L_A), \bot)^2, \\ &\quad ((B \rightsquigarrow L_B, A \rightsquigarrow L_A), (\rho(B) \leftarrow \mathbf{Dec}(A))), \\ &\quad ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow L_A), (\rho(B) \leftarrow \mathbf{Dec}(A))), \\ &\quad ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow \pi_2(m(\rho(A)))), \\ &\quad (\rho(B) \leftarrow \mathbf{Dec}(A)))... \end{aligned}$$

while

$$\begin{aligned}\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_c\llbracket M\rrbracket\rrbracket) \;=\;& (\bot,\bot),((B \rightsquigarrow L_B),\bot)^2,((B \rightsquigarrow L_B, A \rightsquigarrow L_A),\bot)^5,\\ & ((B \rightsquigarrow L_B, A \rightsquigarrow L_A),(\rho(X) \leftarrow X-3)),\\ & ((B \rightsquigarrow L_B, A \rightsquigarrow L_A),(\rho(X) \leftarrow X+3, \rho(X) \leftarrow X-3)),\\ & ((B \rightsquigarrow L_B, A \rightsquigarrow L_A),(\rho(B) \leftarrow \mathbf{Dec}(A))),\\ & ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow L_A),(\rho(B) \leftarrow \mathbf{Dec}(A))),\\ & ((B \rightsquigarrow \pi_2(m(\rho(B)), A \rightsquigarrow \pi_2(m(\rho(A)))),(\rho(B) \leftarrow \mathbf{Dec}(A)))\\ & \ldots\end{aligned}$$

Thus, $\alpha_c[\alpha_e(\mathfrak{S}\llbracket M\rrbracket)](\alpha_e(\mathfrak{S}\llbracket M\rrbracket)) \subseteq \alpha_c[\alpha_e(\mathfrak{S}\llbracket M\rrbracket)](\alpha_e(\mathfrak{S}\llbracket \mathcal{O}_c\llbracket M\rrbracket\rrbracket))$.

7. 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. This means that it is not possible to recognize that a trace σ is an obfuscated version of a trace δ by verifying if σ is a subsequence of δ under abstraction α_e, i.e., $\alpha_e(\sigma) \preceq \alpha_e(\delta)$. In order to tackle a non-conservative transformation $\mathcal{O}$ we can design:

—a transformation T such that $\forall \sigma \in \mathfrak{S}\llbracket P\rrbracket, \exists \delta \in \mathfrak{S}\llbracket \mathcal{O}\llbracket P\rrbracket\rrbracket$: $T(\alpha_e(\sigma)) = T(\alpha_e(\delta))$, or
—an abstraction α such that $\forall \sigma \in \mathfrak{S}\llbracket P\rrbracket, \exists \delta \in \mathfrak{S}\llbracket \mathcal{O}\llbracket P\rrbracket\rrbracket$: $\alpha(\sigma) \preceq \alpha(\delta)$, with $\alpha \not\sqsubseteq \alpha_e$

The idea of the first strategy mentioned above 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 transformation T would reduce the restricted program semantics and the malware semantics to the canonical version before checking for infection (see Section 7.1 for a detailed example). This idea of program normalization has already been used to deal with some obfuscations commonly used by metamorphic malware (e.g., [Christodorescu et al. 2005; Lakhotia and Mohammed 2004; Walenstein et al. 2006]).

On the other hand, in Section 7.2 we show how it is possible to handle a class of non-conservative obfuscations through a further abstraction of the malware semantics, namely by designing abstraction α of the second strategy listed above. In this case the idea is to weaken the notion of program infection in order to weaken the conservative condition. In this way a wider class of obfuscating transformations can be classified as conservative and we prove that results analogous to Theorem 3 still hold for this relaxed definition of infection and conservativeness.

Another possible approach comes from Theorem 1 that states that if α is preserved by $\mathcal{O}$ then the semantic malware detector on α is complete with respect to $\mathcal{O}$. Recall that, given a program transformation $\mathcal{O}$: **P** $\rightarrow$ **P**, it is possible to systematically derive the most concrete abstraction preserved by $\mathcal{O}$ [Dalla Preda and Giacobazzi 2005]. 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.

7.1 An Example: Canonical Variable Renaming

Let us 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 is a simple obfuscating transformation, often used to prevent signature matching, that replaces the names of variables with some different new names. As most code of interest in malware detection is already compiled and does not contain variable names, we cannot directly consider the effect of variable renaming has on compiled code. Instead, we assume that every environment function associates a variable V_L to a memory location L. Let $\mathcal{O}_v : \mathbf{P} \times \Pi \rightarrow \mathbf{P}$ denote the obfuscating transformation that, given a program P, renames its variables according to a mapping $\pi \in \Pi$, where $\pi : var\,[\![P]\!] \rightarrow Names$ is a bijective function that relates the name of each program variable to its new name.

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

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:

$$\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v(P,\pi)]\!]) = \alpha_v[\pi](\alpha_e(\mathfrak{S}\,[\![P]\!]))$$

where $\alpha_v : \Pi \rightarrow (\mathcal{X}^* \rightarrow \mathcal{X}^*)$ 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 introduce the notion of *canonical variable renaming* $\widehat{\pi}$. The idea of canonical mappings is that there exists a renaming $\pi : var\,[\![P]\!] \rightarrow var\,[\![Q]\!]$ that transforms program P into program Q, namely such that $\mathcal{O}_v(P,\pi) = Q$, iff $\alpha_v[\widehat{\pi}](\alpha_e(\mathfrak{S}\,[\![Q]\!])) = \alpha_v[\widehat{\pi}](\alpha_e(\mathfrak{S}\,[\![P]\!]))$. This means that a program Q is a renamed version of program P iff Q and P are indistinguishable after canonical renaming. In the following we define a possible canonical renaming for the variables of a given a program.

Let $\{V_i\}_{i\in\mathbb{N}}$ be a set of canonical variable names. The set $\mathbf{L}$ of memory locations is an ordered set with ordering relation $\leq_L$. With a slight abuse of notation we denote with $\leq_L$ also the lexicographical order induced by $\leq_L$ on sequences of memory locations. Let us define the ordering $\leq_\Sigma$ over traces Σ^* where, given $\sigma, \delta \in \Sigma^*$:

$$\sigma \leq_\Sigma \delta \quad \text{if} \quad \begin{cases} |\sigma| \leq |\delta| \text{ or} \\ |\sigma| = |\delta| \text{ and } lab(\sigma_1)lab(\sigma_2)...lab(\sigma_n) \leq_L lab(\delta_1)lab(\delta_2)...lab(\delta_n) \end{cases}$$

where $lab(\langle C,(\rho,m)\rangle) = lab\,[\![C]\!]$. It is clear that, given a program P, the ordering $\leq_\Sigma$ on its traces induces an order on the set $\mathcal{Z} = \alpha_e(\mathfrak{S}\,[\![P]\!])$ of its environment-memory traces, i.e., given $\sigma, \delta \in \mathfrak{S}\,[\![P]\!]$:

$$\sigma \leq_\Sigma \delta \Rightarrow \alpha_e(\sigma) \leq_\mathcal{Z} \alpha_e(\delta)$$

By definition, the set of variables assigned in $\mathcal{Z}$ is exactly $var\,[\![P]\!]$, therefore a canonical renaming $\hat{\pi}_P : var\,[\![P]\!] \rightarrow \{V_i\}_{i\in\mathbb{N}}$, is such that $\alpha_e(S\,[\![\mathcal{O}_v\,[\![P,\hat{\pi}_P]\!]]\!]) = \alpha_v[\hat{\pi}_P](\mathcal{Z})$. Let $\bar{\mathcal{Z}}$ denote the list of environment-memory traces of $\mathcal{Z} = \alpha_e(\mathfrak{S}\,[\![P]\!])$ ordered following the order defined above. Let B be a list, then $hd(B)$ returns the first element of the list, $tl(B)$ returns list B without the first element, $B : e$ ($e : B$) is the list resulting by inserting element e at the end (beginning) of B, $B[i]$ returns the i-th element of the list, and $e \in B$ means that e is an element of B. Note that program execution starts from the

```
Input: A list of context sequences $\bar{\mathcal{Z}}$, with $\mathcal{Z} \in \alpha_e(\mathfrak{S}[\![P]\!])$.
Output: A list $Rename[\mathcal{Z}]$ that associates canonical variable $V_i$ to the variable in the
        list position $i$.

$Rename[\mathcal{Z}] = List(hd(\bar{\mathcal{Z}}))$
$\bar{\mathcal{Z}} = tl(\bar{\mathcal{Z}})$
while ($\bar{\mathcal{Z}} \neq \emptyset$) do
    $trace = List(hd(\bar{\mathcal{Z}}))$
    while ($trace \neq \emptyset$) do
        if ($hd(trace) \notin Rename[\mathcal{Z}]$) then
            $Rename[\mathcal{Z}] = Rename[\mathcal{Z}] : hd(trace)$
        end
        $trace = tl(trace)$
    end
    $\bar{\mathcal{Z}} = tl(\bar{\mathcal{Z}})$
end
```

Algorithm 1: Canonical renaming of variables.

uninitialized environment $\rho_{uninit} = \lambda X.\bot$, and that each command assigns at most one variable. Let $def(\rho)$ denote the set of variables that have defined (i.e., non-$\bot$) values in an environment ρ. This means that considering $s \in \mathcal{X}^*$ we have that $def(\rho_{i-1}) \subseteq def(\rho_i)$, and if $def(\rho_{i-1}) \subset def(\rho_i)$ then $def(\rho_i) = def(\rho_{i-1}) \cup \{X\}$ where $X \in \mathbf{X}$ is the new variable assigned to memory location $\rho_i(X)$. Given $s \in \mathcal{X}^*$, let us define $List(s)$ as the list of variables in s ordered according to their assignment time. Formally, let $s = (\rho_1, m_1)(\rho_2, m_2)...(\rho_n, m_n) = (\rho_1, m_1)s'$:

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

Algorithm 1, given a list $\bar{\mathcal{Z}}$ encoding the ordering $\leq_{\mathcal{Z}}$ on context traces in $\alpha_e(\mathfrak{S}[\![P]\!])$, and given $List(s)$ for every $s \in \alpha_e(\mathfrak{S}[\![P]\!])$ encoding the assignment ordering of variables in s, returns the list $Rename[\mathcal{Z}]$ encoding the ordering of variables in $\alpha_e(\mathfrak{S}[\![P]\!])$. Given $\mathcal{Z} = \alpha_e(\mathfrak{S}[\![P]\!])$ we rename its variables following the canonical renaming $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$ that associates the new canonical name V_i to the variable of P in the i-th position in the list $Rename[\mathcal{Z}]$. Thus, the canonical renaming $\widehat{\pi}_P : var[\![P]\!] \to \{V_i\}_{i\in\mathbb{N}}$ is defined as follows:

$$\widehat{\pi}_P(X) = V_i \Leftrightarrow Rename[\mathcal{Z}][i] = X \tag{2}$$

The following result is necessary to prove that the mapping $\widehat{\pi}_P$ defined in Equation (2) is a canonical renaming.

LEMMA 2. Given two programs $P, Q \in \mathbf{P}$ let $\mathcal{Z} = \alpha_e(\mathfrak{S}[\![P]\!])$ and $\mathcal{Y} = \alpha_e(\mathfrak{S}[\![Q]\!])$. The following hold:

(1) $\alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\mathcal{Y}) \Rightarrow \exists \pi : var[\![P]\!] \to var[\![Q]\!] : \alpha_v[\pi](\mathcal{Z}) = \mathcal{Y}$

(2) $(\exists \pi : var[\![P]\!] \to var[\![Q]\!] : \alpha_v[\pi](\mathcal{Z}) = \mathcal{Y})$ and $(\alpha_v[\pi](s) = t \Rightarrow (\bar{\mathcal{Z}}[i] = s$ and $\mathcal{Y}[i] = t)) \Rightarrow \alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\mathcal{Y})$

PROOF. (1) Assume $\alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\mathcal{Y})$, i.e., we have that $\{\alpha_v[\widehat{\pi}_P](s) \mid s \in \mathcal{Z}\} = \{\alpha_v[\widehat{\pi}_Q](t) \mid t \in \mathcal{Y}\}$. This means that $|var\,[\![\mathcal{Z}]\!]\,| = |var\,[\![\mathcal{Y}]\!]\,| = k$, and that $\widehat{\pi}_P : var\,[\![\mathcal{Z}]\!] \rightarrow \{V_1...V_k\}$ while $\widehat{\pi}_Q : var\,[\![\mathcal{Y}]\!] \rightarrow \{V_1...V_k\}$. Recall that $var\,[\![\mathcal{Z}]\!] = var\,[\![P]\!]$ and $var\,[\![\mathcal{Y}]\!] = var\,[\![Q]\!]$. Let us define $\pi : var\,[\![P]\!] \rightarrow var\,[\![Q]\!]$ as $\pi = \widehat{\pi}_Q^{-1} \circ \widehat{\pi}_P$. The mapping π is bijective since it is obtained as composition of bijective functions. Let us show that π satisfies the condition on the left, namely that $\mathcal{Y} = \alpha_v[\pi](\mathcal{Z})$. To prove this we show that given $s \in \mathcal{Z}$ and $t \in \mathcal{Y}$ such that $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](t)$ then $\alpha_v[\pi](s) = t$. Let $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](t) = (\widehat{\rho}_1, m_1)...(\widehat{\rho}_n, m_n)$, while $s = (\rho_1^s, m_1)...(\rho_n^s, m_n)$ and $t = (\rho_1^t, m_1)...(\rho_n^t, m_n)$. Then:

$$\begin{aligned}
\alpha_v[\pi](s) &= (\rho_1^s \circ \pi^{-1}, m_1)...(\rho_n^s \circ \pi^{-1}, m_n)\\
&= (\rho_1^s \circ \widehat{\pi}_P^{-1} \circ \widehat{\pi}_Q, m_1)...(\rho_n^s \circ \widehat{\pi}_P^{-1} \circ \widehat{\pi}_Q, m_n)\\
&= (\widehat{\rho}_1 \circ \widehat{\pi}_Q, m_1)...(\widehat{\rho}_n \circ \widehat{\pi}_Q, m_n)\\
&= (\rho_1^t, m_1)...(\rho_n^t, m_n) = t
\end{aligned}$$

(2) Assume $\exists \pi : var\,[\![P]\!] \rightarrow var\,[\![Q]\!]$ such that $\mathcal{Y} = \alpha_v[\pi](\mathcal{Z})$. By definition $\mathcal{Y} = \{\alpha_v[\pi](s) \mid s \in \mathcal{Z}\}$. Let us show that $\alpha_v[\widehat{\pi}_P](\mathcal{Z}) = \alpha_v[\widehat{\pi}_Q](\{\alpha_v[\pi](s) \mid s \in \mathcal{Z}\})$. We prove this by showing that $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](\alpha_v[\pi](s))$. By definition we have that $|\mathcal{Y}| = |\mathcal{Z}|$ and $|var\,[\![P]\!]\,| = |var\,[\![Q]\!]\,| = k$, moreover we have $\pi : var\,[\![P]\!] \rightarrow var\,[\![Q]\!]$. Given $s \in \mathcal{Z}$ and $t \in \mathcal{Y}$ such that $t = \alpha_v[\pi](s)$ then $|s| = |t|$ and $|var\,[\![s]\!]\,| = |var\,[\![t]\!]\,|$, thus $List(s)[i] = X$ and $List(t)[i] = \pi(X)$, moreover, by hypothesis, $\bar{\mathcal{Z}}[i] = s$ and $\bar{\mathcal{Y}}[i] = t$. This hold for every pair of traces obtained trough renaming. Therefore, considering the canonical rename for $\mathcal{Y}$ as given by $\widehat{\pi}_Q = \widehat{\pi}_P \circ \pi^{-1}$, we have that $\forall s \in \mathcal{Z}, t \in \mathcal{Y}$ such that $\alpha_v[\pi](s) = t$ then $\alpha_v[\widehat{\pi}_P](s) = \alpha_v[\widehat{\pi}_Q](t)$. In fact:

$$\begin{aligned}
\alpha_v[\widehat{\pi}_Q](t) &= \alpha_v[\widehat{\pi}_Q](\alpha_v[\pi](s))\\
&= \alpha_v[\widehat{\pi}_Q]((\rho_1^s \circ \pi^{-1}, m_1)...(\rho_n^s \circ \pi^{-1}, m_n))\\
&= (\rho_1^s \circ \pi^{-1} \circ \widehat{\pi}_Q^{-1}, m_1)...(\rho_n^s \circ \pi^{-1} \circ \widehat{\pi}_Q^{-1}, m_n)\\
&= (\rho_1^s \circ \pi^{-1} \circ \pi \circ \widehat{\pi}_P^{-1}, m_1)...(\rho_n^s \circ \pi^{-1} \circ \pi \circ \widehat{\pi}_P^{-1}, m_n)\\
&= (\rho_1^s \circ \widehat{\pi}_P^{-1}, m_1)...(\rho_n^s \circ \widehat{\pi}_P^{-1}, m_n)\\
&= (\widehat{\rho}_1, m_1)...(\widehat{\rho}_n, m_n) = \alpha_v[\widehat{\pi}_P](s).
\end{aligned}$$

which concludes the proof. □

Let $\widehat{\Pi}$ denote a set of canonical variable renaming, the additive function $\alpha_v : \widehat{\Pi} \rightarrow (\wp(\mathcal{X}^*) \rightarrow \wp(\mathcal{X}_c^*))$, where $\mathcal{X}_c$ denotes execution contexts where environments are defined on canonical variables, is an approximation that abstracts from the names of variables. Thus, we have the following Galois connection:

$$\langle \wp(\mathcal{X}^*), \subseteq \rangle \underset{\alpha_v[\widehat{\Pi}]}{\overset{\gamma_v[\widehat{\Pi}]}{\leftrightarrows}} \langle \wp(\mathcal{X}_c^*), \subseteq \rangle$$

The following result, where $\widehat{\pi}_M$ and $\widehat{\pi}_{P_r}$ denote respectively the canonical rename of the malware variables and of restricted program variables, shows that the semantic malware detector on $\alpha_v[\widehat{\Pi}]$ is complete and sound for variable renaming.

THEOREM 4. $\exists \pi : \mathcal{O}_v(M, \pi) \hookrightarrow P$ iff

$$\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \alpha_v[\widehat{\pi}_M](\alpha_e(\mathfrak{S}\,[\![M]\!])) \subseteq \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])))$$

PROOF. ($\Rightarrow$) Completeness: Assume that $\mathcal{O}_v\,[\![M, \pi]\!] \hookrightarrow P$, this means that $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $P_r = \mathcal{O}_v\,[\![M, \pi]\!]$. Thus $\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])) = \alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v\,[\![M, \pi]\!]]\!])$. In order to conclude the proof we have to prove the following $\alpha_v[\widehat{\pi}_M](\alpha_e(\mathfrak{S}\,[\![M]\!])) \subseteq \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v\,[\![M, \pi]\!]]\!]))$. Recall that $\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v\,[\![M, \pi]\!]]\!]) = \alpha_v[\pi](\alpha_e(\mathfrak{S}\,[\![M]\!]))$. Following Lemma 2 point 2 we have that:

$$\alpha_v[\widehat{\pi}_M](\alpha_e(\mathfrak{S}\,[\![M]\!])) = \alpha_v[\widehat{\pi}_{P_r}](\alpha_v[\pi](\alpha_e(\mathfrak{S}\,[\![M]\!]))) = \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v\,[\![M, \pi]\!]]\!]))$$

which concludes the proof.
($\Leftarrow$) Soundness: Assume that there exists $lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \alpha_v[\widehat{\pi}_M](\alpha_e(\mathfrak{S}\,[\![M]\!])) \subseteq \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])))$. Let α_R be the program restriction that satisfies the above equation with equality: $\alpha_v[\widehat{\pi}_M](\alpha_e(\mathfrak{S}\,[\![M]\!])) = \alpha_v[\widehat{\pi}_{P_r}](\alpha_e(\alpha_R(\mathfrak{S}\,[\![P]\!])))$. It is clear that $\alpha_R(\mathfrak{S}\,[\![P]\!]) \subseteq \alpha_r(\mathfrak{S}\,[\![P]\!])$. From Lemma 2 point 1 we have that $\exists \pi : var\,[\![M]\!] \rightarrow var\,[\![P_R]\!]$ such that $\alpha_e(\alpha_R(\mathfrak{S}\,[\![P]\!])) = \alpha_v[\pi](\alpha_e(\mathfrak{S}\,[\![M]\!])) = \alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v\,[\![M, \pi]\!]]\!])$, namely $\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_v\,[\![M, \pi]\!]]\!]) = \alpha_e(\alpha_R(\mathfrak{S}\,[\![P]\!])) \subseteq \alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))$, meaning that $\mathcal{O}_v\,[\![M, \pi]\!] \hookrightarrow P$. $\square$

7.2 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 for every malicious behavior there exists a program behavior that matches it. In the following we show how this notion of infection can be weakened in three 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 behavior (i.e., bad states). Second, we can require program P to match a proper subset of malicious behaviors (i.e., bad behaviors). Furthermore these two notions of malware infection can be combined by requiring program P to match some states on a subset of malicious behaviors. Finally, the infection condition can be expressed in terms of a sequence of actions rather than a sequence of execution contexts (i.e., bad actions). Once again, bad actions can be combined with either bad states, or bad behaviors, or both. It is clear that a deeper understanding of the malware behavior is necessary in order to specify each of the proposed simplifications. We will discuss how it is possible to expand the set of conservative obfuscations by weakening the notion of malware infection. The basic idea is that a further abstraction of the infection condition implies a weaker notion of conservativeness.

Bad States

The maliciousness of a malware behavior may be expressed by the fact that some (malware) states are reached in a certain order during program execution. Observe that this condition is clearly implied by, i.e., weaker than, the (standard) matching relation between all malware traces and the restricted program traces.

Let the *bad states* of a malware refer to those states that capture the malicious behavior. Assume that we have an oracle that, given a malware M, returns the set of its bad states $Bad(\Sigma_M) \subseteq \Sigma\,[\![M]\!]$. These states could be selected based on a security policy. For example, the states could represent the result of network operations. This means that in order to verify if program P is infected by malware M, we have to check whether the malicious sequences of bad states are present in P. Let us define the trace transformation $\alpha_{Bad(\Sigma_M)} : \mathcal{X}^* \to \mathcal{X}^*$ which considers only the bad contexts in a given trace $s = \xi_1 s'$:

$$\alpha_{Bad(\Sigma_M)}(s) = \begin{cases} \epsilon & \text{if } s = \epsilon \\ \xi_1 \alpha_{Bad(\Sigma_M)}(s') & \text{if } \xi_1 \in \alpha_e(Bad(\Sigma_M)) \\ \alpha_{Bad(\Sigma_M)}(s') & \text{otherwise} \end{cases}$$

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

DEFINITION 7. A program P is infected by a vanilla malware M with bad states $Bad(\Sigma_M)$, i.e., $M \hookrightarrow_{Bad(\Sigma_M)} P$, if $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ such that:

$$\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket)) \subseteq \alpha_{Bad(\Sigma_M)}(\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))$$

This means that the standard notion of conservative transformations can be weakened according to the following.

DEFINITION 8. An obfuscation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ is *conservative with respect to* $Bad(\Sigma_M)$ if:

$$\forall \sigma \in \mathfrak{S} \llbracket P \rrbracket, \exists \delta \in \mathfrak{S} \llbracket \mathcal{O} \llbracket P \rrbracket \rrbracket : \ \alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) \preceq \alpha_{Bad(\Sigma_M)}(\alpha_e(\delta))$$

When program infection is characterized by Definition 7, the semantic malware detector on $\alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha(S \llbracket M \rrbracket))] \circ \alpha_{Bad(\Sigma_M)}$ is complete and sound for the obfuscating transformations that are conservative with respect to $Bad(\Sigma_M)$.

THEOREM 5. Let $Bad(\Sigma_M)$ be the set of bad states of a vanilla malware M. Then: *Completeness*: For every obfuscation $\mathcal{O}$ which is conservative with respect to $Bad(\Sigma_M)$, if $\mathcal{O} \llbracket M \rrbracket \hookrightarrow_{Bad(\Sigma_M)} P$ there exists $lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ such that:

$$\begin{aligned} &\alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))) \subseteq \\ &\qquad\qquad \alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))) \end{aligned}$$

Soundness: If there exists $lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ such that:

$$\begin{aligned} &\alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))) \subseteq \\ &\qquad\qquad \alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))) \end{aligned}$$

then there exists an obfuscation $\mathcal{O}$ that is conservative with respect to $Bad(\Sigma_M)$ such that $\mathcal{O} \llbracket M \rrbracket \hookrightarrow P$.

PROOF. Completeness: Let $\mathcal{O}$ be a conservative obfuscation with respect to $Int(M)$ such that $\mathcal{O} \llbracket M \rrbracket \hookrightarrow_{Bad(\Sigma_M)} P$, then it means that $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ such that $P_r = \mathcal{O} \llbracket M \rrbracket$, namely $\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket)) = \alpha_{Bad(\Sigma_M)}(\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))$. Therefore, we have that:

$$\begin{aligned} &\alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket))) = \\ &\qquad\qquad \alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S} \llbracket M \rrbracket))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)))) \end{aligned}$$

Thus, we have to show that:

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

By hypothesis $\mathcal{O}$ is conservative with respect to $Bad(\Sigma_M)$, thus we have that for every $\sigma \in \mathfrak{S}[\![M]\!]$, there exists $\delta \in \mathfrak{S}[\![\mathcal{O}[\![M]\!]]\!] : \alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) \preceq \alpha_{Bad(\Sigma_M)}(\alpha_e(\delta))$. Moreover, for every $s \in \alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!])))$ there exists $\sigma \in \mathfrak{S}[\![M]\!] : s = \alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma))$, therefore $\forall \sigma \in \mathfrak{S}[\![M]\!]$, there exists $\delta \in \mathfrak{S}[\![\mathcal{O}[\![M]\!]]\!]$ such that $s = \alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) \preceq \alpha_{Bad(\Sigma_M)}(\alpha_e(\delta))$, and $\alpha_{Bad(\Sigma_M)}(\alpha_e(\delta)) = t \in \alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![\mathcal{O}[\![M]\!]]\!]))$. This means that for every $s \in \alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!]))$, there exists $t \in \alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![\mathcal{O}[\![M]\!]]\!]))$ such that $s \in SubSeq(t)$. Hence, $\forall s \in \alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!]))$ we have that

$$s \in \alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![\mathcal{O}[\![M]\!]]\!])))$$

which concludes the proof.
Soundness: Assume that $\exists lab_r[\![P]\!] \in \wp(lab[\![P]\!])$ such that:

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

This means that $\forall \sigma \in \mathfrak{S}[\![M]\!]$:

$$\alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) \subseteq \alpha_c[\alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!]))](\alpha_{Bad(\Sigma_M)}(\alpha_e(\alpha_r(\mathfrak{S}[\![P]\!]))))$$

and for every $\sigma \in \mathfrak{S}[\![M]\!]$ there exists $\delta \in \alpha_r(\mathfrak{S}[\![P]\!])$ such that $\alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) \in \alpha_{Bad(\Sigma_M)}(\alpha_e(\mathfrak{S}[\![M]\!])) \cap SubSeq(\alpha_{Bad(\Sigma_M)}(\alpha_e(\delta)))$. This means that $\forall \sigma \in \mathfrak{S}[\![M]\!]$ there exists $\delta \in \alpha_r(\mathfrak{S}[\![P]\!])$ such that $\alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) \preceq \alpha_{Bad(\Sigma_M)}(\alpha_e(\delta))$, which means that P_r is a conservative obfuscation of M with respect to $Bad(\Sigma_M)$. $\square$

It is clear that transformations that are non-conservative according to Definition 6 may be conservative with respect to $Bad(\Sigma_M)$, meaning that knowing the set of bad states of a malware allows us to handle also some non-conservative obfuscations. For example the abstraction $\alpha_{Bad(\Sigma_M)}$ may allow the semantic malware detector to deal with the 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
$L_1 : A_1 \to L_2$
$L_2 : A_2 \to L_3$
$L_3 : A_3 \to L_4$
$L_4 : A_4 \to L_5$
$L_5 : A_5 \to L_6$

$\mathcal{O}[\![M]\!]$
$L_1 : A_1 \to L_2$
$L_2 : A_3 \to L_3$
$L_3 : A_2 \to L_4$
$L_4 : A_4 \to L_5$
$L_5 : A_5 \to L_6$

In this case actions A_2 and A_3 are independent, meaning that $\mathscr{A}[\![A_2]\!](\mathscr{A}[\![A_3]\!](\rho, m)) = \mathscr{A}[\![A_3]\!](\mathscr{A}[\![A_2]\!](\rho, m))$ for every $(\rho, m) \in \mathcal{E} \times \mathcal{M}$. Considering malware M, we have the trace $\sigma = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$ where:

$$\begin{aligned}
\sigma_1 &= \langle L_1 : A_1 \to L_2, (\rho, m)\rangle = \langle L_1 : A_1 \to L_2, \xi_1^\sigma\rangle \\
\sigma_2 &= \langle L_2 : A_2 \to L_3, (\mathscr{A}\llbracket A_1\rrbracket(\rho, m))\rangle \\
\sigma_3 &= \langle L_3 : A_3 \to L_4, (\mathscr{A}\llbracket A_2\rrbracket(\mathscr{A}\llbracket A_1\rrbracket(\rho, m)))\rangle \\
\sigma_4 &= \langle L_4 : A_4 \to L_5, (\mathscr{A}\llbracket A_3\rrbracket(\mathscr{A}\llbracket A_2\rrbracket(\mathscr{A}\llbracket A_1\rrbracket(\rho, m))))\rangle \\
\sigma_5 &= \langle L_5 : A_5 \to L_6, (\mathscr{A}\llbracket A_4\rrbracket(\mathscr{A}\llbracket A_3\rrbracket(\mathscr{A}\llbracket A_2\rrbracket(\mathscr{A}\llbracket A_1\rrbracket(\rho, m)))))\rangle \\
&= \langle L_5 : A_5 \to L_6, \xi_5^\sigma\rangle
\end{aligned}$$

while considering the obfuscated version, we have the trace $\delta = \delta_1\delta_2\delta_3\delta_4\delta_5$, where:

$$\begin{aligned}
\delta_1 &= \langle L_1 : A_1 \to L_2, (\rho, m)\rangle = \left\langle L_1 : A_1 \to L_2, \xi_1^\delta\right\rangle \\
\delta_2 &= \langle L_2 : A_3 \to L_3, (\mathscr{A}\llbracket A_1\rrbracket(\rho, m))\rangle \\
\delta_3 &= \langle L_3 : A_2 \to L_4, (\mathscr{A}\llbracket A_3\rrbracket(\mathscr{A}\llbracket A_1\rrbracket(\rho, m)))\rangle \\
\delta_4 &= \langle L_4 : A_4 \to L_5, (\mathscr{A}\llbracket A_2\rrbracket(\mathscr{A}\llbracket A_3\rrbracket(\mathscr{A}\llbracket A_1\rrbracket(\rho, m))))\rangle \\
\delta_5 &= \langle L_5 : A_5 \to L_6, (\mathscr{A}\llbracket A_4\rrbracket(\mathscr{A}\llbracket A_2\rrbracket(\mathscr{A}\llbracket A_3\rrbracket(\mathscr{A}\llbracket A_1\rrbracket(\rho, m)))))\rangle \\
&= \left\langle L_5 : A_5 \to L_6, \xi_5^\delta\right\rangle
\end{aligned}$$

Let $Bad(\Sigma_M) = \{\sigma_1, \sigma_5\}$. Then $\alpha_{Bad(\Sigma_M)}(\alpha_e(\sigma)) = \xi_1^\sigma\xi_5^\sigma$ as well as $\alpha_{Bad(\Sigma_M)}(\alpha_e(\delta)) = \xi_1^\delta\xi_5^\delta$, which concludes the example. It is obvious that $\xi_1^\sigma = \xi_1^\delta$, moreover $\xi_5^\delta = \xi_5^\sigma$ follows from the independence of A_2 and A_3.

Bad Behaviors

Program trace semantics expresses malware behavior on every possible input. It is clear that it may happen that only some of the inputs cause the malware to have a malicious behavior (e.g., consider a virus that starts its payload only after a certain date). In this case, maliciousness is properly expressed by a subset of malware traces that identify the so called *bad behaviors* of the malware. Assume we have an oracle that given a malware M returns the set $T \subseteq \mathfrak{S}\llbracket M\rrbracket$ of its bad behaviors. Thus, in order to verify if P is infected by M, we check whether program P matches the malicious behaviors T. The following definition characterizes the presence of malware M in terms of its bad behaviors T.

DEFINITION 9. A program P is infected by a vanilla malware M with bad behaviors $T \subseteq \mathfrak{S}\llbracket M\rrbracket$, i.e., $M \hookrightarrow_T P$ if:

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

It is interesting to observe that, when program infection is characterized by Definition 9, all the results obtained in Section 5 still hold if we replace $\mathfrak{S}\llbracket M\rrbracket$ with T. In particular, we can weaken the original notion of conservative transformation by saying that a transformation is conservative with respect to T if every malware trace that belongs to T is a subsequence of some obfuscated malware trace.

DEFINITION 10. An obfuscation $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ is *conservative with respect to* T if:

$$\forall \sigma \in \alpha_e(T), \exists \delta \in \mathfrak{S}\llbracket \mathcal{O}\llbracket M\rrbracket\rrbracket : \ \alpha_e(\sigma) \preceq \alpha_e(\delta)$$

Also in this case we have that obfuscations that are non-conservative may be conservative with respect to T. Consider for example an obfuscating transformation that modifies in a different way the instructions belonging to the true and to the false path of each conditional branch. In particular, assume that the false path is modified in a conservative way while the true one in a non-conservative way. In this case the transformation is conservative with respect to the traces obtained following the false path for every conditional branch. When considering obfuscations that are conservative with respect to bad behaviors, we have that Theorem 3 still holds simply by replacing $\mathfrak{S}\llbracket M \rrbracket$ with T.

Clearly the two abstractions can be composed. In this case a program P is infected by a malware M if there exists a program restriction that matches the set of bad sequences of states obtained abstracting the bad behaviors of the malware, i.e., $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) :$ $\alpha_e(\alpha_{Bad(\Sigma_M)}(T)) \subseteq \alpha_e(\alpha_{Bad(\Sigma_M)}(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)))$.

Bad Actions

To conclude, we present a matching relation based on *bad program actions* rather than environment-memory evolutions. In fact, sometimes, a malicious behavior can be characterized as the execution of a sequence of bad actions. In this case we consider the syntactic information contained in program states. The main difference with purely syntactic approaches is the ability to observe actions in their execution order and not in the order in which they appear in the code. Assume we have an oracle that given a malware M returns the set $Bad(M) \subseteq act \llbracket M \rrbracket$ of actions capturing the essence of the malicious behavior. In this case, in order to verify if program P is infected by malware M, we check whether the execution sequences of bad actions of the malware match the ones of the program.

DEFINITION 11. A program P is infected by a vanilla malware M with bad actions $Bad(M)$, i.e., $M \hookrightarrow_{Bad(M)} P$ if:

$$\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \alpha_a(\mathfrak{S}\llbracket M \rrbracket) \subseteq \alpha_a(\alpha_r(\mathfrak{S}\llbracket P \rrbracket))$$

Where, given the set $Bad \subseteq act \llbracket M \rrbracket$ of bad actions, the abstraction α_a returns the sequence of malicious actions executed by each trace. Formally, given a trace $\sigma = \sigma_1\sigma'$:

$$\alpha_a(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ A_1\alpha_a(\sigma') & \text{if } A_1 \in Bad(M) \\ \alpha_a(\sigma') & \text{otherwise} \end{cases}$$

Even if this abstraction considers syntactic information (program actions), it is able to deal with certain kinds of obfuscations. In fact, considering the sequence of malicious actions in a trace, we observe actions in their execution order, and not in the order in which they are written in the code. This means that, for example, we are able to ignore unconditional jumps and therefore we can deal with code reordering. Once again, abstraction α_a can be combined with bad states and/or bad behaviors. For example, program infection can be characterized as the sequences of bad actions present in the bad behaviors of malware M, i.e., $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket)$ such that $\alpha_a(\alpha_e(T)) \subseteq \alpha_a(\alpha_e(\alpha_r(\mathfrak{S}\llbracket P \rrbracket)))$.

It is clear that the notion of infection given in Definition 4 can be weakened in many other ways, following the example given by the above simplifications. This possibility of adjusting malware infection with respect to the knowledge of the malicious behavior we are searching for proves the flexibility of the proposed semantic framework.

8. COMPOSITION

In general a malware uses multiple obfuscating transformations concurrently to prevent detection, therefore we have to consider the composition of non-conservative obfuscations (Lemma 1 regards composition of conservative obfuscations). Investigating the relation between abstractions α_1 and α_2, on which the semantic malware detector is complete (resp. sound) respectively for obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$, and the abstraction that is complete (resp. sound) for their compositions, i.e., for $\{\mathcal{O}_1 \circ \mathcal{O}_2, \mathcal{O}_2 \circ \mathcal{O}_1\}$, we have obtained the following result.

THEOREM 6. Given two abstractions α_1 and α_2 and two obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$ then:

(1) if the semantic malware detector on α_1 is complete for $\mathcal{O}_1$, the semantic malware detector on α_2 is complete for $\mathcal{O}_2$, and $\alpha_1 \circ \alpha_2 = \alpha_2 \circ \alpha_1$, then the semantic malware detector on $\alpha_1 \circ \alpha_2$ is complete for $\{\mathcal{O}_1 \circ \mathcal{O}_2, \mathcal{O}_2 \circ \mathcal{O}_1\}$;

(2) if the semantic malware detector on α_1 is sound for $\mathcal{O}_1$, the semantic malware detector on α_2 is sound for $\mathcal{O}_2$, and $\alpha_1(X) \subseteq \alpha_1(Y) \Rightarrow X \subseteq Y$, then the semantic malware detector on $\alpha_1 \circ \alpha_2$ is sound for $\mathcal{O}_1 \circ \mathcal{O}_2$.

PROOF. (1) Recall that the semantic malware detector on α_i is complete for $\mathcal{O}_i$ if $\mathcal{O}_i\,[\![M]\!] \hookrightarrow P \Rightarrow \exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \alpha_i(\alpha_e(\mathfrak{S}\,[\![P]\!])) \subseteq \alpha_i(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])))$.
Assume that $\mathcal{O}_1\,[\![\mathcal{O}_2\,[\![P]\!]]\!] \hookrightarrow P$, this means that there exists $lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$: $\mathfrak{S}\,[\![\mathcal{O}_1\,[\![\mathcal{O}_2\,[\![P]\!]]\!]]\!] = \alpha_r(\mathfrak{S}\,[\![P]\!])$. Since the semantic malware detector on α_1 is complete for $\mathcal{O}_1$, we have that: $\alpha_1(\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!])) \subseteq \alpha_1(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])))$. Abstraction α_2 is monotone and therefore:

$$\alpha_2(\alpha_1(\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!]))) \subseteq \alpha_2(\alpha_1(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))))$$

In general we have that $\mathcal{O}_2\,[\![M]\!] \hookrightarrow \mathcal{O}_2\,[\![M]\!]$, and since α_2 is complete we have that $\alpha_2(\alpha_e(\mathfrak{S}\,[\![M]\!])) \subseteq \alpha_2(\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!]))$. Abstraction α_1 is monotone and therefore $\alpha_1(\alpha_2(\alpha_e(\mathfrak{S}\,[\![M]\!]))) \subseteq \alpha_1(\alpha_2(\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!])))$. Since α_1 and α_2 commute we have:

$$\alpha_2(\alpha_1(\alpha_e(\mathfrak{S}\,[\![M]\!]))) \subseteq \alpha_2(\alpha_1(\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!])))$$

Thus, $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \alpha_1(\alpha_2(\alpha_e(\mathfrak{S}\,[\![M]\!]))) \subseteq \alpha_2(\alpha_1(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))))$. The proof that $\mathcal{O}_2\,[\![\mathcal{O}_1\,[\![M]\!]]\!] \hookrightarrow P$ implies that there exists $lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $\alpha_1(\alpha_2(\alpha_e\mathfrak{S}\,[\![M]\!])) \subseteq \alpha_1(\alpha_2(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))))$ is analogous.

(2) We have to prove that if $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $\alpha_1(\alpha_2(\alpha_e(\mathfrak{S}\,[\![P]\!]))) \subseteq \alpha_1(\alpha_2(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))))$ then $\mathcal{O}_1\,[\![\mathcal{O}_2\,[\![M]\!]]\!] \hookrightarrow P$.
Assume $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$: $\alpha_1(\alpha_2(\alpha_e(\mathfrak{S}\,[\![P]\!]))) \subseteq \alpha_1(\alpha_2(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))))$, since $\alpha_1(X) \subseteq \alpha_1(Y) \Rightarrow X \subseteq Y$ we have that $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $\alpha_2(\alpha_e(\mathfrak{S}\,[\![P]\!])) \subseteq \alpha_2(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])))$. The semantic malware detector on α_2 is sound by hypothesis, therefore $\mathcal{O}_2\,[\![M]\!] \hookrightarrow P$, namely there exists $lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!]) \subseteq \alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))$. Abstraction α_1 is monotone and therefore we have that $\alpha_1(\alpha_e(\mathfrak{S}\,[\![\mathcal{O}_2\,[\![M]\!]]\!])) \subseteq \alpha_1(\alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!])))$. The semantic malware detector on α_1 is sound by hypothesis and therefore $\mathcal{O}_1\,[\![\mathcal{O}_2\,[\![M]\!]]\!] \hookrightarrow P$. □

Thus, in order to propagate completeness through composition $\mathcal{O}_1 \circ \mathcal{O}_2$ and $\mathcal{O}_2 \circ \mathcal{O}_1$ the corresponding abstractions have to be independent, i.e., they have to commute. On the other side, in order to propagate soundness through composition $\mathcal{O}_1 \circ \mathcal{O}_2$ the abstraction α_1, corresponding to the last applied obfuscation, has to be an order-embedding, namely α_1 has to be both order-preserving and order-reflecting, i.e., $\alpha_1(X) \subseteq \alpha_1(Y) \Leftrightarrow X \subseteq Y$.

Observe that, when composing a non-conservative obfuscation $\mathcal{O}$, for which the semantic malware detector on $\alpha_{\mathcal{O}}$ is complete, with a conservative obfuscation $\mathcal{O}_c$, the commutation condition $\alpha_{\mathcal{O}} \circ \alpha_c = \alpha_c \circ \alpha_{\mathcal{O}}$ of point 1 of the above theorem is satisfied if and only if $(\alpha_e(\sigma) \preceq \alpha_e(\delta)) \Leftrightarrow \alpha_{\mathcal{O}}(\alpha_e(\sigma)) \preceq \alpha_{\mathcal{O}}(\alpha_e(\delta))$. In fact, only in this case α_c and $\alpha_{\mathcal{O}}$ commute, as shown by the following equations:

$$\begin{aligned}
\alpha_{\mathcal{O}}(\alpha_c[S](\alpha_e(\sigma))) &= \alpha_{\mathcal{O}}(S \cap \mathit{SubSeq}(\alpha_e(\sigma))) \\
&= \left\{\alpha_{\mathcal{O}}(\alpha_e(\delta)) \,\middle|\, \alpha_e(\delta) \in S \cap \mathit{SubSeq}(\alpha_e(\sigma))\right\} \\
&= \alpha_{\mathcal{O}}(S) \cap \left\{\alpha_{\mathcal{O}}(\alpha_e(\delta)) \,\middle|\, \alpha_e(\delta \preceq \alpha_e(\sigma))\right\}
\end{aligned}$$

$$\begin{aligned}
\alpha_c[\alpha_{\mathcal{O}}(S)](\alpha_{\mathcal{O}}(\alpha_e(\sigma))) &= \alpha_{\mathcal{O}}(S) \cap \mathit{SubSeq}(\alpha_{\mathcal{O}}(\alpha_e(\sigma))) \\
&= \alpha_{\mathcal{O}}(S) \cap \left\{\alpha_{\mathcal{O}}(\alpha_e(\delta)) \,\middle|\, \alpha_{\mathcal{O}}(\alpha_e(\delta)) \preceq \alpha_{\mathcal{O}}(\alpha_e(\sigma))\right\}
\end{aligned}$$

EXAMPLE 4. Let us consider $\mathcal{O}_v \left[\!\left[\mathcal{O}_c \left[\!\left[M\right]\!\right], \pi\right]\!\right]$ obtained by obfuscating the portion of malware M in Example 2 through variable renaming and some conservative obfuscations:

$\mathcal{O}_v \left[\!\left[\mathcal{O}_c \left[\!\left[M\right]\!\right], \pi\right]\!\right]$	
L_1	: $\texttt{assign}(D, L_B) \to L_2$
L_2	: $\texttt{skip} \to L_4$
L_c	: $\mathit{cond}(E) \to \{L_O, L_F\}$
L_4	: $\texttt{assign}(E, L_A) \to L_5$
L_5	: $\texttt{skip} \to L_c$
L_O	: $P^T \to \{L_T, L_k\}$
L_T	: $D := \mathit{Dec}(E) \to L_{T_1}$
L_{T_1}	: $\texttt{assign}(\pi_2(D), D) \to L_{T_2}$
L_{T_2}	: $\texttt{assign}(\pi_2(E), E) \to L_c$
L_k	: ...
L_F	: ...

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

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

Namely, given the abstractions α_c and α_v on which, by definition, the semantic malware detector is complete respectively for $\mathcal{O}_c$ and $\mathcal{O}_v$, the semantic malware detector on $\alpha_c \circ \alpha_v$ is complete for the composition $\mathcal{O}_v \circ \mathcal{O}_c$.

9. CASE STUDIES

We illustrate the application of our semantics-based framework to some existing malware-detection schemes. In each case, we follow the steps elucidated in Section 2, by first describing a semantic detector that, while operating on the program trace semantics, is equivalent to the malware detector of interest (i.e., they classify programs in the same way). Second we prove or disprove the soundness and completeness of the semantic detector against various obfuscation classes. As case studies, we chose a generic scan-string (signature-based) detector, the semantics-aware detector introduced in [Christodorescu et al. 2005], and the model-checking detector introduced in [Kinder et al. 2005]. We are particularly interested in a wide range of approaches to malware detection in order to underscore the flexibility and expressiveness of our framework for reasoning about such detectors.

9.1 Soundness and Completeness of a Signature-Based Detector

By investigating the effects that signature matching detection schemes have on program trace semantics we are able to certify the degree of precision of these detection schemes. We can express the signature of a malware M as a proper subset $S \subseteq M$ of "consecutive" malicious commands, formally $S = C_1, ..., C_n$ where $\forall i \in [1, n-1] : suc\,[\![C_i]\!] = lab\,[\![C_{i+1}]\!]$. Given a malware M, $S \subseteq M$ is an *ideal signature* if it unequivocally identifies infection, meaning that $S \subseteq P \Leftrightarrow M \hookrightarrow P$. Signature-based malware detectors, given an ideal signature S of a malware M (provided for example by a perfect oracle OR_S) and a possibly infected program P, syntactically verify infection according to the following test:

$$\textit{Syntactic Test}: \quad S \subseteq P$$

Let us consider the semantic counterpart of the syntactic signature matching test. Given a malware M and its signature $S \subseteq M$, let $lab_s\,[\![M]\!] = lab\,[\![S]\!]$ denote the malware restriction identifying the commands composing the signature. Observe that the semantics of the malware restricted to its signature corresponds to the semantics of the signature, i.e., $\alpha_s(\mathfrak{S}\,[\![M]\!]) = \mathfrak{S}\,[\![S]\!]$. Thus, we can say that a program P is infected by a malware M if there exists a restriction of program trace semantics that matches the semantics of the malware restricted to its signature:

$$\textit{Semantic Test}: \quad \exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \alpha_s(\mathfrak{S}\,[\![M]\!]) = \alpha_r(\mathfrak{S}\,[\![P]\!])$$

which can be equivalently expressed as $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \mathfrak{S}\,[\![S]\!] = \alpha_r(\mathfrak{S}\,[\![P]\!])$. The following result shows that the syntactic and semantic tests are equivalent, meaning that they detect the same set of infected programs.

PROPOSITION 1. Given a signature S of a malware M we have that:

$$S \subseteq P \;\Leftrightarrow\; \exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \mathfrak{S}\,[\![S]\!] = \alpha_r(\mathfrak{S}\,[\![P]\!])$$

PROOF. ($\Rightarrow$) $S \subseteq P$ means that $\forall C \in S \Rightarrow C \in P$, namely that $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : P_r = S$. Therefore, $\alpha_r(\mathfrak{S}\,[\![P]\!]) = \mathfrak{S}\,[\![P_r]\!] = \mathfrak{S}\,[\![S]\!]$. ($\Leftarrow$) If $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \mathfrak{S}\,[\![S]\!] = \alpha_r(\mathfrak{S}\,[\![P]\!])$, it means that $|\mathfrak{S}\,[\![S]\!]| = |\alpha_r(\mathfrak{S}\,[\![P]\!])|$ and that $\forall \sigma \in \mathfrak{S}\,[\![S]\!]$, $\exists \delta \in \alpha_r(\mathfrak{S}\,[\![P]\!])$ such that $\sigma = \delta = (C_1, (\rho_1, m_1)), ..., (C_k, (\rho_k, m_k))$. This means that for every $\sigma \in \mathfrak{S}\,[\![S]\!]$ and $\delta \in \alpha_r(\mathfrak{S}\,[\![P]\!])$ such that $\sigma = \delta$, we have that $cmd\,[\![\sigma]\!] = \cup_{i \in [1,k]} C_i = cmd\,[\![\delta]\!]$, and therefore $S = cmd(\mathfrak{S}\,[\![S]\!]) = cmd(\mathfrak{S}\,[\![P_r]\!]) \subseteq P$, namely $S \subseteq P$. $\square$

Observe that by applying abstraction α_e to the semantic test we have that $M \hookrightarrow P$ if:

$$\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]) : \alpha_e(\alpha_s(\mathfrak{S}\,[\![S]\!])) = \alpha_e(\alpha_r(\mathfrak{S}\,[\![P]\!]))$$

which corresponds to the standard infection condition specified by Definition 4 where the semantics of malware M has been restricted to its signature S and the set-inclusion relation has been replaced by equality. It is clear that, in this setting, by replacing $\mathfrak{S}\,[\![M]\!]$ with $\mathfrak{S}\,[\![S]\!]$ we can obtain results analogous to the one proved following Definition 4 of infection.

Proving Soundness and Completeness of a Signature-based Detector

First of all we need to define a trace-based malware detector that is equivalent to the signature-based algorithm. Next, this semantic formalization is used to prove soundness and completeness of the signature based approach.

Step 1: Designing an equivalent trace-based detector. This point is actually solved by Proposition 1. In fact, let $\mathcal{A}_S$ denote the malware detector based on the signature matching algorithm. This syntactic algorithm is based on an oracle OR_S that, given a malware M, returns its ideal signature S such that: $S \subseteq P \;\Leftrightarrow\; M \hookrightarrow P$, or, equivalently, $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \mathfrak{S} \llbracket S \rrbracket = \alpha_r(\mathfrak{S} \llbracket P \rrbracket) \;\Leftrightarrow\; M \hookrightarrow P$. Let D_S be the trace-based detector that classifies a program P as infected by a malware M with signature S, if $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \mathfrak{S} \llbracket S \rrbracket = \alpha_r(\mathfrak{S} \llbracket P \rrbracket)$. From Proposition 1 it follows that $\mathcal{A}_S(M, P) = 1$ if and only if $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \mathfrak{S} \llbracket S \rrbracket = \alpha_r(\mathfrak{S} \llbracket P \rrbracket)$ if and only if $D_S(M, P) = 1$.

Step 2: Prove soundness and completeness of D_S. Let us identify the class of obfuscating transformations that the trace-based detector D_S is able to handle. The following result shows that D_S is sound with respect to every obfuscation if the signature oracle OR_S is perfect, namely D_S is oracle-sound with respect to every obfuscation. Observe that here we are assuming that the identity function is an obfuscator.

PROPOSITION 2. D_S is oracle-sound with respect to every obfuscation.

PROOF. Given a malware M with signature S we have that: $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \mathfrak{S} \llbracket S \rrbracket = \alpha_r(\mathfrak{S} \llbracket P \rrbracket) \;\Rightarrow\; M \hookrightarrow P$, follows from the hypothesis that OR_S is a perfect oracle that returns an ideal signature. □

This confirms the general belief that signature matching algorithms have a low false positive rate. In fact, the presence of false positives is caused by the imperfection in the signature extraction process, meaning that in order to improve the signature matching algorithm we have to concentrate in the design of efficient techniques for signature extraction.

Let us introduce the class $\mathbb{O}_S$ of obfuscating transformations that *preserve signatures*. We say that $\mathcal{O}$ preserves signatures, i.e., $\mathcal{O} \in \mathbb{O}_S$, when for every malware M with signature S the semantics of signature S is present in the semantics of the obfuscated malware $\mathcal{O} \llbracket M \rrbracket$, formally when:

$$\mathfrak{S} \llbracket S \rrbracket = \alpha_s(\mathfrak{S} \llbracket M \rrbracket) \;\Rightarrow\; \begin{cases} \exists lab_R \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket \in \wp(lab \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket) : \\ \quad \mathfrak{S} \llbracket S \rrbracket = \alpha_R(\mathfrak{S} \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket) \end{cases} \qquad (\ddagger)$$

The above condition can equivalently be expressed in syntactic terms as

$$S \subseteq M \;\Rightarrow\; S \subseteq \mathcal{O} \llbracket M \rrbracket$$

The following result shows that D_S is oracle-complete for $\mathcal{O}$ if and only if $\mathcal{O}$ preserves signatures.

PROPOSITION 3. D_S is oracle-complete for $\mathcal{O} \Leftrightarrow \mathcal{O} \in \mathbb{O}_S$.

PROOF. ($\Leftarrow$) Assume that $\mathcal{O} \in \mathbb{O}_S$, then we have to show that: $\mathcal{O} \llbracket M \rrbracket \hookrightarrow P \;\Rightarrow\; \exists lab_{\mathbf{R}} \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \mathfrak{S} \llbracket S \rrbracket = \alpha_{\mathbf{R}}(\mathfrak{S} \llbracket P \rrbracket)$. Observe that $\mathcal{O} \llbracket M \rrbracket \hookrightarrow P$, means that $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : P_r = \mathcal{O} \llbracket M \rrbracket$, namely $\exists lab_r \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \alpha_r(\mathfrak{S} \llbracket P \rrbracket) = \mathfrak{S} \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket$. From (‡), we have that $\exists lab_R \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket \in \wp(lab \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket) : \mathfrak{S} \llbracket S \rrbracket = \alpha_R(\mathfrak{S} \llbracket \mathcal{O} \llbracket M \rrbracket \rrbracket)$, and therefore $\mathfrak{S} \llbracket S \rrbracket = \alpha_R(\alpha_r(\mathfrak{S} \llbracket P \rrbracket)) = \alpha_{\mathbf{R}}(\mathfrak{S} \llbracket P \rrbracket)$. ($\Rightarrow$) Assume that D_S is complete for $\mathcal{O}$, this means that $\mathcal{O} \llbracket M \rrbracket \hookrightarrow P \Rightarrow \exists lab_{\mathbf{R}} \llbracket P \rrbracket \in \wp(lab \llbracket P \rrbracket) : \mathfrak{S} \llbracket S \rrbracket \subseteq \alpha_{\mathbf{R}}(\mathfrak{S} \llbracket P \rrbracket)$, meaning that there is a restriction of program P that matches signature S. Thus, program P can be restricted to a signature preserving transformation of M. □

This means that a signature based detection algorithm $\mathcal{A}_S$ is oracle-complete with respect to the class of obfuscations that preserve malware signatures, namely the ones belonging to $\mathbb{O}_S$. Unfortunately, a lot of commonly used obfuscating transformations do not preserve signatures, namely are not in $\mathbb{O}_S$. Consider for example the code reordering obfuscation $\mathcal{O}_J$. It is easy to show that $\mathcal{A}_S$ is not complete for $\mathcal{O}_J$. In fact, given a malware M with signature $S \subseteq M$, we have that, in general, $S \not\subseteq \mathcal{O}_J [\![M]\!]$, since jump instructions are inserted between the signature commands changing therefore the signature. In particular, consider signature $S \subseteq M$ such that $S = C_1, ..., C_n$ we have that $S \not\subseteq \mathcal{O}_J [\![M]\!]$, while $S' \subseteq \mathcal{O}_J [\![M]\!]$, where $S' = C'_1 J^* C'_2 J^* ... J^* C'_n$, where J denotes a command implementing an unconditional jump, namely of the form $L : \texttt{skip} \rightarrow L'$, and C'_i is given by command C_i with labels updated according to jump insertion. This means that when $\mathcal{O}_J [\![M]\!] \hookrightarrow P$ then $\forall lab_R [\![\mathcal{O} [\![M]\!]]\!] \in \wp(lab [\![\mathcal{O} [\![M]\!]]\!]) : \mathfrak{S} [\![S]\!] \not\subseteq \alpha_R(\mathfrak{S} [\![\mathcal{O} [\![M]\!]]\!])$. Observe that incompleteness is caused by the fact that D_S, being equivalent to $\mathcal{A}_S$, is strongly related to program syntax, and therefore the insertion of an innocuous jump instruction is able to confuse it.

Following the same strategy, it is possible to show that $\mathcal{A}_S$ is not complete for opaque predicate insertion, semantic NOP insertion and substitution of equivalent commands. Thus, in general, the class of conservative transformations does not preserve malware signatures, i.e., $\mathbb{O}_c \not\subseteq \mathbb{O}_S$, meaning that conservative obfuscations are able to foil signature matching algorithms. Hence, it turns out that $\mathcal{A}_S$ is not complete, namely it is imprecise, for a wide class of obfuscating transformations. This is one of the major drawbacks of signature-based approaches. A common improvement of $\mathcal{A}_S$ consists in considering regular expressions instead of signatures. Namely, given a signature $S = C_1, ..., C_n$, the detector $\mathcal{A}_S^+$ verifies if $C'_1 C^* C'_2 C^* ... C^* C'_n \subseteq P$, where C stands for any command in $\mathbb{C}$ and C'_i is a command with the same action as C_i. It is clear that this allows $\mathcal{A}_S^+$ to deal with the class of obfuscating transformations that are conservative with respect to signatures, as for example code reordering $\mathcal{O}_J$. Let $\mathbb{O}_{cs}$ denote the class of obfuscations that are conservative with respect to signatures, where $\mathcal{O} \in \mathbb{O}_{cs}$ if for every malware M with signature S there exists $S' \subseteq \mathcal{O} [\![M]\!]$ such that $S = C_1 C_2 ... C_n$ and $S' = C'_1 C^* C'_2 C^* ... C^* C'_n$. However, this improvement does not handle all conservative obfuscations in $\mathbb{O}_c$. For example, the substitution of equivalent commands $\mathcal{O}_I$ belongs to $\mathbb{O}_c$ but not to $\mathbb{O}_{cs}$.

9.2 Completeness of Semantics-Aware Malware Detector $\mathcal{A}_{MD}$

An algorithm called *semantics-aware malware detection* was proposed in [Christodorescu et al. 2005]. This approach to malware detection uses instruction semantics to identify malicious behavior in a program, even when obfuscated.

The obfuscations considered in [Christodorescu et al. 2005] 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 I) is based on the list described in the semantics-aware malware detection paper.

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 $\mathcal{A}_{MD}$ attempts to find a subset of the program P that matches the com-

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

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

mands 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 $C = L_1 : A \to S$, 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 $\mathscr{C}^k [\![P|_\theta]\!] (C_1, (\rho, m))$, 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:

$$\begin{aligned} mem\,[\![(C, (\rho, m))]\!] &= m \\ env\,[\![(C, (\rho, m))]\!] &= \rho \end{aligned}$$

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 $C\left[\!\left[n^T\right]\!\right]$ according to the map ν_θ, we obtain the program command $C\left[\!\left[n^P\right]\!\right] = C\left[\!\left[n^T\right]\!\right] [X/\nu_\theta\left(X, n^T\right)]$.
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 $act\left[\!\left[C_m^T\right]\!\right]$ and used in $act\left[\!\left[C_n^T\right]\!\right]$, 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

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

Oracle	*Notation*	*Description*
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.

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 $T|_\theta = \left\{C\,[\![m^T]\!], C_1^T, \ldots, C_k^T, C\,[\![n^T]\!]\right\}$ the sequence of commands corresponding to the template path θ.

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

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

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 II. Thus $\mathcal{A}_{MD} = D^{\mathcal{OR}}$, for the set of oracles $\mathcal{OR} = \{OR_{CFG}, OR_{SNop}\}$. Our goal is then to verify whether $\mathcal{A}_{MD}$ is $\mathcal{OR}$-complete with respect to the obfuscations from Table I. Since three of those obfuscations (code reordering, semantic-nop insertion, and substitution of equivalent commands) are conservative, we only need to check $\mathcal{OR}$-completeness of $\mathcal{A}_{MD}$ for each individual obfuscation. We would then know (from Lemma 1) if $\mathcal{A}_{MD}$ is also $\mathcal{OR}$-complete with respect to any combination of these obfuscations.

We follow the proof strategy proposed in Section 2. 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.

When applying α_{SAMD} to a trace $\sigma \in \mathfrak{S}\,[\![P]\!]$, $\sigma = (C'_1, (\rho'_1, m'_1)) \ldots (C'_n, (\rho'_n, m'_n))$, to a set of variable maps $\{\pi_i\}$, and a set of location maps $\{\gamma_i\}$, we obtain the following abstract trace:

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

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

Let us define abstraction α_{Act} simply strips all labels from the commands in a trace $\sigma = (C_1,(\rho_1,m_1))\sigma'$, as follows:

$$\alpha_{Act}(\sigma) = \begin{cases}\epsilon & \text{if } \sigma=\epsilon\\ (\mathit{act}\,[\![C_1]\!],(\rho_1,m_1))\alpha_{Act}(\sigma') & \mathit{otherwise}\end{cases}$$

DEFINITION 12. An *α-semantic malware detector* is a malware detector on the abstraction α, i.e., it classifies the program P as infected by a malware M, $M \hookrightarrow P$, if

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

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 4. 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 $\mathcal{A}_{MD}(P,M) = D_{Tr}(\mathfrak{S}\,[\![P]\!],\mathfrak{S}\,[\![M]\!])$.

PROOF. To show that $\mathcal{A}_{MD} = D_{Tr}$, we can equivalently show that for all programs $P, M \in \mathbf{P}$ $\mathcal{A}_{MD}(P,M) = 1 \iff \exists \mathit{lab}_r\,[\![P]\!] \in \wp(\mathit{lab}\,[\![P]\!])$, $\exists\{\pi_i\}_{i\ge1}$, and $\exists\{\gamma_i\}_{i\ge1}$ such that $\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!],\{\pi_i\},\{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}\,[\![P]\!]),\{\pi_i\},\{\gamma_i\}))$. Since π_i renames variables only from P (i.e., $\forall V \in \mathbf{V}\setminus \mathit{var}\,[\![P]\!]$, π_i is the identity function, namely $\forall X : \pi_i(X) = X$), and similarly γ_i remaps locations only from P, then we have that $\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!],\{\pi_i\},\{\gamma_i\}) = \mathfrak{S}\,[\![M]\!]$.

$(\Rightarrow)$ Assume that $\mathcal{A}_{MD}(P,M) = 1$. Let G^M be the CFG of malware M and let $\mathit{Path}(G^M)$ denote the set of all paths on G^M. We can construct the restriction $\mathit{lab}_r\,[\![P]\!]$ from the path-sensitive map μ_θ as follows:

$$\mathit{lab}_r\,[\![P]\!] = \bigcup_{\theta\in \mathit{Paths}(G^M)} \left\{\mathit{lab}\,\left[\!\left[C\,\left[\!\left[\mu_\theta\left(v^M\right)\right]\!\right]\right]\!\right]\,|v^M\in\theta\right\}$$

Following the above construction $\mathit{lab}_r\,[\![P]\!]$ collects the labels of program commands whose nodes corresponds to a template node through μ_θ. The variable maps $\{\pi_i\}$ can be defined based on ν_θ. For a path $\theta = v_1^M \to \dots \to v_k^M$, $\pi_i(X) = \nu_\theta\left(X, v_i^M\right)$. Similarly, $\gamma_i(L) = L'$ if $\mathit{lab}\,\left[\!\left[C\,\left[\!\left[v_i^M\right]\!\right]\right]\!\right] = L'$ and $\mathit{lab}\,\left[\!\left[C\,\left[\!\left[\mu_\theta\left(v_i^M\right)\right]\!\right]\right]\!\right] = L$.

Let $\sigma \in \mathfrak{S}\,[\![M]\!]$ and denote by $\theta = v_1^M \to \dots \to v_k^M$ 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:

$$\dots \to \mu_\theta\left(v_1^M\right) \to \dots \to \mu_\theta\left(v_k^M\right) \to \dots$$

Let $\delta \in \alpha_r(\mathfrak{S}\llbracket P \rrbracket)$ be the trace corresponding to the path χ in G^P,

$$\delta = \ldots \left\langle C\left\llbracket \mu_\theta\left(v_1^M\right)\right\rrbracket, (\rho_1^P, m_1^P)\right\rangle \ldots \left\langle C\left\llbracket \mu_\theta\left(v_k^M\right)\right\rrbracket, (\rho_k^P, m_k^P)\right\rangle \ldots$$

For two states i and $j > i$ of the trace σ, denote the intermediate states in the trace δ by $\left\langle C_1'^P, (\rho_1'^P, m_1'^P)\right\rangle \ldots \left\langle C_l'^P, (\rho_l'^P, m_l'^P)\right\rangle$, i.e., $\delta =$

$$..\left\langle C\left\llbracket \mu_\theta\left(v_i^M\right)\right\rrbracket, (\rho_i^P, m_i^P)\right\rangle \left\langle C_1'^P, (\rho_1'^P, m_1'^P)\right\rangle \ldots \left\langle C_l'^P, (\rho_l'^P, m_l'^P)\right\rangle \left\langle C\left\llbracket \mu_\theta\left(v_j^M\right)\right\rrbracket, (\rho_j^P, m_j^P)\right\rangle ..$$

From step 1 of algorithm $\mathcal{A}_{MD}$, we have that the following holds:

$$\begin{aligned} act\left\llbracket C\left\llbracket \mu_\theta\left(v_i^M\right)\right\rrbracket\right\rrbracket [X/\pi_i(X)] &= act\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket \\ \gamma_i\left(lab\left\llbracket C\left\llbracket \mu_\theta\left(v_i^M\right)\right\rrbracket\right\rrbracket\right) &= lab\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket \\ \gamma_i\left(suc\left\llbracket C\left\llbracket \mu_\theta\left(v_i^M\right)\right\rrbracket\right\rrbracket\right) &= suc\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket \end{aligned}$$

From step 2 of algorithm $\mathcal{A}_{MD}$, we know that for any template variable X^M that is defined in $C\left\llbracket v_i^M\right\rrbracket$ and used in $C\left\llbracket v_j^M\right\rrbracket$ (for $1 \le i < j \le k$), we have that:

$$\mathscr{E}\left\llbracket \nu_\theta(X^M, v_i^M)\right\rrbracket (\rho, m) = \mathscr{E}\left\llbracket \nu_\theta(X^M, v_j^M)\right\rrbracket (env\llbracket s \rrbracket, mem\llbracket s \rrbracket)$$

where $s \in \mathscr{C}^l\left(\left\langle \mu\left(v_i^M\right)\right\rangle, (\rho, m)\right)$. Since we have that $act\left\llbracket C\left\llbracket \mu_\theta\left(v_i^M\right)\right\rrbracket\right\rrbracket [X/\pi_i(X)] = act\left\llbracket C\left\llbracket v_i^M\right\rrbracket\right\rrbracket$, it follows that $\rho_i^P(\nu_\theta(X^M, v_i^M)) = \rho_j^P(\nu_\theta(X^M, v_j^M))$. Moreover, 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, $m_i^M = m_i^P \circ \gamma_i$. Then it follows that for every $\sigma \in \mathfrak{S}\llbracket M \rrbracket$, there exists $\delta \in \alpha_r(\mathfrak{S}\llbracket P \rrbracket)$ such that:

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

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

$(\Leftarrow)$ Assume that $lab_r\llbracket P \rrbracket$, $\{\pi_i\}_{i\ge1}$, and $\{\gamma_i\}_{i\ge1}$ exist such that:

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

We will show that $\mathcal{A}_{MD}$ returns 1, 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 $\forall i$, $act\left\llbracket C_i^M\right\rrbracket [X/\pi_i(X)] = A_i$. Similarly, there exists $\delta \in \alpha_r(\mathfrak{S}\llbracket P \rrbracket)$, with $\delta = \left\langle C_1^P, (\rho_1^P, m_1^P)\right\rangle \ldots \left\langle C_k^P, (\rho_k^P, m_k^P)\right\rangle$, such that $\forall i$, $act\left\llbracket C_i^P\right\rrbracket [X/\pi_i(X)] = A_i$, $\rho_i^P = \rho_i^M \circ \pi_i^{-1}$, 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 $\mu(v_{lab\llbracket C_i^M \rrbracket}) = v_{lab\llbracket C_i^P \rrbracket}$. 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 $X^M \in var\,[\![M]\!]$ that is defined by action A_i and later used by action A_j in the trace σ', for $j > i$, such that $\rho^M_{i+1}(X^M) = \rho^M_j(X^M)$. Let X^P_i be the program variable corresponding to X^M at program command C^P_i, and X^P_j the program variable corresponding to X^M at program command C^P_j:

$$x^P_i = \nu(X^M, v_{lab[\![C^M_i]\!]}) \qquad\qquad x^P_j = \nu(X^M, v_{lab[\![C^M_j]\!]})$$

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

$$\delta' = \ldots\langle C^P_i, (\rho^P_i, m^P_i)\rangle \ldots \langle C^P_j, (\rho^P_j, m^P_j)\rangle \ldots$$

where $1 \le i < j \le k$. Let θ be a path in the *CFG* of P, $\theta = v^P_1 \to \ldots \to v^P_k$, such that $v^P_{lab[\![C^P_i]\!]} \to v^P_1 \to \ldots \to v^P_k \to v^P_{lab[\![C^P_j]\!]}$ is also a path in the *CFG* of P. Since $\rho^M_{i+1}(X^M) = \rho^M_j(X^M)$, then $\rho^P_{suc[\![C^P_i]\!]}(X^P_i) = \rho^M_{i+1}(\pi_i(X^P_i)) = \rho^M_{i+1}(X^M) = \rho^M_j(X^M) = \rho^P_j(\pi_j(X^P_j)) = \rho^P_j(X^P_j)$. But $suc\,[\![C^P_i]\!] = lab\,[\![C^P\,[\![v_1]\!]]\!]$ in the trace δ'. As $\mathscr{E}\,[\![X^P_i]\!]\,(\rho, m) = \rho(x^P_i)$, it follows that

$$\mathscr{E}\left[\!\!\left[\nu(X^M, v_{lab[\![C^M_i]\!]})\right]\!\!\right](\rho, m) = \mathscr{E}\left[\!\!\left[\nu(X^M, v_{lab[\![C^M_j]\!]})\right]\!\!\right](env\,[\![s]\!]\,, mem\,[\![s]\!])$$

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

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 13. According to the *semantics-aware malware detection* algorithm a program P is infected by a vanilla malware M, i.e. $M \hookrightarrow P$, if:

$$\begin{aligned}&\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]),\ \{\pi_i\}_{i\ge1},\ \{\gamma_i\}_{i\ge1}\ :\\ &\quad \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!]\,, \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}\,[\![P]\!]), \{\pi_i\}, \{\gamma_i\})).\end{aligned}$$

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

$$\mathcal{O}_J(M) \hookrightarrow P \Rightarrow \begin{cases} \exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!]\,,\ \{\pi_i\}_{i\ge1},\ \{\gamma_i\}_{i\ge1} : \\ \quad \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!]\,, \{\pi_i\}, \{\gamma_i\})) \subseteq \\ \quad\quad \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}\,[\![P]\!]), \{\pi_i\}, \{\gamma_i\})) \end{cases}$$

PROOF. If $\mathcal{O}_J\,[\![M]\!] \hookrightarrow P$, and given that $\mathcal{O}_J$ inserts only `skip` commands into a program, then $\exists lab_r\,[\![P]\!] \in \wp(lab\,[\![P]\!])$ such that $P_r = \mathcal{O}_J\,[\![M]\!] \setminus Skip$, where *Skip* is a set of `skip` commands inserted by $\mathcal{O}_J$, as defined in Section 6. Let $M' = \mathcal{O}_J(M) \setminus Skip$. Then $\alpha_r(\mathfrak{S}\,[\![P]\!]) = \mathfrak{S}\,[\![M']\!]$. Thus we have to prove that

$$\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!]\,, \{\pi_i\}, \{\gamma_i\})) \subseteq \alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\,[\![M']\!]\,, \{\pi_i\}, \{\gamma_i\}))$$

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, $\pi_i = Id_{var[\![P]\!]}$ and $\gamma_j = Id_{lab[\![P]\!]}$. It follows that $\alpha_{SAMD}(\mathfrak{S}\,[\![M']\!], \{Id_{var[\![P]\!]}\}, \{Id_{lab[\![P]\!]}\}) = \mathfrak{S}\,[\![M']\!]$ and $\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!], \{Id_{var[\![P]\!]}\}, \{Id_{lab[\![P]\!]}\}) = \mathfrak{S}\,[\![M]\!]$. Thus, it remains to show that $\alpha_{Act}(\mathfrak{S}\,[\![M]\!]) \subseteq \alpha_{Act}(\mathfrak{S}\,[\![M']\!])$. By the definition of $\mathcal{O}_J$, we have that $M' = \mathcal{O}_J\,[\![M]\!] \setminus \mathit{Skip} = (M \setminus S) \cup \eta(S)$, for some $S \subset M$. But $\eta(S)$ only updates the labels of the commands in S, and thus we have:

$$\begin{aligned}\alpha_{Act}(\mathfrak{S}\,[\![M']\!]) &= \alpha_{Act}(\mathfrak{S}\,[\![(M \setminus S) \cup \eta(S)]\!]) \\ &= \alpha_{Act}(\mathfrak{S}\,[\![M]\!]).\end{aligned}$$

It follows that $\alpha_{Act}(\mathfrak{S}\,[\![M]\!]) \subseteq \alpha_{Act}(\mathfrak{S}\,[\![\mathcal{O}_J\,[\![M]\!] \setminus \mathit{Skip}]\!])$. □

Similar proofs confirm that D_{Tr} is $\mathcal{OR}$-complete w.r.t. variable renaming and semantic-nop insertion. 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 obfuscations, as the following result illustrates.

PROPOSITION 6. A semantics-aware malware detector is not complete on α_{SAMD} w.r.t. all conservative obfuscations $\mathcal{O}_c \in \mathbb{O}_c$.

PROOF. 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

$$\alpha_{Act}(\alpha_{SAMD}(\mathfrak{S}\,[\![M]\!], \{\pi_i\}, \{\gamma_i\})) \subseteq \\ \alpha_{Act}(\alpha_{SAMD}(\alpha_r(\mathfrak{S}\,[\![\mathcal{O}_c(M)]\!]), \{\pi_i\}, \{\gamma_i\})) \quad (3)$$

cannot hold for any restriction $lab_r\,[\![\mathcal{O}_c\,[\![M]\!]]\!] \in \wp(lab\,[\![\mathcal{O}_c\,[\![M]\!]]\!])$ 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., $\mathfrak{S}\,[\![P]\!] \cap \mathfrak{S}\,[\![\mathcal{O}_I\,[\![P]\!]]\!] = \emptyset$) – 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 3 holds, where $\mathcal{O}_c = \mathcal{O}_I$. Then $\exists\sigma \in \mathfrak{S}\,[\![M]\!]$ and $\exists\delta \in \mathfrak{S}\,[\![\mathcal{O}_I\,[\![M]\!]]\!]$ 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 $\forall i$, $act\,[\![C_i]\!] = act\,[\![C'_i]\!]\,[X/\pi_i(X)]$. But from the definition of the obfuscating transformation $\mathcal{O}_I$ above, we know that $\forall\sigma \in \mathfrak{S}\,[\![M]\!]$, $\forall\delta \in \mathfrak{S}\,[\![\mathcal{O}_I\,[\![M]\!]]\!]$, $\exists i \geq 1$ such that $C_i \in \sigma$, $C'_i \in \delta$, and $\forall\pi : \mathbf{X} \to \mathbf{X}$, $act\,[\![C_i]\!] \neq act\,[\![C'_i]\!]\,[X/\pi(X)]$. Hence we have a contradiction. □

The cause for this incompleteness is the fact that the abstraction applied by D_{Tr} still preserves some of the actions from the program. 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 malware (i.e., $\mathfrak{S}\,[\![M]\!] \cap \mathfrak{S}\,[\![\mathcal{O}_I(M)]\!] = \emptyset$). For example, the transformation could modify the command at M's start label. Such an obfuscation, because it affects at least one action of M on every path through the program $P = \mathcal{O}_I(M)$, will defeat the detector.

9.3 Soundness and Completeness of a Model Checking-Based Detector

In this section we study the soundness and completeness of another malware-detection algorithm, based on model checking against a specification language called *CTPL*, introduced by Kinder, Katzenbeisser, Schallhart, and Veith [2005] as an extension to Computation Tree Logic (CTL).

The model checking-based detection algorithm, which we denote $\mathcal{A}_{CTPL}$, is an extension of the classic model-checking algorithm [Clarke, Jr. et al. 2001]. The program P is modeled as a Kripke structure K_P which is then checked against a specification of malicious behavior given as a CTPL formula ψ. The Kripke structure K_P is derived from the interprocedural control-flow graph of the disassembled binary, with one state per program instruction. K_P states are labeled with two atomic predicates, one for the corresponding program instruction, written $\mathtt{instr}(p_1, \ldots, p_n)$, and one for the corresponding program location, written $\#\mathrm{loc}(L)$. A program P is malicious if $K_P, s_0 \models \psi$, where s_0 is an initial state. $K_P, s_0 \models \psi$ is also written $K_P \models \psi$.

The semantics of a model K_P of a program P is derived from the semantics of P. An abstraction can represent the projection of $\mathfrak{S}\,[\![P]\!]$ into $\mathfrak{S}\,[\![K_P]\!]$. Let α_{Cmd} be the abstraction that computes a command trace from a regular trace: $\sigma = (C_1, (\rho_1, m_1))\sigma'$:

$$\alpha_{Cmd}(\sigma) = \begin{cases} \epsilon & \text{if } \sigma = \epsilon \\ C_1 \alpha_{Cmd}(\sigma') & \text{otherwise.} \end{cases}$$

By abuse of notation, we will extend α_{Cmd} to sets of traces and write, without introducing ambiguity, $\alpha_{Cmd}(A) = \bigcup_{\sigma \in A} \alpha_{Cmd}(\sigma)$. Thus $\mathfrak{S}\,[\![K_P]\!] = \alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])$, by the construction of K_P as given by Kinder *et al.* [2005].

Given a CTPL formula ψ, we would like to construct an abstraction α_ψ such that $K_P \models \psi \iff \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]) \neq \emptyset$. In other words, a model that satisfies a malicious CTPL formula has a non-empty set of abstract traces, obtained through the abstraction constructed from the malicious CTPL formula. In this context we assume that the CTPL formula ψ was provided by an oracle OR_{CTPL} from some malware M, i.e., if $\psi = OR_{CTPL}(M)$, then $K_P \models \psi \iff M \hookrightarrow P$.

We ease the task of designing the abstraction α_ψ by using the fact that the CTL operators $\mathbf{EX}$, $\mathbf{EG}$, and $\mathbf{EU}$, together with the logical operators $\neg$ and $\wedge$, are sufficient to express all CTL formulas [Clarke, Jr. et al. 2001]. Thus we do not need to consider all 13 CTL and logical operators. CTPL adds to CTL free variables that appear as possible arguments to the instruction predicates and to location predicates. The universe of values for the free variables is finite. Thus each CTPL formula can be converted into a disjunction of CTL formulas over constants by simply iterating over the universe of values. If ψ is a CTPL formula, then $K_P \models \psi \iff K_P \models \bigvee_{(v_1,\ldots,v_k)\in\mathcal{U}^k} \psi[x_i/v_i]$, where x_i represent the free variables in the CTPL formula ψ, $\mathcal{U}$ is the universe of values for a free variable, and $\psi[x_i/v_i]$ is a CTL formula obtained from ψ by instantiating the free variable x_i with a value $v_i \in \mathcal{U}$. For our purposes, it is sufficient to focus on the three core CTL operators and the two core logical operators.

The abstraction α_ψ can be defined recursively on the structure of the CTPL formula ψ. Such a construction is well defined since CTPL formulas have finite length. If A is a set of sequences, $A = \{\langle x_0, \ldots, x_i, \ldots\rangle\}$, we write *Next*$^{k+1}(A)$ for the set of sequences $\{y : \langle x_0, \ldots, x_k\rangle y \in A\}$, for all $k \geq 0$. The abstraction α_ψ operates on a set of traces, e.g., $A = \mathfrak{S}\,[\![K_P]\!]$, which are simply sequences of commands. If the set A is empty, then the abstraction α_ψ returns the empty set, $\alpha_\psi(\emptyset) = \emptyset$ for any CTPL formula ψ. Otherwise, the abstraction is defined with respect to the particular structure of ψ.

$$\begin{aligned}
\alpha_{\mathbf{EX}\psi}(A) &= \left\{\sigma \in A : \sigma = s\sigma' \wedge \sigma' \in \alpha_\psi\left(\mathit{Next}^1(A)\right)\right\}\\
\alpha_{\mathbf{EG}\psi}(A) &= \left\{\sigma \in A : \forall k \geq 0 \,.\, \sigma = s_0 \dots s_k\sigma' \wedge \sigma' \in \alpha_\psi\left(\mathit{Next}^{k+1}(A)\right)\right\}\\
\alpha_{\mathbf{E}[\psi\mathbf{U}\psi']}(A) &= \left\{\sigma \in A : \exists k \geq 0 \,.\, \begin{array}{l}\forall 0 \leq j < k \,.\, \alpha_\psi\left(\mathit{Next}^j(A)\right) \neq \emptyset\\ \wedge\\ \alpha_{\psi'}\left(\mathit{Next}^k(A)\right) \neq \emptyset\end{array}\right\}\\
\alpha_{\neg\psi}(A) &= A \setminus (\alpha_\psi(A))\\
\alpha_{\psi\wedge\psi'}(A) &= \alpha_\psi(A) \cap \alpha_{\psi'}(A)\\
\alpha_{\mathtt{instr}(p_1,\dots,p_n)}(A) &= \{\sigma \in A : \sigma = s\sigma' \wedge \mathit{act}\,[\![s]\!] = \mathtt{instr}(p_1, \dots, p_n)\}\\
\alpha_{\#\mathrm{loc}(L)}(A) &= \{\sigma \in A : \sigma = s\sigma' \wedge \mathit{lab}\,[\![\mathit{cmd}\,[\![\{s\}]\!]]\!] = L\}
\end{aligned}$$

Let us write $\mathfrak{S}\,[\![P]\!]\,(s)$ to mean the trace semantics of P when the starting state is s. In other words, $\mathfrak{S}\,[\![P]\!]\,(s) = \{\sigma \in \mathfrak{S}\,[\![P]\!] : \sigma = s\sigma'\}$. We note that we can express the trace semantics of P as the union of trace semantics starting in any start state, $\mathfrak{S}\,[\![P]\!] = \bigcup_{s_0 \in \mathit{Init}} \mathfrak{S}\,[\![P]\!]\,(s_0)$, where Init is the set of initial states of P. Furthermore, if $\sigma = s_0 s_1 \dots s_k \sigma'$, then $\mathfrak{S}\,[\![P]\!]\,(s_k) = \mathit{Next}^k(\mathfrak{S}\,[\![P]\!]\,(s_0))$.

PROPOSITION 7. $K_P, s \models \psi \iff \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$.

PROOF. ($\Rightarrow$) Assume $K_P, s \models \psi$. We perform the proof by induction on the structure of ψ.

—$K_P, s \models \mathbf{EX}\psi$
Then a trace $\sigma \in \mathfrak{S}\,[\![K_P]\!]\,(s)$ exists that has at least two states, $\sigma = s\sigma' = ss'\sigma''$, such that $K_P, s' \models \psi$. By inductive hypothesis, $K_P, s' \models \psi$ implies that $\alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s')) \neq \emptyset$. Then there exists $\bar{\sigma} \in \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s'))$. Note that $s'\mathfrak{S}\,[\![K_P]\!] \subseteq \mathit{Next}^1(\mathfrak{S}\,[\![K_P]\!]\,(s))$ and we have that $\bar{\sigma} \in \alpha_\psi(\mathit{Next}^1(\mathfrak{S}\,[\![K_P]\!]\,(s)))$. As a result, because there exists $\delta \in \mathfrak{S}\,[\![K_P]\!]\,(s)$ such that $\delta = s\bar{\sigma}$, we have that $\delta \in \alpha_{\mathbf{EX}\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s))$. Thus, $\alpha_{\mathbf{EX}\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$.

—$K_P, s \models \mathbf{EG}\psi$
Then a trace $\sigma \in \mathfrak{S}\,[\![K_P]\!]\,(s)$ exists such that $\sigma = s_0 s_1 \dots s_i \dots$ and $K_P, s_i \models \psi$, for all $i \geq 0$. Note that $s_0 = s$. We have that if $K_P, s_i \models \psi$, then $\alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_i)) \neq \emptyset$ (by the induction hypothesis). Then there exists $\bar{\sigma} \in \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_i))$. As $\mathfrak{S}\,[\![K_P]\!]\,(s_i) = \mathit{Next}^i(\mathfrak{S}\,[\![K_P]\!]\,(s_0))$, then $\bar{\sigma} \in \alpha_\psi(\mathit{Next}^i(\mathfrak{S}\,[\![K_P]\!]\,(s_0)))$. Thus, we have that for all $i \geq 0$, there exists a $\bar{\sigma}$ such that $\sigma = s_0 \dots s_i\bar{\sigma}$ and $\bar{\sigma} \in \alpha_\psi(\mathit{Next}^{i+1}(\mathfrak{S}\,[\![K_P]\!]\,(s_0)))$. It follows that $\sigma \in \alpha_{\mathbf{EG}\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s))$ and $\alpha_{\mathbf{EG}\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$.

—$K_P, s \models \mathbf{E}[\psi\mathbf{U}\psi']$
Then, by the definition of $\mathbf{EU}$, there exists $k \geq 0$ such that a trace $\sigma = s_0 s_1 \dots s_k \dots$, with $s_0 = s$, satisfies the following properties: $\forall 0 \leq i < k \,.\, K_P, s_i \models \psi$ and $K_P, s_k \models \psi'$. By the induction hypothesis, it follows that $\forall 0 \leq i < k \,.\, \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_i)) \neq \emptyset$ and $\alpha_{\psi'}(\mathfrak{S}\,[\![K_P]\!]\,(s_k)) \neq \emptyset$. As $\alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_i)) = \alpha_\psi(\mathit{Next}^i(\mathfrak{S}\,[\![K_P]\!]\,(s_0)))$ and $s = s_0$, it then follows that $\alpha_{\mathbf{E}[\psi\mathbf{U}\psi']}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$.

—$K_P, s \models \neg\psi$
Then $K_P, s \not\models \psi$, implying that $\alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s)) = \emptyset$. We have $\alpha_{\neg\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s)) = A \setminus \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s)) = A$, which is indeed non-empty.

—$K_P, s \models \psi \wedge \psi'$
There exists $\sigma = s\sigma'$ such that both ψ and ψ' are satisfied. As $K_P, s \models \psi$ and $K_P, s \models \psi'$, then $\sigma \in \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s))$ and $\sigma \in \alpha_{\psi'}(\mathfrak{S}\,[\![K_P]\!]\,(s))$. Thus, $\alpha_{\psi\wedge\psi'}(\mathfrak{S}\,[\![K_P]\!]\,(s)) = \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s)) \cap \alpha_{\psi'}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \supseteq \{\sigma\} \neq \emptyset$.

—$K_P, s \models \mathtt{instr}(p_1, \ldots, p_n)$
There exists $\sigma = s\sigma'$ such that $\mathtt{instr}(p_1, \ldots, p_n)$ is satisfied by s. Then we have that $\mathit{act}\,[\![s]\!] = \mathtt{instr}(p_1, \ldots, p_n)$, implying that $\alpha_{\mathtt{instr}(p_1,\ldots,p_n)}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$.

—$K_P, s \models \#\mathrm{loc}(L)$
There exists $\sigma = s\sigma'$ such that $\#\mathrm{loc}(L)$ is satisfied by s. Then $\mathit{lab}\,[\![\mathit{cmd}\,[\![\{s\}]\!]]\!] = \#\mathrm{loc}(L)$, implying that $\alpha_{\#\mathrm{loc}(L)}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$.

($\Leftarrow$) Assume $\alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$. We perform the proof by induction on the structure of ψ.

—$\alpha_{\mathbf{EX}\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that $\sigma = s\sigma'$ and $\sigma' \in \alpha_\psi(\mathit{Next}^1(\mathfrak{S}\,[\![K_P]\!]\,(s)))$. This implies that $\sigma' \in \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_1))$. Then, by the induction hypothesis, $K_P, s_1 \models \mathbf{E}\psi$. Thus, $K_P, s \models \mathbf{EX}\psi$.

—$\alpha_{\mathbf{EG}\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that for all $k \geq 0$, we have $\sigma = s_0 \ldots s_k\sigma'$ and $\sigma' \in \alpha_\psi(\mathit{Next}^{k+1}(\mathfrak{S}\,[\![K_P]\!]))$. Note that $s_0 = s$ and let $\sigma' = s_{k+1}\sigma''$. It follows that $\sigma' \in \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_{k+1}))$. Then, by the induction hypothesis, $K_P, s_{k+1} \models \psi$. We now have that $\forall k \geq 0\ .\ K_P, s_{k+1} \models \psi$. Thus, $K_P, s \models \mathbf{EG}\psi$.

—$\alpha_{\mathbf{E}[\psi\mathbf{U}\psi']}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that for some $k \geq 0$, we have the following properties hold: $\alpha_\psi(\mathit{Next}^j(\mathfrak{S}\,[\![K_P]\!])) \neq \emptyset$, for $0 \leq j < k$, and $\alpha_{\psi'}(\mathit{Next}^k(\mathfrak{S}\,[\![K_P]\!])) \neq \emptyset$. It follows that $\alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s_j)) \neq \emptyset$, for $0 \leq j < k$. By the induction hypothesis, $K_P, s_j \models \psi$ for $0 \leq j < k$ and $K_P, s_k \models \psi'$. Thus, $K_P, s \models \mathbf{E}[\psi\mathbf{U}\psi']$.

—$\alpha_{\neg\psi}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that $\sigma \notin \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s))$. Then $K_P, s \not\models \psi$, thus $K_P, s \models \neg\psi$.

—$\alpha_{\psi\wedge\psi'}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that $\sigma \in \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]\,(s))$ and $\sigma \in \alpha_{\psi'}(\mathfrak{S}\,[\![K_P]\!]\,(s))$. By induction hypothesis, it follows that $K_P, s \models \psi$ and $K_P, s \models \psi'$. Thus, $K_P, s \models \psi \wedge \psi'$.

—$\alpha_{\mathtt{instr}(p_1,\ldots,p_n)}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that $\sigma = s\sigma'$ and $\mathit{act}\,[\![s]\!] = \mathtt{instr}(p_1, \ldots, p_n)$. Then s satisfies $\mathtt{instr}(p_1, \ldots, p_n)$, thus $K_P, s \models \mathtt{instr}(p_1, \ldots, p_n)$.

—$\alpha_{\#\mathrm{loc}(L)}(\mathfrak{S}\,[\![K_P]\!]\,(s)) \neq \emptyset$
There exists σ in $\mathfrak{S}\,[\![K_P]\!]\,(s)$ such that $\sigma = s\sigma'$ and $\mathit{lab}\,[\![\mathit{cmd}\,[\![\{s\}]\!]]\!] = L$. Then s satisfies $\#\mathrm{loc}(L)$, thus $K_P, s \models \#\mathrm{loc}(L)$.

which concludes the proof. □

As a corrolary, $K_P \models \psi \iff \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]) \neq \emptyset$ follows from the fact that $\mathfrak{S}\,[\![P]\!] = \bigcup_{s_0 \in \mathit{Init}} \mathfrak{S}\,[\![P]\!]\,(s_0)$. We note that Proposition 7 holds under the assumption that the oracles OR_{CTPL} (which provides ψ) and OR_{CFG} (which contructs K_P) are perfect.

Step 1: Design an Equivalent Trace-Based Detector. We can now define a trace-based detector D_{CTPL} based on the abstractions α_ψ and α_{Cmd}.

DEFINITION 14. A program P is ψ-malicious if $\alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])) \neq \emptyset$.

Based on this definition, we can show that D_{CTPL} and $\mathcal{A}_{CTPL}$ are equivalent. By Proposition 7 and its corollary, we have that $K_P \models \psi \iff \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]) \neq \emptyset$. But $\mathfrak{S}\,[\![K_P]\!] = \alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])$ by the construction of K_P from P. It follow immediately that $K_P \models \psi \iff \alpha_\psi(\mathfrak{S}\,[\![K_P]\!]) \neq \emptyset \iff \alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])) \neq \emptyset$.

Step 2: Check for Soundness and Completeness. Let $\mathbb{O}_{CTPL}$ be the set of obfuscations against which D_{CTPL} is complete. We identify several classes of obfuscations that are members of $\mathbb{O}_{CTPL}$ and show that $\mathbb{O}_c \not\subseteq \mathbb{O}_{CTPL}$. As a first step, we prove that D_{CTPL} is sound if the oracle OR_{CTPL} returns an ideal CTPL formula.

PROPOSITION 8. *D_{CTPL} is oracle-sound with respect to every obfuscation.*

PROOF. Given a malware M, the perfect oracle OR_{CTPL} returns a CTPL formula ψ such that $K_P \models \psi \Rightarrow M \hookrightarrow P$. As $K_P \models \psi \iff \alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!]))$, it follows that $D_{CTPL}(P) = D_{CTPL}(M) \Rightarrow M \hookrightarrow P$ and thus D_{CTPL} is sound relative to the oracle OR_{CTPL}. □

We are interested in the completeness of D_{CTPL} with respect to various obfuscations. In other words, as D_{CTPL} is complete with respect to $\mathbb{O}_{CTPL}$, we wish to discover the obfuscation classes members of $\mathbb{O}_{CTPL}$. Let us consider the set of conservative obfuscations $\mathbb{O}_c$.

The model checking-based malware detector is resilient to the code reordering obfuscation $\mathcal{O}_J$ and the opaque predicate insertion obfuscations $\mathcal{O}_T$ and $\mathcal{O}_U$. The code reordering obfuscation $\mathcal{O}_J$ relabels commands and inserts jump instructions. Because α_ψ by construction does not take into account labels, only actions (i.e., instructions), and because the inserted jump actions do not affect the operation of D_{CTPL}, it follows that $\mathcal{O}_J \in \mathbb{O}_{CTPL}$. Through similar reasoning we can show that $\mathcal{O}_T \in \mathbb{O}_{CTPL}$ and $\mathcal{O}_U \in \mathbb{O}_{CTPL}$.

Unfortunately, D_{CTPL} is not complete with respect to all conservative obfuscations. From the conservative obfuscations we discussed in Section 6, semantic nop insertion $\mathcal{O}_N$ and substitution of equivalent commands $\mathcal{O}_I$ can produce obfuscated program variants that evade detection by D_{CTPL}. For example, $\mathcal{O}_I$ will substitute commands in a program such that $\alpha_{Cmd}(\sigma) \neq \alpha_{Cmd}(\sigma')$, where $\sigma \in \mathfrak{S}\,[\![P]\!]$ and $\sigma' \in \mathfrak{S}\,[\![\mathcal{O}_I(P')]\!]$. Then α_ψ applied to σ' will fail to filter actions of interest. Let us define *syntactically conservative obfuscations* $\mathbb{O}_{c!}$ as the set of conservative obfuscations that preserve commands, i.e., $\mathcal{O} : \mathbf{P} \to \mathbf{P}$ is a syntactically conservative obfuscation if $\mathcal{O}$ is conservative and $\forall \alpha \in \mathfrak{S}\,[\![P]\!], \exists \delta \in \mathfrak{S}\,[\![\mathcal{O}(P)]\!] : \alpha_{Cmd}(\sigma) \preceq \alpha_{Cmd}(\delta)$.

PROPOSITION 9. $\mathbb{O}_{c!} \subseteq \mathbb{O}_{CTPL}$.

PROOF. Consider a syntactically conservative obfuscation $\mathcal{O} \in \mathbb{O}_{c!}$. From the definition of $\mathbb{O}_{c!}$, we have that $\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!]) \preceq \alpha_{Cmd}(\mathfrak{S}\,[\![\mathcal{O}(P)]\!])$. Hence, it follows that:

$$\alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])) \preceq \alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![\mathcal{O}(P)]\!])). \tag{4}$$

If $D_{CTPL}(P) = 1$, then that $\alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])) \neq \emptyset$. Equation 4 implies:

$$\alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![P]\!])) \preceq \alpha_\psi(\alpha_{Cmd}(\mathfrak{S}\,[\![\mathcal{O}(P)]\!])) \neq \emptyset,$$

and we have $D_{CTPL}(\mathcal{O}\,[\![P]\!]) = 1$. Thus, $\mathcal{O} \in \mathbb{O}_{c!}$. □

Furthermore, the set of obfuscations $\mathbb{O}_{CTPL}$ is strictly larger than $\mathbb{O}_{c!}$ as it contains at least one non-conservative obfuscation, the variable-renaming obfuscation $\mathcal{O}_v$ of Section 7. Variable renaming works by changing variables consistently throughout the program, according to a map $\pi \in \Pi$ that relates names of variables between the original program and the obfuscated program, $\pi : var[\![P]\!] \to Names$. In the CTPL notation, $\mathcal{U}$ is the finite universe of variable names, and thus we have $var[\![P]\!] \subseteq \mathcal{U}$ and $Names = \mathcal{U}$. Let ϕ be a CTL formula (i.e., without free variables). We can show that if a model K satisfies ϕ, then a renamed model $K[x/v]$ (where the variable x was renamed to v) satisfies $\phi[x/v]$ (where are similar renaming from x was v applied). This observation follows directly from the fact that satisfaction in CTPL reduces to syntactic equality between actions in traces ($act[\![s]\!]$) and CTPL formula predicates ($\texttt{instr}(p_1, \ldots, p_n)$). Consider a program P such that $D_{CTPL}(P) = 1$ for some CTPL formula ψ. Let π be a map of a renaming obfuscation $\mathcal{O}_v^\pi$ applied to P. We wish to establish that the obfuscated malware $\mathcal{O}_v^\pi[\![P]\!]$ is indeed detected by D_{CTPL} as malicious. We know that $K_P \models \psi$, i.e., $K_P \models (\bigvee_{(v_1,\ldots,v_k)\in\mathcal{U}^k} \psi[x_i/v_i])$. This is equivalent to $\bigvee_{(v_1,\ldots,v_k)\in\mathcal{U}^k} K_P \models \psi[x_i/v_i]$. As a result, there exists a renaming $(v_1, \ldots, v_k) \in \mathcal{U}^k$ such that $K_P \models \psi[x_i/v_i]$. Note that $\psi[x_i/v_i]$ is a CTL formula. According to the observation above, renaming preserves the satisfaction relation on CTL formulas, and we have $K_P \models \psi[x_i/v_i] \iff K_P[v_i/\pi(v_i)] \models (\psi[x_i/v_i])[v_i/\pi(v_i)]$. As a result, $K_P[v_i/\pi(v_i)] \models \psi[x_i/\pi(v_i)]$ (in CTL), which implies that $K_P[v_i/\pi(v_i)] \models \psi$ (in CTPL). But $K_P[v_i/\pi(v_i)]$ is the model for $\mathcal{O}_v^\pi[\![P]\!]$, meaning that $D_{CTPL}(\mathcal{O}_v^\pi[\![P]\!]) = 1$. Thus, $\mathbb{O}_v \in \mathbb{O}_{CTPL}$.

10. RELATED WORK

Code obfuscation has been extensively studied in the context of protecting the intellectual property of programs. The goal of an obfuscation technique is to transform a program in order to make it harder (ideally impossible) to reverse engineer while preserving its functionality [Collberg et al. 1997; 1998; Chow et al. 2001; Linn and Debray 2003; Dalla Preda and Giacobazzi 2005; Dalla Preda and Giacobazzi 2005]. Cryptographers are also pursuing research on the question of possibility of obfuscation [Barak et al. 2001; Wee 2005; Goldwasser and Kalai 2005].

An introduction to theoretical computer virology can be found in [Cohen 1985]. In particular, Cohen proposes a formal definition of computer virus based on Turing's model of computation, and proves that precise virus detection is undecidable [Cohen 1987], namely that there is no algorithm that can reliably detect all viruses. Cohen shows also that the detection of evolutionary variants of viruses is undecidable, namely that metamorphic malware detection is undecidable [Cohen 1989]. A related undecidability result is the one presented in [Chess and White 2000], where the authors prove the existence of a virus type that cannot be detected. Adleman applies formal computability theory to viruses and viruses detection, showing that the problem is intractable [Adleman 1988].

Despite these results, proving that in general viruses detection is impossible, it is possible to develop ad-hoc detection schemes that work for specific viruses (malware). In fact, there is a considerable body of literature on techniques for malware detection [Ször 2005]. These techniques base their detection on syntactic elements of the program. As argued earlier, malware writers often resort to metamorphism in order to avoid such syntactic detection. In particular, code obfuscation is successfully used by hackers to confuse the misuse-detection schemes that are sensitive to slight modifications of program syntax.

Some attempts to create obfuscation-resilient schemes for identifying malware are not practical as they suffer from high false positive rates. In particular, *anomaly detection* algorithms are based on a notion of normal program behavior and classify as malicious any behavior deviating from normality [McHugh 2001]. Anomaly detection does not need any a priori knowledge of the malicious code and can therefore detect previously unseen malware. Due to the difficulty of classifying what is normal, this technique usually produces many false alarms (systems often exhibit unseen or unusual behaviors that are not malicious). For example anomaly detection using statistical methods suffers from such limitations [Li et al. 2005; Kolter and Maloof 2004]. Other approaches can only provide a post-infection forensic capability – as for example correlation of network events to detect propagation after infection [Gupta and Sekar 2003].

With the advent of metamorphic malware, the malware detection community has begun to face the above mentioned theoretical limits and to develop detection systems based on formal methods of program analysis. We agree with Lakhotia and Singh, who state that *"formal methods for analyzing programs for compilers and verifiers when applied to anti-virus technologies are likely to produce good results for the current generation of malicious code"* [Lakhotia and Singh 2000]. In the following we briefly present some of the existing approaches to malware detector based on formal methods.

Program Semantics. Christodorescu and Jha observe that the main deficiency of misuse detection is its purely syntactic nature, that ignores the meaning of instructions, namely their semantics [Christodorescu et al. 2005]. Following this observation, they propose an approach to malware detection that considers the malware semantics, namely the malware behavior, rather than its syntax. Malicious behavior is described through a template, namely a generalization of the malicious code that expresses the malicious intent while eliminating implementation details. The idea is that a template does not distinguish between irrelevant variants of the same malware obtained through obfuscation processes. For example, a template uses symbolic variable/constants to handle variable and register renaming, and it is related to the malware control flow graph in order to deal with code reordering. Then, they propose an algorithm that verifies if a program presents the template behavior, using some unification process between program variables/constants and malware symbolic variables/constants. This detection approach is able to handle a limited set of obfuscations commonly used by malware writers.

Static Analysis. Bergeron *et al.* propose a malware-detection scheme based on the detection of suspicious system call sequences [Bergeron et al. 2001]. In particular, they consider a reduction (subgraph) of the program control flow graph, which contains only the nodes representing certain system calls. Next they check if such subgraph presents known malicious sequences of system calls.

Christodorescu and Jha describe a malware detection system based on language containment and unification [Christodorescu and Jha 2003]. The malicious code and the possibly infected program are modeled as automata using unresolved symbols and placeholders for registers to deal with some types of obfuscations. In this setting, a program presents a malicious behavior if the intersection between the language of the malware automaton and the one of the program automaton is not empty.

Model Checking. Singh and Lakhotia specify malicious behaviors through a formula in linear temporal logic (LTL), and then use the model checker SPIN to check if this property is satisfied by the control flow graph of a suspicious program [Singh and Lakhotia 2003].

Kinder *et al.* introduce a new temporal logic CTPL (Computation Tree Predicate Logic), which is an extension of the branching time temporal logic CTL, that takes into account register renaming, allowing a succinct and natural presentation of malicious code patterns [Kinder et al. 2005]. They develop a model checking algorithm for CTPL that, checking if a program satisfies a malware property expressed by a CTPL formula, verifies if the program is infected by the considered malicious behavior.

Model checking techniques have recently been used also in worm quarantine applications [Briesemeister et al. 2005]. Worm quarantine techniques seek to dynamically isolate the infected population from the population of uninfected systems, in order to fight malware infection.

Program Slicing. Lo *et al.* develop a programmable static analysis tool, called MCF (Malicious Code Filter) [Lo et al. 1995], that uses program slicing and flow analysis to detect malicious code. Their approach relies on *tell-tale signs*, namely on program properties that characterize the maliciousness of a program. MCF slices the program with respect to these tell-tale signs in order to get a smaller program segment that might perform malicious actions. These segments are further analyzed in order to determine the existence of a malicious behavior.

Data Mining. Data mining techniques try to discover new knowledge in large data collections. In particular, data mining identifies hidden patterns and trends that a human would not be able to discover efficiently on large databases, employing, for example, machine learning and statistical analysis methods. Lee *et al.* study ways to apply data mining techniques to intrusion detection [Lee et al. 2000; Lee and Stolfo 1998; Lee et al. 1999]. The basic idea is to use data mining techniques to identify patterns of relevant system features, describing program and user behavior, in order to recognize both anomalies and known malicious behaviors.

11. 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 work 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. For example, in this framework, it is possible to show that the signature-based malware detector is generally sound but not complete, as well as that the semantics-aware malware detector proposed by Christodorescu *et al.* is complete with respect to some commonly used malware obfuscations, and that the model checking-based malware detector of Kinder *et al.* is generally sound while it is complete only for certain obfuscations. Our framework uses a trace semantics to characterize the behaviors of both the malware and the program being analyzed. It shows how we can get around the effects of obfuscations by using abstract interpretation to “hide” irrelevant aspects of these behaviors. Thus, given an obfuscating transformation $\mathcal{O}$, the key point is to characterize the proper semantic abstraction that recognizes infection even if the malware is obfuscated through $\mathcal{O}$.

So far, given an obfuscating transformation $\mathcal{O}$, we assume that the proper abstraction α, which discards the details changed by the obfuscation and preserves maliciousness, is provided by the malware detector designer. We are currently investigating how to design a systematic (ideally automatic) methodology for deriving an abstraction α that leads to a sound and complete semantic malware detector. As a first step in this direction, we observe that if abstraction α 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. This, characterization may help us in the design of suitable abstractions that are able to deal with a given obfuscation.

Other approaches to the automatic design of abstraction α can rely on *monitoring* malware execution in order to extract its malicious behaviors, i.e., the set of malicious (abstract) traces that characterizes the malign intent. The idea is that every time that a malware exhibits a malicious intent (for example every time it violates some security policies) the behavior is added to the set of malicious ones. Another possibility we are interested in is the use of *data mining* techniques to extract maliciousness in malware behaviors. Preliminary work in this area has shown that empirical data mining techniques can successfully identify behavior that is unique to a malware [Christodorescu et al. 2007]. In this case, given a sufficient wide class of malicious variants we can analyze their semantics and use data mining to extract common features.

For future work in designing malware detectors, an area of great promise is that of detectors that focus on interesting actions. Depending on the execution environment, certain states are reachable only through particular actions. For example, system calls are the only way for a program to interact with OS-mediated resources such as files and network connections. If the malware is characterized by actions that lead to program states in an unique, unambiguous way, then all applicable obfuscation transformations are conservative. As we showed, a semantic malware detector that is both sound and complete for a class of conservative obfuscations exists, if an appropriate abstraction can be designed. In practice, such an abstraction cannot be precisely computed, due to undecidability of program trace semantics – a future research task is to find suitable approximations that minimize false positives while preserving completeness.

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 [Kinder et al. 2005]. Observe that abstraction α actually defines a set of program traces that are equivalent up to $\mathcal{O}$. In model checking, sets of program traces are represented by formulas of some linear/branching temporal logic. Hence, we aim at defining a temporal logic whose formulas are able to express normal forms of obfuscations together with operators for composing them. This would allow us to use standard model checking algorithms to detect malware 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

ADLEMAN, L. M. 1988. An abstract theory of computer viruses. In *Proceedings of Advances in cryptology* (*CRYPTO'88*). LNCS, vol. 403. Springer, Berlin/Heidelberg.

BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. 2001. On the (im)possibility of obfuscating programs. In *Advances in Cryptology (CRYPTO'01)*. Lecture Notes in Computer Science, vol. 2139. Springer, Santa Barbara, CA, USA, 1 – 18.

BERGERON, J., DEBBABI, M., DESHARNAIS, J., ERHIOUI, M. M., LAVOIE, Y., AND TAWBI, N. 2001. Static detection of malicious code in executable programs. In *Symposium on Requirements Engineering for Infor-*

mation Security. Published online, `http://www.sreis.org/old/2001/index.html` (last accessed on May 31, 2007).

BRIESEMEISTER, L., PORRAS, P. A., AND TIWARI, A. 2005. Model checking of worm quarantine and counter-quarantine under a group defense. Tech. Rep. SRI-CSL-05-03, SRI International, Computer Science Laboratory.

CHESS, D. AND WHITE, S. 2000. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference (VB2000)*. Virus Bulletin, Orlando, FL, USA.

CHOW, S., GU, Y., JOHNSON, H., AND ZAKHAROV, V. 2001. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of the 4th International Information Security Conference (ISC'01)*, G. Davida and Y. Frankel, Eds. Lecture Notes in Computer Science, vol. 2200. Springer, Malaga, Spain, 144–155.

CHRISTODORESCU, M. AND JHA, S. 2003. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium* (*Security '03*). USENIX Association, Berkeley, CA, USA, 169–186.

CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. 2007. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07)*. To appear, Dubrovnik, Croatia, pages TBD.

CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. 2005. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE Computer Society, Los Alamitos, CA, USA, 32–46.

CHRISTODORESCU, M., KINDER, J., JHA, S., KATZENBEISSER, S., AND VEITH, H. 2005. Malware normalization. Tech. Rep. 1539, University of Wisconsin, Madison, Wisconsin, USA. Nov.

CLARKE, JR., E. M., GRUMBERG, O., AND PELED, D. A. 2001. *Model Checking*. The MIT Press, Cambridge, MA, USA.

COHEN, F. 1985. Computer viruses. Ph.D. thesis, University of Southern California.

COHEN, F. 1989. Computational aspects of computer viruses. *Computers and Security 8,* 4, 325.

COHEN, F. B. 1987. Computer viruses: Theory and experiments. *Computers and Security 6*, 22–35.

COLLBERG, C., THOMBORSON, C., AND LOW, D. 1997. A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Sciences, The University of Auckland. July.

COLLBERG, C., THOMBORSON, C., AND LOW, D. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, San Diego, CA, USA, 184–196.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, Los Angeles, CA, USA, 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'79)*. ACM Press, San Antonio, TX, USA, 269–282.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation frameworks. *Journal of Logic and Computation 2,* 4 (Aug.), 511–547.

COUSOT, P. AND COUSOT, R. 2002. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM Press, Portland, OR, USA, 178–190.

DALLA PREDA, M., CHRISTODORESCU, M., JHA, S., AND DEBRAY, S. 2007. A semantics-based approach to malware detection. In *Proceedings of the 32nd ACM Symp. on Principles of Programming Languages* (*POPL '07*). ACM Press, Nice, France, 377–388.

DALLA PREDA, M. AND GIACOBAZZI, R. 2005. Control code obfuscation by abstract interpretation. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*. IEEE Computer Society, Los Alamitos, CA, USA, 301–310.

DALLA PREDA, M. AND GIACOBAZZI, R. 2005. Semantics-based code obfuscation by abstract interpretation. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05)*. Lecture Notes in Computer Science, vol. 3580. Springer, Lisboa, Portugal, 1325–1336.

DETRISTAN, T., ULENSPIEGEL, T., MALCOM, Y., AND VON UNDERDUK, M. S. 2003. Polymorphic shellcode engine using spectrum analysis. *Phrack 11,* 61 (Aug.), published online at `http://www.phrack.org` (last accessed on Jan. 16, 2004).

GOLDWASSER, S. AND KALAI, Y. T. 2005. On the impossibility of obfuscation with auxiliary input. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*. IEEE Computer Society, Washington, DC, USA, 553–562.

GUPTA, A. AND SEKAR, R. 2003. An approach for detecting self-propagating email using anomaly detection. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID'03),* G. Vigna, E. Jonsson, and C. Kruegel, Eds. Lecture Notes in Computer Science, vol. 2820. Springer, Pittsburgh, PA, USA, 55–72.

INTEL CORPORATION. 2001. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation.

JORDAN, M. 2002. Dealing with metamorphism. *Virus Bulletin 2002,* 10 (Oct.), 4–6.

KINDER, J., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. 2005. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05),* K. Julisch and C. Krügel, Eds. Lecture Notes in Computer Science, vol. 3548. Springer, Vienna, Austria, 174–187.

KOLTER, J. Z. AND MALOOF, M. A. 2004. Learning to detect malicious executables in the wild. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*. ACM Press, Seattle, WA, USA, 470–478.

LAKHOTIA, A. AND MOHAMMED, M. 2004. Imposing Order on Program Statements to Assist Anti-Virus Scanners. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*. IEEE Computer Society, Washington, DC, USA, 161–170.

LAKHOTIA, A. AND SINGH, P. K. 2000. Challenges in getting "formal" with viruses. In *Virus Bulletin*. Virus Bulletin Ltd., Abingdon, England.

LEE, W., NIMBALKAR, R. A., YEE, K. K., PATIL, S. B., DESAI, P. H., TRAN, T. T., AND STOLFO, S. J. 2000. A data mining and CIDF based approach for detecting novel and distributed intrusions. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection (RAID 2000)*. LNCS, vol. 1907. Springer, Berlin/Heidelberg, 49–65.

LEE, W. AND STOLFO, S. 1998. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 79–93.

LEE, W., STOLFO, S., AND MOK, K. W. 1999. A data mining framework for building intrusion detection models. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P'99)*. IEEE Computer Society, Los Alamitos, CA, USA, 120–132.

LI, W.-J., WANG, K., STOLFO, S. J., AND HERZOG, B. 2005. Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Systems, Man, and Cybernetics (SMC) Workshop on Information Assurance (IAW'05)*. United States Military Academy, IEEE Computer Society, West Point, NY, 64–71.

LINN, C. AND DEBRAY, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM Press, Washington, DC, USA, 290–299.

LO, R. W., LEVITT, K. N., AND OLSSON, R. A. 1995. Mcf: A malicious code filter. *Computers & Security 14,* 541–566.

MCHUGH, J. 2001. Intrusion and intrusion detection. *International Journal of Information Security 1,* 1, 14–35.

MORLEY, P. 2001. Processing virus collections. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*. Virus Bulletin, Prague, Czech Republic, 129–134.

NACHENBERG, C. 1997. Computer virus-antivirus coevolution. *Communications of the ACM 40,* 1 (Jan.), 46–51.

RAJAAT. 1999. Polymorphism. *29A Magazine 1,* 3, 1–2.

SINGH, P. AND LAKHOTIA, A. 2003. Static verification of worm and virus behaviour in binary executables using model checking. In *Proceedings of the 4th IEEE Information Assurance Workshop*. IEEE Computer Society, Los Alamitos, CA, USA.

SYMANTEC CORPORATION. 2006. *Symantec Internet Security Threat Report: Trends for January 06–June 06*. Vol. X. Symantec Corporation, Cupertino, CA, USA.

SZÖR, P. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Boston, MA, USA.

SZÖR, P. AND FERRIE, P. 2001. Hunting for metamorphic. In *Proceedings of the 2001 Virus Bulletin Conference (VB2001)*. Virus Bulletin, Prague, Czech Republic, 123 – 144.

WALENSTEIN, A., MATHUR, R. CHOUCHANE, M. R., AND, LAKHOTIA, A 2006. Normalizing Metamorphic Malware Using Term Rewriting. In *Proceedings of the 6th International Workshop on Source Code Analysis and Manipulation (SCAM'06)*. 75–84, IEEE Computer Society Press.

WEE, H. 2005. On obfuscating point functions. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)*. ACM Press, Baltimore, MD, USA, 523–532.

Z0MBIE. 2001a. Automated reverse engineering: Mistfall engine. Published online at `http://www.madchat.org//vxdevl/papers/vxers/Z0mbie/autorev.txt` (last accessed on Sep. 29, 2006).

Z0MBIE. 2001b. Real Permutating[sic] Engine. Published online at `http://vx.netlux.org/vx.php?id=er05` (last accessed on Sep. 29, 2006).

Received July 2007; accepted October 2007